UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO

BRUNO TUNES DE MELLO

SISTEMA DE NAVEGAÇÃO EMBARCADO DE TEMPO REAL PARA UM HELICÓPTERO MINIATURA AUTÔNOMO

VITÓRIA – ES
SETEMBRO/2014

BRUNO TUNES DE MELLO

SISTEMA DE NAVEGAÇÃO EMBARCADO DE TEMPO REAL PARA UM HELICÓPTERO MINIATURA AUTÔNOMO

Parte manuscrita do Projeto de Graduação do aluno **Bruno Tunes de Mello**, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Orientador: Prof. Dr. Mário Sarcinelli Filho
Coorientador: Prof. Msc. Lucas Vago Santana

VITÓRIA – ES
SETEMBRO/2014

BRUNO TUNES DE MELLO

SISTEMA DE NAVEGAÇÃO EMBARCADO DE TEMPO REAL PARA UM HELICÓPTERO MINIATURA AUTÔNOMO

Parte manuscrita do Projeto de Graduação do aluno **Bruno Tunes de Mello**, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Aprovada em 16 de Setembro de 2014.

COMISSÃO EXAMINADORA:

Prof. Dr. Mário Sarcinelli Filho
Universidade Federal do Espírito Santo
Orientador

Prof. Msc. Lucas Vago Santana
Instituto Federal do Espírito Santo – Campus Linhares
Coorientador

Prof. Dr. Raquel Frizera Vassalo
Universidade Federal do Espírito Santo
Examinadora

Prof. Dr. Evandro Ottoni Teatini Salles
Universidade Federal do Espírito Santo
Examinador

Agradecimentos

Agradeço à Deus pelo dom da vida, por guiar os meus caminhos e pelas oportunidades com as quais fui abençoado.

Agradeço à minha mãe, Sueli, pela dedicação e apoio que me deram suporte para concluir mais uma etapa em minha vida. À minha irmã Ana Karla pelo incentivo e pelos conselhos que muito me ajudaram. Ao meu pai José Carlos por me ensinar a valorizar o estudo e o esforço.

Ao meu orientador Prof. Dr. Mário Sarcinelli Filho pela oportunidade de conhecer este grupo de pesquisa, primeiro como aluno de Iniciação Científica e agora no desenvolvimento deste projeto de graduação. Ao meu coorientador Prof. Msc. Lucas Vago Santana pelas orientações, motivações e conselhos que foram fundamentais para o progresso deste trabalho. Ao companheiro de pesquisa Prof. Msc. Igor Pizetta por acompanhar o meu trabalho enquanto aluno de Iniciação Científica e me ensinar vários aspectos deste projeto.

Aos companheiros da graduação e do Laboratório de Automação Inteligente 1 e 2 pelos momentos de conversa, estudo e descontração que foram importantes para renovar as forças. E aos professores e servidores do Departamento de Engenharia Elétrica pelo empenho em suas atividades.

A todos vocês, muito obrigado.

Resumo

Devido à grande variedade de aplicações, a navegação autônoma de veículos aéreos não tripulados tem sido objeto de estudo em diversos grupos de pesquisas ao redor do mundo. Neste contexto, este projeto de graduação consiste em desenvolver um sistema de navegação autônoma para um helicóptero de escala reduzida. O sistema projetado terá um computador de bordo portando um sistema operacional Linux com suporte a recursos de tempo real, de forma que o processamento dos algoritmos de controle esteja dentro dos prazos definidos pelas restrições de tempo. Um sistema de instrumentação baseado em sensores inerciais e visão computacional será usado para determinar a posição e orientação do veículo no espaço tridimensional. Os dados inerciais serão lidos de uma unidade de medição inercial e filtrados através de um método de fusão sensorial. Uma câmera a bordo da aeronave será usada para capturar imagens que, processadas, irão permitir inferência a respeito de deslocamentos bidimensionais do veículo.

Lista de Figuras

Lista de Tabelas

Sumário

1 INTRODUÇÃO

1.1 Apresentação

A robótica móvel é uma área da tecnologia em grande evolução, presente em diversas aplicações, com o emprego em atividades de pesquisa, industriais e militares. Através dela, podem-se realizar tarefas que seriam impossíveis ou de alto risco para pessoas, como a exploração de petróleo e gás em águas profundas, missões espaciais, supervisão e atuação em ambientes com qualquer tipo de contaminação (como os radioativos) ou de atmosfera explosiva, inspeção interna de instalações (como dutos) e até mesmo tarefas de rotina como transporte de cargas em locais internos e limpeza (SIEGWART et al., 2011, p. 1—5).

Robôs móveis são máquinas construídas sobre estruturas não fixas, com um sistema de locomoção, como rodas, esteiras, pernas, propulsores, asas ou pás rotativas, que lhe confere mobilidade dentro do ambiente no qual está inserido. Estes robôs podem ser autônomos, quando existe um conjunto de sensores que confere capacidade de percepção do ambiente e localização no espaço para a sua navegação ou podem ser teleoperados, quando um operador localizado remotamente controla o posicionamento e ações do veículo (SIEGWART et al., 2011, p. 2).

Dentro deste campo científico, encontram-se os robôs móveis aéreos, cuja pesquisa e desenvolvimento têm crescido nos últimos anos. Estes robôs são veículos com capacidade de realizar navegação autônoma, executando as tarefas designadas e são denominados de UAV (do inglês Unmanned Aerial Vehicles) ou VANT (do termo Veículo Aéreo Não Tripulado). Robôs aéreos são os mais apropriados para atividades em que se necessite cobrir grandes áreas, como em plantações que envolvem agricultura de precisão (como a pulverização eficiente de pesticidas), supervisão do espaço aéreo e tráfego urbano, monitoramento de áreas de proteção ambiental (como desmatamento de florestas) e da poluição (medição de níveis de partículas e componentes no ar), intervenção em ambientes hos-

tis (como atmosferas radioativas, áreas incendiadas e vulcões em erupção), supervisão de infraestruturas como linhas de transmissão e dutos de gás e óleo, além das operações militares de reconhecimento de terrenos e vigilância (BRANDÃO, 2013, p. 63).

Os VANTs podem ser tanto mais leves que o ar, como os balões e os dirigíveis, quanto mais pesados que o ar, cujos exemplos podem ser classificados em aeronaves de asas fixas e de pás rotativas. Aviões e planadores são exemplos do tipo de asas fixas, enquanto que os helicópteros e quadrimotores são exemplos de veículos de pás rotativas (CASTILLO, 2005, p. 2).

Aeronaves de pás rotativas possuem mobilidade tridimensional maior que as de asas fixas, pois possuem a capacidade de realizar manobras multidirecionais durante voo. A sua estrutura aerodinâmica permite que os veículos decolem e aterrissem verticalmente, realizem voo pairado, desloquem-se longitudinalmente e lateralmente mantendo a mesma cota vertical, além de poder mudar de direção completamente e de deter o seu movimento de forma abrupta (CASTILLO, 2005, p. 2).

Como consequência da alta manobrabilidade, do ponto de vista de controle os VANTs de pás rotativas são mais complexos que as outras máquinas voadoras, sendo inerentemente instáveis, multivariáveis e com dinâmica altamente acoplada. Entretanto, apesar das dificuldades, a grande diversidade de aplicações tem motivado pesquisadores nas áreas de controle a desenvolverem controladores que confiram capacidade de navegação autônoma para a realização de atividades, apoiados no avanço da tecnologia de sistemas computacionais embarcados e sensores (SANTANA, 2011, p. 16).

O desenvolvimento de um sistema de navegação autônoma se concentra em projetar uma plataforma de hardware e software que seja capaz de desempenhar o processamento necessário e de um controlador que seja capaz de lidar com a dinâmica do veículo. Leitura e filtragem dos dados de sensores, além da execução dos algoritmos de controle de voo são as principais tarefas do sistema computacional, que deve obedecer às restrições relativas aos tempos de resposta estabelecidos. Em muitos trabalhos, usa-se visão computacional para tornar mais completo o sensoriamento do robô, além dos sensores inerciais que formam a base da instrumentação de veículos aéreos. Como em Phang e outros (2010), o sistema de controle se baseia em processamento de imagens. Uma câmera a bordo do

VANT gera os quadros de imagem, um algoritmo realiza a detecção do caminho, e são usados controladores PID para correção do posicionamento.

O controlador, por sua vez, deve lidar com as não linearidades do modelo matemático, além do alto acoplamento das variáveis do sistema. Os trabalhos científicos em geral, na área de VANTs de pás rotativas, propõem controladores baseados nas diversas estratégias de controle, como controle linear (geralmente PID), não-linear, controle ótimo, robusto, adaptativo, preditivo e inteligente. Alguns projetos incluem mais de uma destas técnicas de controle, na tentativa de obter um sistema mais completo e de maior eficiência. Por exemplo, em Brandão (2013) são propostos controladores não lineares tanto para um helicóptero miniatura quanto para um quadrimotor em situações de restrição de movimento (locomoção em apenas um eixo), missões de posicionamento, seguimento de trajetória e rastreamento de caminhos no espaço tridimensional.

No trabalho de Moreira e outros (2010) são utilizados controladores clássicos PID em cascata, formando multimalhas para a estabilização e controle do helimodelo. Um controlador híbrido de redes neurais e controle preditivo baseado em modelo foi utilizado em Mohammadzaheri e Chen (2007) para um helicóptero com restrição de movimento. Para melhorar o desempenho, foi usado um compensador Fuzzy do tipo Takagi-Sugeno, tornando o sistema mais rápido e menos oscilatório.

Este projeto de graduação espera contribuir para o desenvolvimento de sistemas de navegação autônoma, propondo utilizar um sistema embarcado de baixo custo e com capacidade de processamento, visto que o computador de bordo é um modelo lançado recentemente no mercado, contando com vários recursos e alta velocidade para um processador embarcado, e que, até onde se pode perceber, ainda não foi usado em pesquisas científicas com VANTs.

Deve-se destacar que muitos dos avanços realizados no projeto de veículos aéreos se deve ao progresso da eletrônica (precisamente da microeletrônica). Tanto o surgimento de sensores mais robustos quanto o lançamento no mercado de processadores que operam em maiores frequências são os principais fatores que permitem o processamento de controladores mais eficientes, uma vez que o volume de computação necessário para os cálculos é alto.

O computador de bordo em tempo real também contribuirá na continuação de trabalhos

futuros no laboratório LAI (Laboratório de Automação Inteligente) do Departamento de Engenharia Elétrica da UFES, já que o computador será usado pelos integrantes do grupo de pesquisa em seus projetos de controle de VANTs.

Neste contexto, a proposta do projeto de graduação é desenvolver um sistema embarcado de tempo real com aplicação para a navegação autônoma de um helicóptero miniatura. O computador de bordo conterá um sistema operacional Linux que será capacitado com recursos de tempo real. A instrumentação desenvolvida será composta por sensores inerciais e uma câmera a bordo do veículo, de forma a obter a posição e orientação da aeronave no espaço, para realizar o seu controle.

1.2 Objetivos

O objetivo geral deste Projeto de Graduação é desenvolver um sistema de navegação autônoma contendo um computador de bordo portando um sistema operacional baseado em Linux com suporte de tempo real que irá conferir capacidade de navegação autônoma à um helicóptero de escala reduzida através de sua posição e orientação no espaço, obtidos de dados sensoriais filtrados e visão computacional.

Os objetivos específicos do trabalho proposto são:

- Integrar funcionalidades de uma estrutura de tempo real a um sistema operacional baseado em Linux.
- Desenvolver um algoritmo de captura e processamento de imagens de uma câmera para a inferência da posição do veículo.
- Utilizar um algoritmo para a realização da leitura e filtragem de dados sensoriais inerciais.
- Realizar experimentos com um controlador e instrumentação embarcados no computador de bordo ainda fora do helicóptero, a fim de validar o computador e sua capacidade de realizar as tarefas em tempo real.

1.3 Estrutura do trabalho

Este projeto de graduação está estruturado em forma de capítulos que seguem uma ordem lógica para fornecer uma descrição completa do sistema desenvolvido.

O Capítulo 1 apresenta uma introdução ao assunto dos veículos de pás rotativas de navegação autônoma. Em seguida, é apresentada a justificativa para o estudo de tal área e os objetivos deste trabalho são descritos ao final.

O Capítulo 2 trata de uma descrição da estrutura de um helicóptero de rádio controle em miniatura, bem como a sua forma de operação. Também é descrito o sistema de navegação proposto de um forma geral, o qual tem os seus componentes detalhados nos próximos capítulos.

No Capítulo 3 trata-se do sistema de instrumentação envolvendo os sensores inerciais e o de ultrassom que, juntos, fornecem as informações à respeito da orientação e localização do veículo no espaço. Além disso, é detalhado o algoritmo de filtragem e fusão de dados utilizado e o módulo embarcado que o executa.

Em seguida, ainda se tratando da instrumentação do veículo, o Capítulo 4 descreve o sistema de visão computacional utilizado para estimar velocidades lineares do veículo através de uma câmera e de um algoritmo de fluxo óptico utilizando a biblioteca de visão computacional OpenCV.

O Capítulo 5 trata do computador de bordo do veículo, apresentado em termos de *hardware* e *software*. O capítulo também trata do uso de sistemas operacionais Linux em aplicações embarcadas e descreve o suporte de tempo real Xenomai utilizado neste projeto.

No Capítulo 6 são apresentados alguns resultados de testes experimentais com o sistema desenvolvido. Cada experimento conta, ainda, com algumas discussões.

O Capítulo 7 trata das conclusões e dos trabalhos futuros a respeito do sistema de nave-

gação.

Por fim, há dois apêndices contendo os códigos desenvolvidos para o módulo sensorial e para o computador de bordo.

2 ESTRUTURA E PRINCÍPIOS DE FUNCIONAMENTO DE UM HELIMODELO

O projeto de um sistema de navegação autônoma envolve o conhecimento dos princípios básicos de funcionamento do helicóptero bem como os aspectos de construção física e suas partes constituintes. A partir deste ponto, podem-se determinar quais recursos necessários que os componentes do sistema embarcado devem conter e como o controlador deverá ser estruturado de forma a atuar corretamente sobre o veículo.

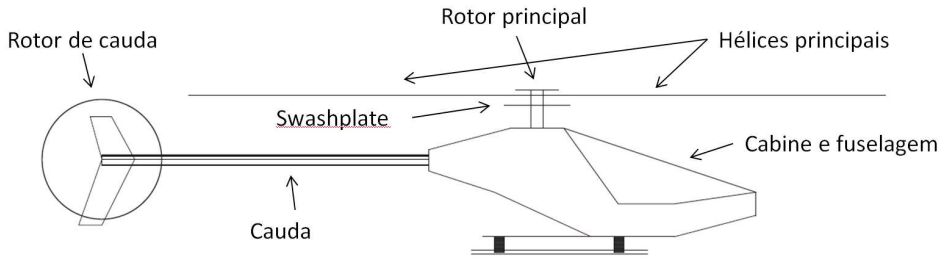
Assim, este capítulo visa apresentar os fundamentos físicos nas quais se baseia a capacidade de voar do helicóptero, além dos principais componentes presentes em um helimodelo. Também será mostrada a arquitetura da plataforma atual embarcada no veículo e o sistema embarcado proposto.

2.1 Estrutura e aerodinâmica de um helicóptero

Um helicóptero é uma aeronave que utiliza asas rotativas para prover elevação, propulsão e controle. Helimodelos de rádio controle, assim como o utilizado neste projeto, possuem partes construtivas muito similares às de um helicóptero real (SANTANA, 2011, p. 17). Além disso, o modelo matemático é o mesmo para ambos.

Um helicóptero é composto por corpo, constituído por uma fuselagem, onde se encontra a cabine, e uma estrutura de cauda. Logo acima da fuselagem se encontra o rotor principal, composto por um eixo acoplado às hélices principais e a motores que geram o movimento. Na cauda está localizado o rotor de cauda, que também apesenta hélices, com menor comprimento. A Figura 1 mostra o esquema de um helicóptero e suas partes.

Figura 1: Estrutura e partes de um helicóptero



Fonte: PIZETTA (2013, modificado pelo autor).

O rotor principal é o responsável por gerar o movimento relativo entre as pás e o ar através da rotação das asas, de forma a produzir uma força de reação. Baseia-se no princípio físico na qual o movimento das hélices tende a acelerar o ar para baixo e, de acordo com a Terceira Lei de Newton (Lei da Ação e Reação), o ar, por sua vez, exerce no veículo uma força no sentido oposto (para cima). Tal força gerada pelo ar é chamada de força de sustentação (reação do ar) (ANDERSON; EBERHARDT, 2010, p. 3–4).

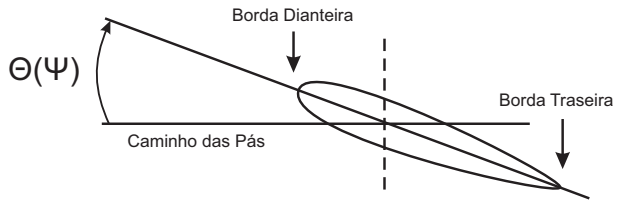
A força de sustentação existe devido ao fato do fluxo de ar (gerado pelo movimento do rotor) apresentar pressões diferentes nas superfícies superior e inferior das hélices. Isto é explicado através do Princípio de Bernoulli, que traduz o princípio de conservação de energia para fluidos. Partindo-se deste teorema, explica-se o Efeito Venturi, pelo qual quanto maior a velocidade de um fluido em um tubo menor é a sua pressão, e quanto menor a velocidade, maior é a pressão. A velocidade pode ser alterada através da mudança da área transversal pela qual o fluido percorre, pois pelo princípio da conservação de massa a quantidade de ar que entra em um tudo deve ser a mesma que sai deste tudo, supondo o fluido incompressível (SEDDON; NEWMAN, 2011, p. 23–25).

Baseado nos princípios expostos anteriormente, quando o fluxo de ar percorre a superfície da hélice, o fluido que percorre a superfície superior possui maior velocidade que o fluxo de ar que passa pela superfície inferior, pois a parte superior da pá tem uma área maior que a parte inferior. Deste modo, a pressão no lado inferior da hélice é maior que no lado superior.

Geralmente, as hélices de helicópteros são simétricas, isto é, as superfícies superior e in-

ferior possuem a mesma área. Para produzir um efeito de assimetria entre estas áreas, faz-se um movimento de inclinação da pá, formando um ângulo $\Theta(\Psi)$ com o plano de rotação da hélice chamado de ângulo de ataque, como mostrado na Figura 2. Efetuando-se o controle do ângulo de ataque, pode-se realizar os movimentos verticais do helicóptero (ANDERSON; EBERHARDT, 2010, p. 13–14).

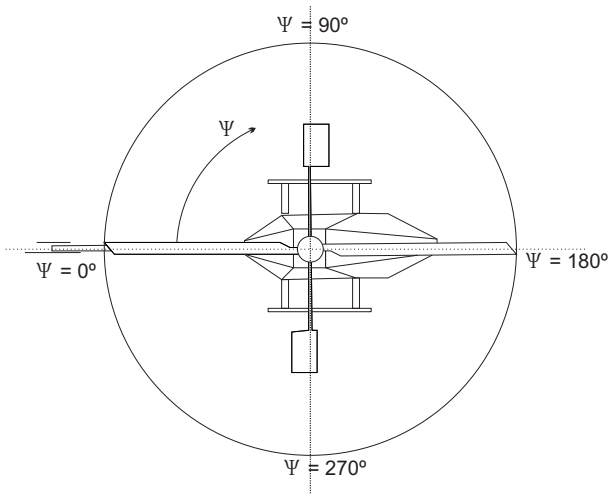
Figura 2: Ângulo de ataque da hélice do helicóptero



Fonte: PIZETTA (2013).

O movimentos laterais e longitudinais do helicóptero são produzidos através da distribuição assimétrica da força ao longo do movimento de rotação das pás. Isto é, gerando-se mais força em uma determinada posição de rotação da hélice do que em outra, o helicóptero tende a se inclinar e a força de sustentação que antes estava direcionada para cima também se inclina. Decompondo essa força, percebe-se a existência de duas componentes: uma vertical para cima e outra horizontal (lateral ou longitudinal), responsável pelo deslocamento do veículo (ANDERSON; EBERHARDT, 2010, p. 230–232). A Figura 3 mostra as posições angulares das hélices ao redor do rotor principal. Observa-se que o ângulo de ataque é função da posição da pá.

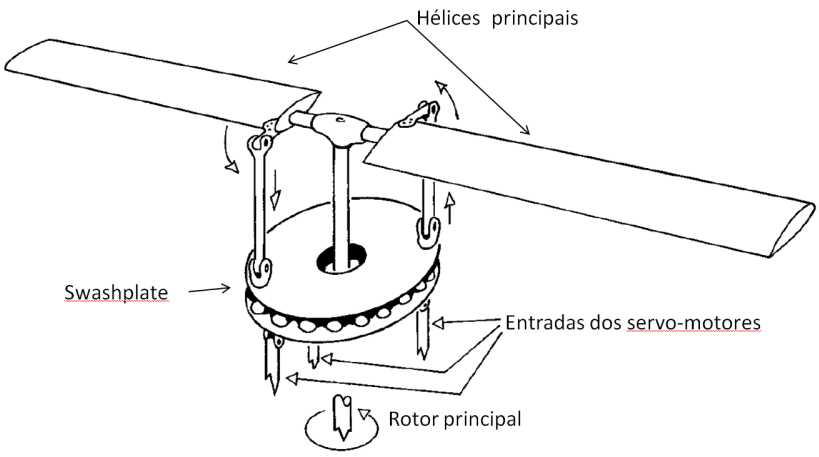
Figura 3: Posições angulares das hélices ao redor do rotor principal



Fonte: PIZZETA (2013).

A geração de tais deslocamentos é possível devido a um mecanismo chamado de *swash-plate* (do termo em português prato oscilante). Este dispositivo tem a finalidade de alterar a inclinação das pás (ângulo de ataque) de forma periódica, promovendo assim a distribuição assimétrica (para os deslocamentos horizontais) ou inclinando as pás de forma igualitária, alterando o módulo do torque produzido (deslocamento vertical) (ANDERSON; EBERHARDT, 2010, p. 226–227). Isto é, o prato oscilante muda a orientação da força de propulsão do helicóptero . A Figura 4 ilustra uma versão simplifica de um *swash-plate*.

Figura 4: *Swashplate* de um helicóptero



Fonte: PADFIELD (2007, modificado pelo autor).

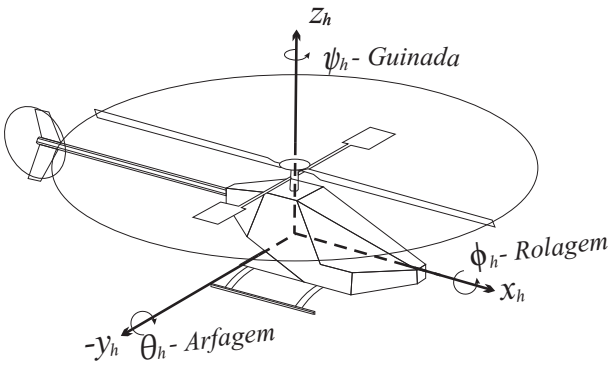
Em um helicóptero real, o piloto controla a posição do *swashplate* através de manetes com atuadores hidráulicos que movimentam o prato oscilante. Já em helimodelos, servo motores conectados a bielas são utilizados para o controle do *swashplate*.

Outro fenômeno associado à aerodinâmica do helicóptero é o efeito de anti-torque. Uma vez que o rotor principal produz torque para mover as hélices, por reação a fuselagem do veículo tende a rotacionar no sentido oposto. Para evitar este movimento indesejado da cabine, usa-se um rotor em uma estrutura de cauda que produz um torque contrário ao da fuselagem e, controlando o rotor de cauda, pode-se rotacionar o helicóptero em seu próprio eixo (PIZETTA, 2013, p. 37).

Por fim, observa-se que o helicóptero possui seis graus de liberdade, sendo três desloca-

mentos lineares e três ângulos de inclinação, apresentando uma manobrabilidade tridimensional, como mostrado na Figura 5.

Figura 5: Graus de liberdade de um helicóptero



Fonte: PIZETTA (2013).

2.2 Princípio de funcionamento de um helimodelo

Um helicóptero miniatura (também denominado helimodelo) se baseia nos mesmos princípios aerodinâmicos de um veículo real, porém existem particularidades principalmente no que se refere ao acionamento das pás e ao controle do voo. Tais aspectos são importantes no projeto do controlador e do sistema embarcado.

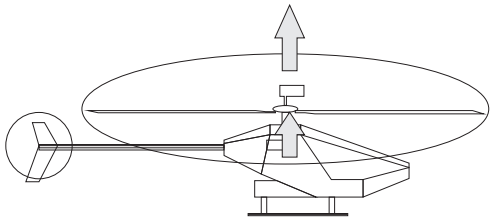
O helimodelo que se propõe utilizar neste projeto é o modelo T-REX 600 ESP da fabricante taiwanesa *ALIGN* mostrado na Figura 6.

Figura 6: Helimodelo T-REX 600 da ALIGN

O controle da posição, velocidade e orientação de um helimodelo no espaço tridimensional é realizado normalmente através de cinco comandos de entrada (SANTANA, 2011, p. 18):

Aileron: controla o passo cíclico lateral do rotor principal. Este comando permite o deslocamento lateral do veículo e o movimento de rolagem, como ilustrado na Figura 7.

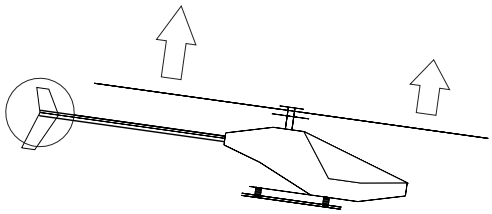
Figura 7: Passo cíclico lateral



Fonte: PIZETTA (2013).

Profundor: controla o passo cíclico longitudinal do rotor principal de forma a produzir o deslocamento na direção longitudinal (avanço ou retrocesso) do veículo e o movimento de arfagem. Está ilustrado na Figura 8.

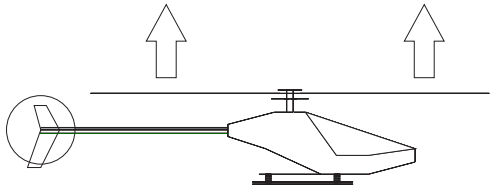
Figura 8: Passo cíclico longitudinal



Fonte: PIZETTA (2013).

Coletivo: controla o passo coletivo no rotor principal. Este comando resulta no deslocamento vertical do helicóptero, alterando a sua altitude. A Figura 9 mostra o efeito do comando.

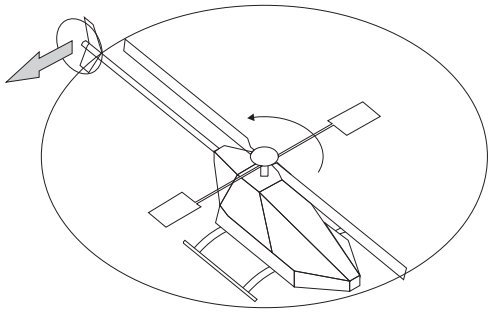
Figura 9: Passo coletivo do rotor principal



Fonte: PIZETTA (2013).

Leme: controla o passo coletivo no rotor de cauda, cuja função é compensar o feito de anti-torque produzido pelo rotor principal. Também é responsável pelo movimento de guinada, como apresentado na Figura 10.

Figura 10: Passo coletivo do rotor de cauda



Fonte: PIZETTA (2013).

Acelerador: controla a velocidade de rotação do rotor principal do helimodelo.

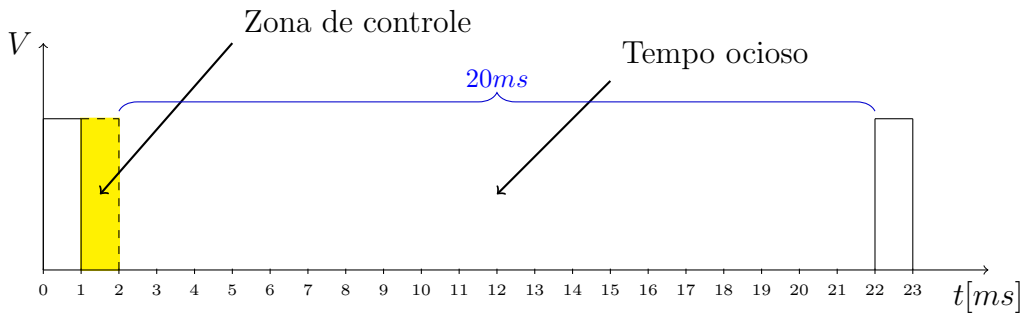
É importante destacar que no helimodelo o motor do rotor principal também é o responsável por produzir o movimento no rotor de cauda, através de um sistema de redução e um eixo mecânico de transmissão. Já em um veículo real, há um motor para o rotor de cauda. Portanto, o comando Acelerador controla não só a velocidade de rotação do rotor principal, mas também a do rotor de cauda (PIZETTA, 2013, p. 37).

O helicóptero elétrico deste trabalho possui quatro servo motores para controlar o deslocamento do veículo. Três dos tais servos motores são utilizados no controle do *swashplate*, e o último destes é usado no *swashplate* da cauda (somente é possível alterar o coletivo).

Os servo motores são acionados através de sinais PWM (do termo em inglês *Pulse Width Modulation*) normalmente gerados por um receptor de rádio frequência. O receptor, por sua vez, recebe um sinal com o método de modulação por posição de pulso (PPM, sigla do termo em inglês *Pulse Position Modulation*) enviado por um transmissor, como um rádio controle utilizado por um operador. Assim, o rádio controle envia um sinal PPM ao receptor, que o converte em sinais PWM para acionar os servomotores (PIZETTA, 2013, p. 39).

Um sinal típico de PWM é ilustrado na Figura 11. Observa-se que o intervalo útil de controle é entre 1ms e 2ms, com pulsos transmitidos a cada 20ms. Assim, a largura do pulso dentro deste intervalo determina o ângulo de rotação do servo motor.

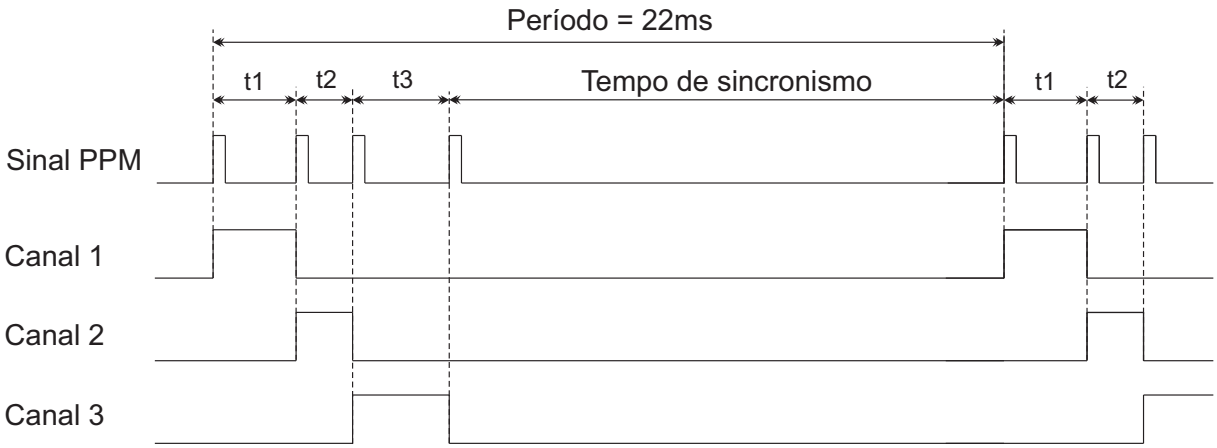
Figura 11: Sinal de PWM aplicado ao servo motor



Fonte: PIZETTA (2013).

A Figura 12 apresenta a conversão do sinal PPM em três sinais PWM. A largura do pulso de um sinal PWM é determinada pelo intervalo de tempo entre dois pulsos no sinal PPM. O sinal PPM é transformado em tantos sinais PWM quanto forem os canais do rádio controle. Cada canal representa uma informação transmitida. O rádio controle utilizado no projeto é o DX8 da fabricante *Spektrum*, possuindo oito canais ao todo (PIZETTA, 2013, p. 39–40).

Figura 12: Conversão de um sinal PPM em sinais PWM



Fonte: PIZETTA (2013).

Outro recurso presente no helimodelo é o conjunto de barras estabilizadoras (*flybar*) presentes no rotor principal. A finalidade de tais barras é facilitar a pilotagem, através do amortecimento dos efeitos de forças externas e suavização do movimento do próprio helicóptero. Elas respondem apenas aos comandos cíclicos e não ao coletivo (SANTANA,

2011, p. 20).

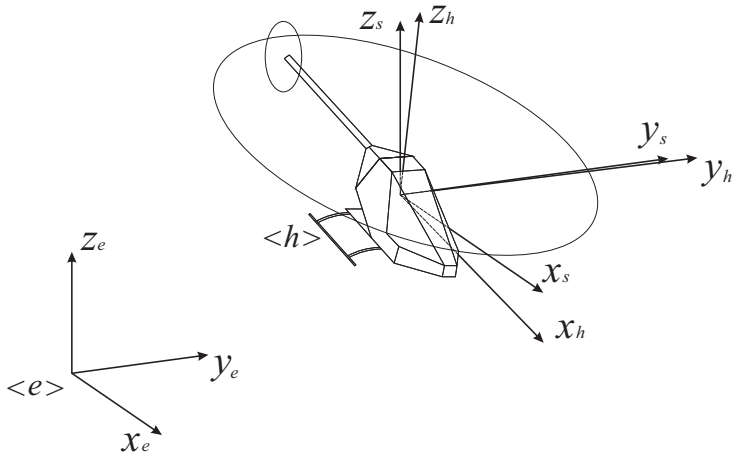
Uma particularidade de helicópteros de rádio controle é que o comando de aceleração é conjunto com o comando de coletivo. Assim, uma mesma manete realiza os dois comandos simultaneamente. O rádio controle permite a saturação da aceleração em um certo valor, mantendo a velocidade de rotação do rotor principal constante.

Por fim, o motor principal do helimodelo é do tipo de corrente contínua sem escovas (conhecido pelo termo em inglês *Brushless DC motor*) e tem a sua velocidade controlada por um módulo eletrônico chamado ESC (do termo em inglês *Electronic Speed Controller*) que também é acionado por um canal PWM.

2.2.1 Sistemas de referência de um helimodelo

Em navegação autônoma, busca-se controlar a posição e orientação de um helimodelo no espaço tridimensional tendo como referência um sistema inercial global, como o sistema terrestre por exemplo. A Figura 13 ilustra os sistemas de referência adotados.

Figura 13: Sistemas de referência adotados



Fonte: PIZETTA (2013).

Na Figura 13, o subscrito e representa o sistema inercial, o sistema do helicóptero (centrado em seu centro de massa é representado pelo subscrito h e o subscrito s representa o sistema espacial, que é o sistema e transladado para a posição do veículo ao longo do tempo.

Uma vez que os sensores a bordo do veículo realizam as medições em relação ao helicóptero, isto é, no sistema de referência do helimodelo, é importante determinar os sistemas inercial e do veículo para que as variáveis sejam devidamente controladas no sistema de interesse (inercial).

2.3 Dispositivos adicionais presentes

Um helimodelo comercial, como o utilizado, não possui qualquer instrumentação ou sistema eletrônico de controle, sendo a única forma de controle através de um rádio controle operado por uma pessoa. Portanto, se faz necessário desenvolver o sistema embarcado e equipar o veículo com os sensores desejados.

Ainda que o objetivo deste trabalho seja o projeto de um sistema de navegação autônoma, o uso de um rádio controle não será dispensado. Uma vez que a pilotagem de um helimodelo demanda experiência, a finalidade de tal dispositivo é prover maior segurança aos pesquisadores e ao equipamento, pois nenhum integrante do grupo de pesquisa possui habilidade para operar aeromodelos. Assim, usando um determinado canal do rádio controle, pode-se escolher entre operar o veículo manualmente quando necessário ou através do controlador nos experimentos.

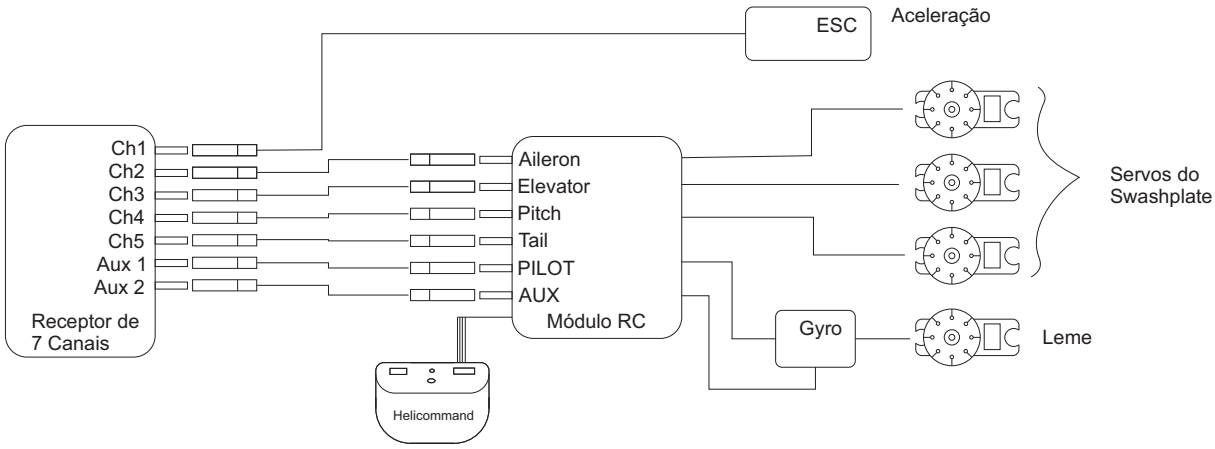
Para aprimorar ainda mais a segurança no uso do helimodelo, foi incluído um estabilizador de baixo nível no sistema embarcado, que é recomendado a pilotos iniciantes. O modelo utilizado é o Helicommand 3D da fabricante alemã CAPTRON. Este dispositivo não confere capacidade de navegação autônoma ao helicóptero, mas sim contribui na estabilização em voo pairado, tentando manter o veículo em uma determinada posição quando o rádio controle está com as manetes na posição neutra de cada uma. Assim, tal módulo atua tanto no caso de controle automático quanto de controle manual (PIZETTA, 2013, p. 56–58).

O estabilizador faz a leitura dos sinais PWM de saída do receptor do rádio através do seu módulo RC e, após processamento de tais sinais, envia sinais de saída para os servo motores. O Helicommand conta com um giroscópio (sensor que mede velocidades angulares nos eixos coordenados) e um sensor semicondutor (CCD, do termo em inglês *Charge-Coupled Device*) voltado para baixo, que realiza a captura de imagens. O fabricante não fornece

detalhes sobre o processamento interno do estabilizador (PIZETTA, 2013, p. 56–58).

A Figura 14 mostra o diagrama de conexão entre o receptor do rádio controle, o Helicommand e os servo motores. Deve-se mencionar que o helimodelo possui um giroscópio (bloco Gyro na figura) que tem a função de auxiliar na estabilização de possíveis oscilações da cauda, evitando o movimento indesejado de guinada (SANTANA, 2011, p. 19).

Figura 14: Esquemático das conexões entre o receptor do rádio e o Helicommand



Fonte: PIZETTA (2013).

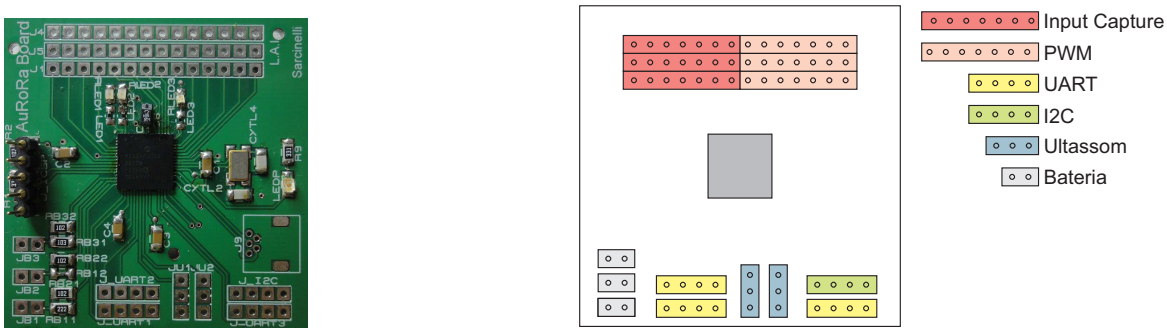
2.3.1 Módulo eletrônico *AuRoRa Board*

Com a finalidade de realizar o acionamento dos servo motores e a leitura dos canais do receptor do rádio, Pizzeta (2013) desenvolveu uma placa de circuito impresso chamada de *AuRoRa Board* contendo um microcontrolador para operar tais funções.

O microcontrolador utilizado é o modelo PIC24FJ256GB206, da fabricane Microchip. Tal dispositivo tem recursos como nove canais de geração de sinais de PWM, usados para acionar os servo motores, e nove canais de *input capture*, um periférico do microcontrolador que faz a leitura de pulsos digitais em sua entrada, usado para ler os canais do rádio controle (PIZETTA, 2013, p. 58–61). A Figura 15 apresenta uma imagem da placa e o seu diagrama esquemático.

A placa conta com diversas interfaces de comunicação, como três portas seriais assíncronas (periférico UART, do termo em inglês *Universal Asynchronous Receiver/Transmitter*)

Figura 15: Placa AuRora Board

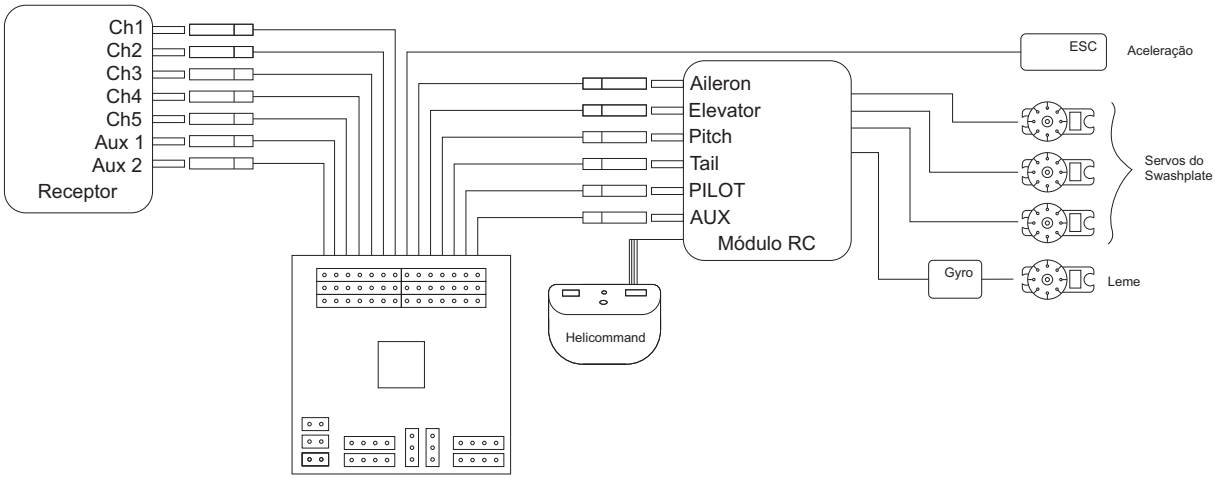


(a) Esquemático da placa (b) Imagem da AuRoRa Board

Fonte: PIZETTA (2013).

para se comunicar com o computador de bordo e sensores, além de uma interface de comunicação I2C (do termo em inglês *Inter-Integrated Circuit*) também para se comunicar com sensores. Canais de conversores analógico para digital (sigla ADC) são usados para ler dados de sensores ultrassônicos e o nível de tensão das baterias (PIZETTA, 2013, p. 58–61). A Figura 16 mostra as conexões entre os componentes apresentados com a inclusão da placa AuRoRa Board.

Figura 16: Esquemático das conexões entre a AuRoRa Board e o sistema anterior



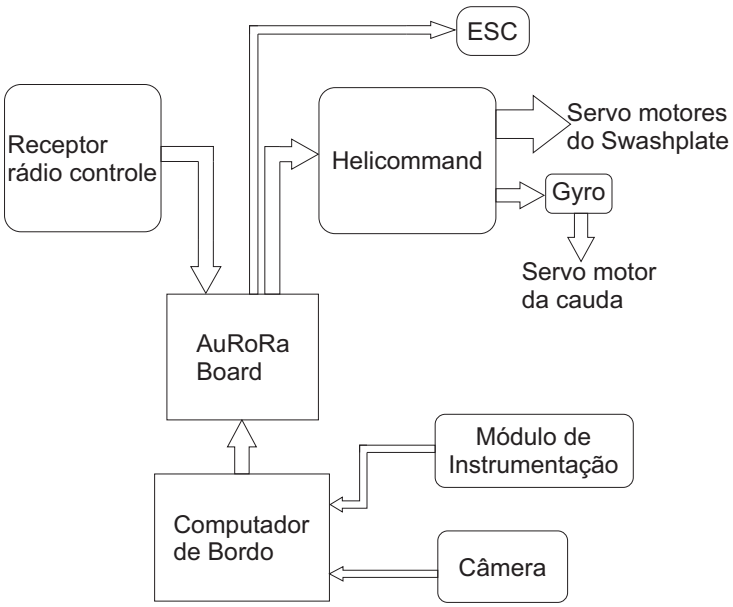
Fonte: PIZETTA (2013).

A partir da figura anterior, observa-se que a AuRoRa Board tem a capacidade de ler os canais do receptor de rádio e, no caso de controle manual, pode copiar os sinais PWM para serem enviados ao Helicommand. Já no caso do controle automático, apesar de ler os canais do receptor, estes são ignorados e as larguras dos pulsos dos sinais PWM transmitidos ao estabilizador são determinadas pelo controlador.

2.4 Sistema embarcado proposto

Do ponto de vista de sistema embarcado, este projeto tem como proposta incluir um computador de bordo e um módulo microcontrolado para a leitura e filtragem dos dados sensoriais. Além disso, propõe-se o uso de uma câmera para o processamento de imagens do solo, completando o sensoriamento do veículo. A Figura 17 apresenta o esquemático de conexões entre os componentes na nova arquitetura, incluindo os novos dispositivos.

Figura 17: Arquitetura proposta



Fonte: Próprio autor.

O computador de bordo conterá um sistema operacional baseado em Linux e executará o controlador. Além disso, receberá dados do módulo de instrumentação e realizará o processamento dos quadros de imagens transmitidos pela câmera.

A partir da visão geral do sistema de navegação mostrado nesta seção, os capítulos seguintes apresentarão em detalhes cada componente da arquitetura proposta.

3 INSTRUMENTAÇÃO DE VOO

Um componente importante de um sistema de navegação autônoma é a instrumentação de voo. Através de sensores, podem-se obter informações à respeito da orientação (chamado também de atitude) e posição do veículo no espaço tridimensional, o que permite controlar devidamente o helimodelo.

Neste capítulo são apresentados os sensores inerciais, sendo estes o conjunto básico de sensoriamento de qualquer sistema de navegação, incluindo aviões, foguetes e satélites. Será descrito um método de realizar a fusão e processamento dos dados sensoriais de forma a obter as informações desejadas. O método de filtragem empregado é o Filtro de Kalman.

3.1 Unidade de medição inercial

A IMU (sigla do termo em inglês *Inertial Measurement Unit*) é um módulo eletrônico composto por um conjunto de sensores cujos dados fornecidos permitem estimar a orientação de um objeto no espaço tridimensional (SANTANA, 2011, p. 30–31). Tais sensores são acelerômetros, giroscópios, podendo haver também magnetômetros. A seguir é apresentada a descrição de cada sensor.

Acelerômetros: são sensores que possuem a capacidade de medir a aceleração linear nos eixos coordenados, isto é, em seu sistema de referência, além de incluir a aceleração da gravidade. Podem ser usados para medir a inclinação de um objeto em relação ao vetor aceleração da gravidade, indicando os ângulos de rolagem e arfagem. Entretanto, o acelerômetro sofre interferências devido a vibrações mecânicas, o que reduz a confiabilidade de suas medições. Como o helimodelo sofre oscilações por causa do movimento do rotor principal, o uso apenas do acelerômetro não é suficiente para

estimar a orientação do veículo (SANTANA, 2011, p. 30).

Giroscópios: são dispositivos que medem as velocidades angulares em torno do seu sistema de referência. As medições dos giroscópios são menos influenciadas por ruídos de alta frequência do que a dos acelerômetros. Porém, estes dispositivos são sujeitos a um fenômeno chamado de *drift*, cuja característica é o deslocamento do referencial zero com o passar do tempo. Isto é, iniciando com o sensor inicialmente estático (medição com valor nulo), após movê-lo e regressar ao estado inicial o valor em sua saída não será zero (SANTANA, 2011, p. 31).

Magnetômetros: são sensores capazes de medir o campo magnético terrestre, indicando a sua orientação. Têm a finalidade de fornecer um segundo vetor de referência para a estimativa do ângulo de guinada, uma vez que os acelerômetros não são capazes de medi-lo. Entretanto, o magnetômetro sofre interferências de materiais metálicos e outras fontes de campo magnético, como o motor do helimodelo, que produzem distorções nas medições do campo. As distorções provocadas por objetos que produzem campo magnético são chamadas de *hard iron*. Já as distorções provocadas pela presença de materiais ferromagnéticos, que apenas distorcem ou defletem o campo existente são chamadas de *soft iron* (SANTANA, 2011, p. 31).

Devido às capacidades e problemas de cada tipo de sensor, apenas a medição de um destes não é suficiente para uma estimativa confiável da orientação de um objeto no espaço. Logo, é necessário utilizar um conjunto com os sensores apresentados anteriormente. Além disso, existem técnicas de compensação de erros e calibração dos sensores para tornar as medições com menos ruídos e valores de *offset*, deixando-as mais confiáveis. Ademais, deve-se usar algum método matemático de filtragem dos ruídos e de fusão de dados, na qual operação com dados de diferentes tipos de sensores são usadas para estimar uma outra variável, como no caso os ângulos em relação a um referencial desejado (SANTANA, 2011, p. 41).

Na literatura pode-se encontrar diversos trabalhos a respeito de filtragem e fusão sensorial, como Madgwick, Vaidyanathan e Harrison (2010) e Li e Mourikis (2013), destacando-se o uso do filtro de Kalman e o filtro de Kalman estendido, que são utilizados neste trabalho para estimar as velocidades lineares e a orientação espacial respectivamente. Os sistemas que possuem a função de obter a orientação no espaço são denominados de AHRS (sigla do termo em inglês *Attitude and Heading Reference Systems*).

3.2 Sensor de ultrassom

De uso comum na robótica, o sensor de ultrassom é utilizado neste trabalho para a medição da altitude do veículo com relação ao solo. O princípio de funcionamento deste dispositivo se baseia na emissão de ondas ultrassônicas e na detecção das ondas refletidas por um ou mais objetos. O tempo de trânsito da onda é calculado e, conhecendo-se o valor da velocidade do som no meio, pode-se determinar a distância.

Porém, para a medição de distância em relação ao solo, a presença de objetos e irregularidades no terreno podem provocar erros nas medições da altitude, uma vez que estes podem gerar reflexões da onda sonora. Por isso, o sensor de ultrassom tem seu uso recomendado apenas em ambientes estruturados, como aqueles com superfícies planas, que é o caso deste projeto. Em ambientes externos, para determinar a altitude, usam-se sensores barométricos, que permitem determinar a altitude em função da pressão atmosférica.

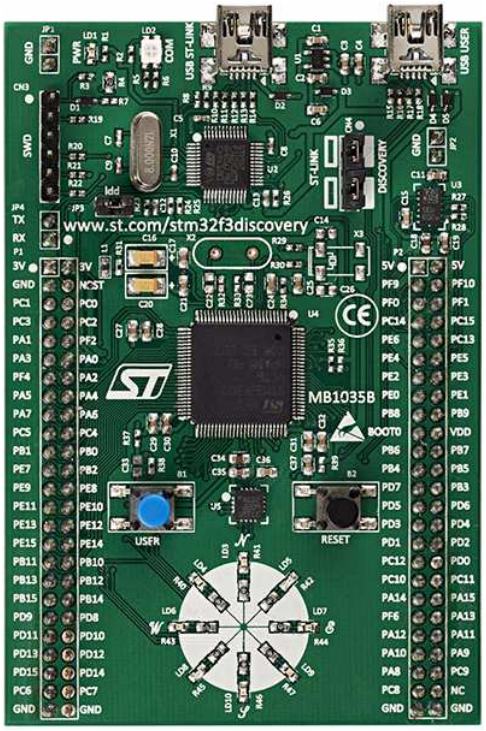
3.3 Módulo embarcado de sensoriamento

Neste projeto, um sistema eletrônico microcontrolado será utilizado para efetuar a leitura dos sensores da IMU e de ultrassom, além de executar os algoritmos de filtragem e fusão sensorial escolhidos. Os dados então são enviados ao computador de bordo através de uma porta de comunicação serial assíncrona. A Figura 18 apresenta o sistema embarcado usado para a instrumentação.

O módulo embarcado é uma placa de desenvolvimento modelo STM32F3DISCOVERY da fabricante STMicroelectronics. Tal placa contém um microcontrolador STM32F303VCT6 da série STM32 F3, cujo núcleo é o ARM Cortex-M4 de 32 *bits*, com $256KB$ de memória *flash*, $48KB$ de RAM, frequência de operação de $72MHz$, e unidade de ponto flutuante (sigla em inglês FPU). O microcontrolador possui ainda diversos periféricos como interfaces de comunicação serial (protocolos SPI, I2C, CAN e USB são exemplos), além de USART (sigla do termo me inglês Universal Synchronous/Asynchronous Receiver/Transmitter), conversores analógico-digital e digital-analógico, temporizadores de 16 e 32 *bits*, canais de PWM, portas de entrada e saída digitais, entre outros.

A placa possui um giroscópio modelo L3GD20 e um acelerômetro integrado com um mag-

Figura 18: Placa de desenvolvimento STM32F3 Discovery



Fonte: STMICROELECTRONICS.

netômetro, modelo LSM303DLHC, todos estes dispositivos da STMicroelectronics, e que podem medir nos três eixos ortogonais. O giroscópio se comunica através de uma interface SPI e a bússola (acelerômetro e magnetômetro) através de uma interface I2C. Assim, os sensores que compõem a IMU já estão contidos no próprio módulo, não necessitando de adquirir uma unidade inercial à parte. Além de tais sensores, estão presentes um programador (e depurador de código) ST-LINK/V2, LEDs, botões e conectores.

A fabricante STMicroelectronics fornece uma interface de programação do núcleo e dos periféricos, encapsulando o *hardware*, tanto para o microcontrolador quanto uma interface própria para a placa de desenvolvimento, envolvendo os LEDs, botões e os sensores. Desta forma, a configuração se torna mais rápida, uma vez que se trabalha com código em um nível mais alto.

Outro recurso importante no que se refere ao *software* é a biblioteca de funções matemáticas disponibilizada pela ARM, a desenvolvedora da arquitetura do núcleo do microcontrolador. Esta biblioteca possui um código que é otimizado para tal arquitetura, o que torna a execução das operações mais rápida. As funções matemáticas relacionadas a álgebra linear presentes na biblioteca, principalmente cálculo matricial, são utilizadas nas

técnicas de filtragem apresentadas nas seções seguintes.

O sensor de ultrassom utilizado é o HC-SR04, que pode medir distâncias entre $2cm$ e $4m$ com resolução de $0,3cm$. A Figura 19 ilustra o modelo utilizado.

Figura 19: Sensor de ultrassom HC-SR04

Tal dispositivo se comunica através da transmissão de um pulso de pelo menos $10us$ enviado para o sensor e este retorna um outro pulso cuja largura é proporcional à distância medida. A equação

$$\text{distância} = \Delta t_{pulso} \times 340/2, \tag{3.1}$$

mostra como é realizado o cálculo da distância, onde Δt_{pulso} é a largura do pulso de resposta enviado pelo sensor e 340m/s é a velocidade aproximada do som no ar. A leitura deste pulso pode ser feita através do recurso *input capture* do microcontrolador: detectam-se as bordas de subida e descida do pulso e armazenam-se os valores do contador de um temporizador entre estes dois instantes, obtendo assim o tempo de duração com alguns cálculos.

É importante destacar o motivo de se ter módulos separados para acionamento dos servos e para a leitura de dados sensoriais. A placa STM32F3Discovery não possui a quantidade de canais de PWM e *input capture* necessários para o acionamento dos servo motores e o PIC da AuRoRa Board não tem a capacidade de processamento que permita realizar as tarefas de instrumentação e acionamento em tempo hábil. Portanto, decidiu-se usar módulos separados para cada função.

3.4 Filtragem e fusão sensorial

Uma vez definidos os sensores da IMU, a partir da obtenção de dados de acelerações lineares, velocidades angulares e orientação do campo magnético terrestre, deve-se então produzir informações úteis à navegação do veículo no espaço tridimensional. A locomoção do robô se baseia no conhecimento de variáveis como acelerações, velocidades, posição e orientação ao longo do tempo e em relação a um referencial predefinido (SANTANA, 2011, p. 41).

Desta forma, para obter as informações desejadas, é necessário utilizar algum método matemático que realize a integração dos dados sensoriais e que resulte nas variáveis de interesse. Além disso, tal algoritmo deve realizar a filtragem dos dados sensoriais, pois estes contêm erros e ruídos que tornam as medidas pouco confiáveis.

Na literatura há diversos algoritmos usados para tais fins e, no geral, envolvem conhecimentos nas áreas de probabilidade e estatística, álgebra linear e otimização, entre outras. Uma abordagem bastante comum é o Filtro de Kalman, como em Watson (2013), e que está descrito logo a seguir.

3.4.1 Filtro de Kalman

O filtro de Kalman é um algoritmo que atua mais como um estimador do que um filtro propriamente (GROVES, 2008, p. 55). Foi desenvolvido por Rudolph E. Kalman em 1960, e tem sido utilizado vastamente em diversas áreas, como exemplo a robótica.

É um procedimento que combina dados sensoriais contendo ruído, como os fornecidos pelos sensores da IMU, para estimar o estado de um sistema linear com incertezas na sua dinâmica, como a presença de distúrbios. As equações do filtro resultam em um estimador tipo preditor/corretor que é ótimo no sentido de que minimiza a covariância estimada do erro (GREWAL; WEILL; ANDREWS, 2007, p. 255-256). O conjunto de equações

$$\hat{\mathbf{x}}_k^- = \mathbf{F}\hat{\mathbf{x}}_{k-1} + \mathbf{B}\mathbf{u}_k + \mathbf{w}_{k-1} \tag{3.2}$$

$$\mathbf{P}_k^- = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{Q} \tag{3.3}$$

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}^T (\mathbf{H}\mathbf{P}_k^- \mathbf{H}^T + \mathbf{R})^{-1} \tag{3.4}$$

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}\hat{\mathbf{x}}_k^-) \tag{3.5}$$

$$\mathbf{P}_k = \mathbf{P}_k^- - \mathbf{K}_k \mathbf{H} \mathbf{P}_k^- \tag{3.6}$$

compõe o algoritmo do filtro de Kalman para o caso discreto, onde:

$\hat{\mathbf{x}}_k$ é o vetor de estados contendo as variáveis que se deseja estimar(por exemplo, aceleração, velocidade e posição) no instante k.

$\mathbf{F}$ é a matriz de transição de estado que aplica os efeitos dos parâmetros do sistema no instante $k-1$ para o instante k.

$\mathbf{u}_k$ é um vetor contendo as entradas de controle.

$\mathbf{B}$ é uma matriz que mapeia os efeitos de cada entrada nos estados do sistema.

$\mathbf{w}_{k-1}$ é o vetor que representa os ruídos de processo. Este vetor é considerado um conjunto de variáveis aleatórias, todas de distribuição normal, com valor médio nulo e covariância dada pela matriz $\mathbf{Q}$.

$\mathbf{P}_k$ é a matriz de covariância associada ao vetor de estados.

$\mathbf{Q}$ é a matriz de covariância dos ruídos do processo.

$\mathbf{K}_k$ é a matriz chamada de ganho de Kalman. Esta matriz promove a minimização dos erros de covariância.

$\mathbf{z}_k$ é o vetor de medições, isto é, dos dados obtidos pelos sensores.

$\mathbf{H}$ é a matriz que mapeia o vetor de estados no espaço do vetor de medições de forma que $\mathbf{z}_k = \mathbf{H}\hat{\mathbf{x}}_k + \mathbf{v}_k$, onde $\mathbf{v}_k$ é o vetor de ruídos de medição, constituído por variáveis aleatórias de distribuição normal com valor médio nulo e covariância dada pela matriz $\mathbf{R}$.

$\mathbf{R}$ é a matriz de covariância associada ao vetor de medições.

O algoritmo do filtro de Kalman é dividido em dois estágios: atualização no tempo e atualização da medição (WELCH; BISHOP, 2001, p. 19–24).

O estágio de atualização no tempo ou também chamado de processo de predição envolve as Equações 3.2 e 3.3. A partir de um valor inicial $\hat{\mathbf{x}}_0$ e $\mathbf{P}_0$ ou de valores no instante anterior $k-1$, faz-se a predição dos valores do instante atual k. As variáveis $\hat{\mathbf{x}}_k^-$ e $\mathbf{P}_k^-$ são ditas *a priori*, pois são determinadas a partir da modelagem do sistema (WELCH; BISHOP, 2001, p. 19–24).

As Equações 3.4, 3.5 e 3.6 compõem o estágio de atualização de medição ou processo de estimação. Nesta etapa busca-se estimar o vetor de estados *a posteriori* $\hat{\mathbf{x}}_k$ a partir de uma combinação linear entre o vetor estimado *a priori* $\hat{\mathbf{x}}_k^-$ e uma subtração ponderada entre o vetor de medições atual e o vetor de estados *a priori*, efetuando uma correção no valor predito. Também ocorre a atualização da matriz de covariância $\mathbf{P}_k$, cuja importância está no fato de que reflete a variância da distribuição do estado (WELCH; BISHOP, 2001, p. 19–24).

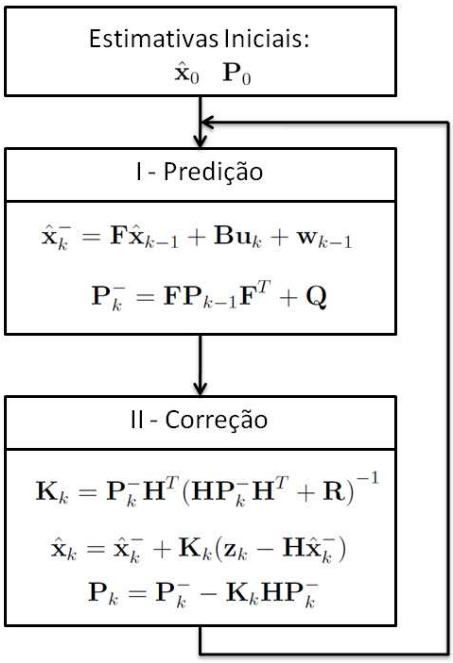
O algoritmo é recursivo. Assim, na próxima iteração, usam-se os valores da iteração anterior para determinar os atuais. Nota-se que o filtro de Kalman exige apenas os valores no instante $k-1$, não sendo necessário um maior histórico de amostras (WELCH; BISHOP, 2001, p. 19–24). A Figura 20 apresenta um esquemático que resume a execução do algoritmo.

3.4.2 Filtro de Kalman estendido

Como mencionado anteriormente, o filtro de Kalman é utilizado em sistemas lineares, isto é, no caso discreto, o processo deve ser modelado por uma equação a diferenças. Entretanto, é possível estender o uso de tal filtro para sistemas não lineares ou que contenha medições não lineares, como no caso da navegação de robôs aéreos. Este filtro é denominado de Filtro de Kalman Estendido (EKF - Extended Kalman Filter, em inglês) (WELCH; BISHOP, 2001, p. 24–31).

No filtro estendido, a matriz de transição de estados $\mathbf{F}$ e a matriz de medição $\mathbf{H}$ são subs-

Figura 20: Fluxograma do filtro de Kalman



Fonte: Próprio autor.

tituídas pelas funções não lineares $f(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{w}_k)$ e $h(\hat{\mathbf{x}}_k, \mathbf{v}_k)$, respectivamente. Ademais, a estrutura básica do filtro permanece a mesma que a do filtro linear (WELCH; BISHOP, 2001, p. 24–31).

Entretanto, no cálculo das matrizes de covariância e ganho de Kalman são usados os modelos linearizados. Da mesma forma que a linearização pela Série de Taylor, a qual diz que pode-se aproximar uma função em torno de um ponto de interesse através de uma série formada pelas derivadas dessa função no ponto, pode-se linearizar os modelos do sistema e de medição em torno da estimativa atual usando derivadas parciais da modelagem.

O conjunto de equações

$$\hat{\mathbf{x}}_k^- = f(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{w}_k) \tag{3.7}$$

$$\mathbf{P}_k^- = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{Q} \tag{3.8}$$

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}^T (\mathbf{H}\mathbf{P}_k^- \mathbf{H}^T + \mathbf{R})^{-1} \tag{3.9}$$

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{z}_k - h(\hat{\mathbf{x}}_k, \mathbf{v}_k)) \tag{3.10}$$

$$\mathbf{P}_k = \mathbf{P}_k^- - \mathbf{K}_k \mathbf{H} \mathbf{P}_k^- \tag{3.11}$$

compõe o algoritmo do filtro de Kalman estendido, onde:

$\boldsymbol{f(\hat{x}_{k-1}, u_k, w_k)}$ é a função não linear que modela o sistema.

$\boldsymbol{h(\hat{x}_k, v_k)}$ é a função não linear que modela o processo de medição.

$\mathbf{F}$ é a matriz jacobiana de $f(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{w}_k)$, isto é,

$$F_{(i,j)} = \left.\frac{\partial f}{\partial \hat{\mathbf{x}}}\right|_{(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k)} \tag{3.12}$$

$\mathbf{H}$ é a matriz jacobiana de $h(\hat{\mathbf{x}}_k, \mathbf{v}_k)$, ou seja,

$$H_{(i,j)} = \left.\frac{\partial h}{\partial \hat{\mathbf{x}}}\right|_{\hat{\mathbf{x}}_{k-1}^-} \tag{3.13}$$

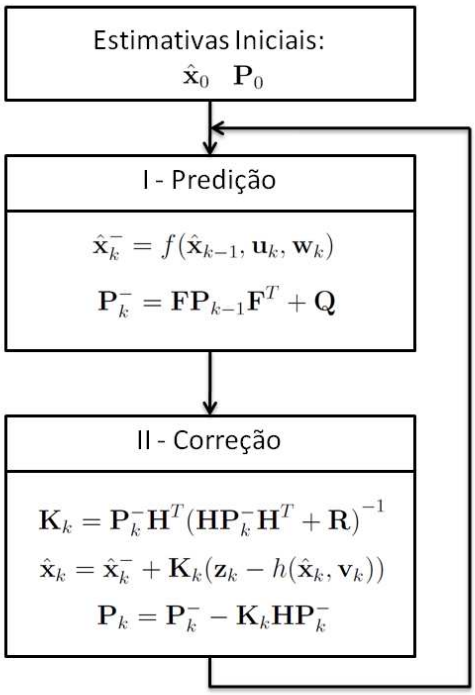
O EKF é executado da mesma forma que o filtro original. Existem os dois estágios, o de predição e o de medição, e os valores *a priori* e *a posteriori*. Segue-se a mesma ordem de computação nas equações, como mostrado na Figura 21.

3.5 Obtenção da orientação

Em sistemas de navegação, uma informação fundamental é a orientação do robô no espaço. Neste contexto, uma estimativa confiável da atitude é necessária, e pode ser obtida através do uso de um método de filtragem, como o filtro de Kalman descrito anteriormente.

Uma vez que o modelo relacionado à obtenção da orientação é não linear, o algoritmo a ser utilizado é o filtro de Kalman estendido. Na literatura, uma abordagem amplamente difundida de representação da atitude de um objeto é através de quatérnios, que então

Figura 21: Fluxograma do filtro de Kalman estendido



Fonte: Próprio autor.

é usada em conjunto com o filtro de Kalman. A representação por meio de quatérnios é descrita a seguir (SANTANA, 2011, p. 42).

3.5.1 Quatérnios

A forma mais comum de representar rotações no espaço é através da utilização de matrizes de rotação e os ângulos de Euler, que são ângulos que um vetor forma em relação aos três eixos coordenados de um referencial. Entretanto, uma alternativa à esta representação clássica é por meio de quatérnios (SANTANA, 2011, p. 42).

A álgebra de quatérnios foi desenvolvida pelo matemático Sir William R. Hamilton em 1843. Um quatérnio é um número com quatro dimensões, sendo uma componente real e as outras três componentes imaginárias (i, j, k) como em (KUIPERS, 1998, p. 11–12).

$$\mathbf{q} = [q_0 \quad q_1 \quad q_2 \quad q_3] = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3, \tag{3.14}$$

onde q_0, q_1, q_2 e q_3 são números reais e $\mathbf{i} = (1, 0, 0)$, $\mathbf{j} = (0, 1, 0)$ e $\mathbf{k} = (0, 0, 1)$, for-

mam a base ortonormal padrão em $\mathbb{R}^3$. Ademais, os quatérnios são chamados de números hipercomplexos, uma generalização do sistema de números complexos, uma vez que $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$ (KUIPERS, 1998, p. 11–12).

A álgebra de quatérnios envolve propriedades e operações como adição, subtração, produto vetorial e escalar, entre outras. Porém, o detalhamento sobre esta matemática está fora do escopo deste trabalho. Para mais informações consultar KUIPERS (1998).

3.5.2 Estrutura do algoritmo

O filtro de Kalman estendido usa a representação por meio de quatérnios (WATSON, 2013, p. 28), sendo que o vetor de estados é o quatérnio que representa a orientação do helimodelo em seu referencial e três valores de *offset* do giroscópio, da seguinte forma

$$\hat{\mathbf{x}} = [q_0 \quad q_1 \quad q_2 \quad q_4 \quad \omega_{xb} \quad \omega_{yb} \quad \omega_{zb}]^T \tag{3.15}$$

O estado inicial é definido com $\mathbf{q} = [1 \quad 0 \quad 0 \quad 0]$ e considerando os valores de *offset* dos giroscópio desconhecidos, de tal forma que $\hat{\mathbf{x}}_0 = [1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]^T$ (WATSON, 2013, p. 28).

A matriz de covariância estimada $\mathbf{P}$ é inicializada com valores iniciais grandes de variância, uma forma de representar a falta de conhecimento sobre o estado estimado atual (WATSON, 2013, p. 28). Assim,

$$\mathbf{P}_0 = \mathbf{I}_{7\times7} \times 10^8, \tag{3.16}$$

onde $\mathbf{I}_{7\times7}$ é uma matriz identidade 7x7.

O giroscópio fornece as velocidades angulares nos eixos x, y e z no sistema de referência do sensor. Assim, a derivada do quatérnio que descreve a taxa de variação da orientação do sistema inercial terrestre em relação ao referencial do sensor pode ser calculada pela Equação 3.18 (WATSON, 2013, p. 28–29). O modelo do estágio de predição que descreve a estimativa atual da orientação a partir da estimativa anterior, do vetor contendo as medições do giroscópio e da integração numérica da derivada do quatérnio é apresentado

na equação

$$\mathbf{q}_k = \mathbf{q}_{k-1} + \dot{\mathbf{q}}_k \times \Delta t, \tag{3.17}$$

onde

$$\dot{\mathbf{q}}(\mathbf{q}, \omega) = \frac{1}{2} \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & q_3 & -q_2 \\ -q_3 & q_0 & q_1 \\ q_2 & -q_1 & q_0 \end{bmatrix} \begin{bmatrix} \omega_x - \omega_{xb} \\ \omega_y - \omega_{yb} \\ \omega_z - \omega_{zb} \end{bmatrix} \tag{3.18}$$

Nesta etapa, os valores de *offset* do giroscópio não mudam. Assim, a função $f(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{w}_k)$ é dada por

$$f(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k, \mathbf{w}_k) = \begin{bmatrix} q_0 + \frac{\Delta t}{2} \times (-q_1(\omega_x - \omega_{xb}) - q_2(\omega_y - \omega_{yb}) - q_3(\omega_z - \omega_{zb})) \\ q_1 + \frac{\Delta t}{2} \times (q_0(\omega_x - \omega_{xb}) + q_3(\omega_y - \omega_{yb}) - q_2(\omega_z - \omega_{zb})) \\ q_2 + \frac{\Delta t}{2} \times (-q_3(\omega_x - \omega_{xb}) + q_0(\omega_y - \omega_{yb}) + q_1(\omega_z - \omega_{zb})) \\ q_3 + \frac{\Delta t}{2} \times (q_2(\omega_x - \omega_{xb}) - q_1(\omega_y - \omega_{yb}) + q_0(\omega_z - \omega_{zb})) \\ \omega_{xb} \\ \omega_{yb} \\ \omega_{zb} \end{bmatrix} \tag{3.19}$$

O Jacobiano $\mathbf{F}$ do modelo pode ser obtido através de sua derivada, resultando em

$$\mathbf{F} = \frac{\partial f}{\partial \hat{\mathbf{x}}} = \begin{bmatrix} 1 & -\frac{\Delta t}{2}\Delta\omega_x & -\frac{\Delta t}{2}\Delta\omega_y & -\frac{\Delta t}{2}\Delta\omega_z & \frac{\Delta t}{2}q_1 & \frac{\Delta t}{2}q_2 & \frac{\Delta t}{2}q_3 \\ \frac{\Delta t}{2}\Delta\omega_x & 1 & -\frac{\Delta t}{2}\Delta\omega_z & \frac{\Delta t}{2}\Delta\omega_y & -\frac{\Delta t}{2}q_0 & -\frac{\Delta t}{2}q_3 & \frac{\Delta t}{2}q_2 \\ \frac{\Delta t}{2}\Delta\omega_y & \frac{\Delta t}{2}\Delta\omega_z & 1 & -\frac{\Delta t}{2}\Delta\omega_x & \frac{\Delta t}{2}q_3 & -\frac{\Delta t}{2}q_0 & -\frac{\Delta t}{2}q_1 \\ \frac{\Delta t}{2}\Delta\omega_z & -\frac{\Delta t}{2}\Delta\omega_y & \frac{\Delta t}{2}\Delta\omega_x & 1 & -\frac{\Delta t}{2}q_2 & \frac{\Delta t}{2}q_1 & -\frac{\Delta t}{2}q_0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.20}$$

onde $\Delta\omega_x = (\omega_x - \omega_{xb})$, $\Delta\omega_y = (\omega_y - \omega_{yb})$ e $\Delta\omega_z = (\omega_z - \omega_{zb})$.

O vetor de medições é composto pelos dados sensoriais do acelerômetro e do magnetômetro como mostrado em 3.21, sendo que tais dados são normalizados em separado, de acordo com a sua natureza, uma vez que a normalização diminui os efeitos de distorções e erros de calibração. O acelerômetro mede a magnitude e direção do campo gravitacional no referencial do sensor, em conjunto com as acelerações lineares devido ao movimento do dispositivo. De modo similar, o magnetômetro fornece a magnitude e direção do campo

magnético da Terra no referencial do sensor, em conjunto com fluxos magnéticos locais e distorções.

$$\mathbf{z} = [acc_x \quad acc_y \quad acc_z \quad mag_x \quad mag_y \quad mag_z]^T \tag{3.21}$$

Porém, inicialmente, será assumido que o acelerômetro mede apenas a aceleração do campo gravitacional e o magnetômetro mede apenas o campo magnético terrestre. Se a direção destes vetores é conhecida, a medição dos campos (gravitacional e magnético) no referencial do sensor irá permitir determinar a orientação do sistema de referência do sensor em relação ao terrestre (MADGWICK; VAIDYANATHAN; HARRISON, 2010, p. 6).

O referencial fixo de gravidade é rotacionado para o referencial do sensor, representado pelo quatérnio unitário $\mathbf{q}$, e mapeado para o vetor de acelerações. Isto é, o campo gravitacional de referência $\vec{\mathbf{g}}$ é alinhado com a direção do campo medido no sistema de referência do sensor através da rotação como apresentado na equação (MADGWICK; VAIDYANATHAN; HARRISON, 2010, p. 7–8).

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = R(\mathbf{q}) \times \vec{\mathbf{g}} = \begin{bmatrix} -2(q_1q_3 - q_0q_2) \\ -2(q_2q_3 + q_0q_1) \\ -q_0^2 + q_1^2 + q_2^2 - q_3^2 \end{bmatrix}, \tag{3.22}$$

onde R($\mathbf{q}$) é a matriz de rotação em notação de quatérnios, e $\vec{\mathbf{g}} = [0 \quad 0 \quad -1]$ é o vetor de aceleração gravitacional de referência, normalizado no eixo z e com sentido para baixo (WATSON, 2013, p. 29–30).

O tratamento dos dados do magnetômetro é feito de forma análoga. O vetor de magnitude do campo magnético terrestre de referência, representado por $\vec{\mathbf{b}} = [b_x \quad b_y \quad b_z]$, pode ser rotacionado no referencial do sensor conforme a equação (WATSON, 2013, p. 29–30)

$$\begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} = R(\mathbf{q}) \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} b_x(q_0^2 + q_1^2 - q_2^2 - q_3^2) + 2b_y(q_1q_2 + q_0q_3) + 2b_z(q_1q_3 - q_0q_2) \\ 2b_x(q_1q_2 - q_0q_3) + b_y(q_0^2 - q_1^2 + q_2^2 - q_3^2) + 2b_z(q_2q_3 + q_0q_1) \\ 2b_x(q_1q_3 + q_0q_2) + 2b_y(q_2q_3 - q_0q_1) + b_z(q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{bmatrix}, \tag{3.23}$$

O campo magnético da Terra não pode ser referenciado facilmente, uma vez que depende da localização no globo terrestre. Assim, usa-se a técnica proposta por Madgwick,

Vaidyanathan e Harrison (2010), na qual o campo magnético da Terra pode ser considerado como um vetor com duas dimensões, existindo-se uma declinação vertical. Assim, assume-se uma componente no eixo x e outra no eixo z.

Entretanto, o magnetômetro sofre fortemente influências de distorções *hard iron* e *soft iron*, o que pode resultar em medições erradas de orientação. A compensação de distorções magnéticas é realizada de tal forma que o campo magnético terrestre medido pelo sensor, representado pelas componentes $(m_x^{'}, m_y^{'}, m_z^{'})$, pode ser calculado já no referencial inercial através dos dados do magnetômetro rotacionados pela estimativa da orientação (quatérnio). Considera-se ainda que a direção do campo magnético de referência é a mesma inclinação que a do campo medido (MADGWICK; VAIDYANATHAN; HARRISON, 2010, p. 11–12).

A equação

$$\begin{bmatrix} m_x^{'} \\ m_y^{'} \\ m_z^{'} \end{bmatrix} = R(\mathbf{q}) \times \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} = \begin{bmatrix} m_x(q_0^2 + q_1^2 - q_2^2 - q_3^2) + 2m_y(q_1q_2 + q_0q_3) + 2m_z(q_1q_3 - q_0q_2) \\ 2m_x(q_1q_2 - q_0q_3) + b_y(q_0^2 - q_1^2 + q_2^2 - q_3^2) + 2m_z(q_2q_3 + q_0q_1) \\ 2m_x(q_1q_3 + q_0q_2) + 2m_y(q_2q_3 - q_0q_1) + m_z(q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{bmatrix} \tag{3.24}$$

contém o cálculo da rotação do campo medido pelo magnetômetro, partindo do referencial do sensor para o referencial inercial. O campo magnético de referência é calculado de forma a ter a mesma inclinação do campo medido e pode ser obtido através da equação(MADGWICK; VAIDYANATHAN; HARRISON, 2010, p. 11–12).

$$\begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} \sqrt{m_x^{'2} + m_y^{'2}} \\ 0 \\ m_z^{'} \end{bmatrix}. \tag{3.25}$$

O novos valores do campo magnético de referência são substituídos no mapeamento original, como mostrado em

$$\begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} = R(\mathbf{q}) \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} b_x(q_0^2 + q_1^2 - q_2^2 - q_3^2) + 2b_z(q_1q_3 - q_0q_2) \\ 2b_x(q_1q_2 - q_0q_3) + 2b_z(q_2q_3 + q_0q_1) \\ 2b_x(q_1q_3 + q_0q_2) + b_z(q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{bmatrix}. \tag{3.26}$$

Isto limita o magnetômetro a influenciar apenas na estimativa do ângulo de guinada, a qual o acelerômetro é insuficiente para estimar (WATSON, 2013, p. 30).

Os mapeamentos do acelerômetro e do magnetômetro podem ser combinados de forma a obter a função não linear de medições $h(\hat{\mathbf{x}}_k)$ e o seu jacobiano $\mathbf{H}$, ambos apresentados nas equações

$$h(\hat{\mathbf{x}}_k) = \begin{bmatrix} a_x \\ a_y \\ a_z \\ m_x \\ m_y \\ m_z \end{bmatrix} = \begin{bmatrix} -2(q_1q_3 - q_0q_2) \\ -2(q_2q_3 + q_0q_1) \\ -q_0^2 + q_1^2 + q_2^2 - q_3^2 \\ b_x(q_0^2 + q_1^2 - q_2^2 - q_3^2) + 2b_z(q_1q_3 - q_0q_2) \\ 2b_x(q_1q_2 - q_0q_3) + 2b_z(q_2q_3 + q_0q_1) \\ 2b_x(q_1q_3 + q_0q_2) + b_z(q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{bmatrix} \tag{3.27}$$

$$\mathbf{H} = \frac{\partial h}{\partial \hat{\mathbf{x}}} = \begin{bmatrix} 2q_2 & -2q_3 & 2q_0 & -2q_1 & 0 & 0 & 0 \\ -2q_1 & -2q_0 & -2q_3 & -2q_2 & 0 & 0 & 0 \\ -2q_0 & 2q_1 & 2q_2 & -2q_3 & 0 & 0 & 0 \\ 2(q_0b_x - q_2b_z) & 2(q_1b_x + q_3b_z) & 2(-q_2b_x - q_0b_z) & 2(-q_3b_x + q_1b_z) & 0 & 0 & 0 \\ 2(-q_3b_x + q_1b_z) & 2(q_2b_x + q_0b_z) & 2(q_1b_x + q_3b_z) & 2(-q_0b_x + q_2b_z) & 0 & 0 & 0 \\ 2(q_2b_x + q_0b_z) & 2(q_3b_x - q_1b_z) & 2(q_0b_x - q_2b_z) & 2(q_1b_x + q_3b_z) & 0 & 0 & 0 \end{bmatrix} \tag{3.28}$$

O modelo de medições $h(\hat{\mathbf{x}}_k)$ mapeia a estimativa do estado no espaço do vetor $\mathbf{z}$, permitindo comparar o valor da predição com os obtidos através dos sensores.

As matrizes $\mathbf{Q}$ e $\mathbf{R}$ são definidas segundo as equações

$$\mathbf{Q} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0,2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0,2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0,2 \end{bmatrix} \tag{3.29}$$

$$\mathbf{R} = \begin{bmatrix} 5\times10^5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5\times10^5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5\times10^5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10^7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10^7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10^7 \end{bmatrix}, \tag{3.30}$$

respectivamente.

O quatérnio estimado pode ser convertido nos ângulos de Euler (arfagem, rolagem e guinada) segundo as Equações 3.31, 3.32 e 3.33 (WATSON, 2013, p. 31).

$$\text{Ângulo de arfagem}(\theta) = \tan^{-1}\left(\frac{2(q_0q_1 + q_2q_3)}{1 - 2(q_1^2 + q_2^2)}\right) \tag{3.31}$$

$$\text{Ângulo de rolagem}(\phi) = \sin^{-1}(2(q_0^2 - q_3^1)) \tag{3.32}$$

$$\text{Ângulo de guinada}(\psi) = \tan^{-1}\left(\frac{2(q_0q_3 + q_1q_2)}{1 - 2(q_2^2 + q_3^2)}\right) \tag{3.33}$$

3.6 Obtenção das velocidades lineares

O modelagem utilizada para estimar as velocidades lineares do helicóptero é simplificada de tal forma que se usa um sistema linear para representar o processo. Portanto, a técnica de filtragem empregada para a finalidade desejada é o filtro de Kalman original. A equação

$$\hat{\mathbf{x}} = [v_x \quad v_y \quad a_x \quad a_y]^T \tag{3.34}$$

apresenta o vetor de estados do filtro.

As velocidades a serem estimadas são as referentes aos eixos x e y do helimodelo, obtidas através da integração dos valores de aceleração fornecidos pelo acelerômetro. Assim, a matriz de transição de estados $\mathbf{F}$ é dada por

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.35}$$

onde Δt representa o período de amostragem dos dados sensoriais.

Como descrito anteriormente, as velocidades são obtidas por meio dos dados sensoriais provenientes do acelerômetro. Portanto, o vetor de medições $\mathbf{z}$ é composto pelos valores de aceleração nos eixos x e y, de tal forma que

$$\mathbf{z} = [acc_x \quad acc_y]^T. \tag{3.36}$$

Como tais acelerações compõem o vetor de estados, a matriz $\mathbf{H}$ que mapeia o espaço de $\mathbf{x}$ no espaço de medições (vetor $\mathbf{z}$) é dada pela equação

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.37}$$

Por fim, as matrizes de covariância $\mathbf{P}$, $\mathbf{Q}$ e $\mathbf{R}$ foram obtidas através de testes experimentais de tal modo que os resultados fossem satisfatórios. As equações

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.38}$$

$$\mathbf{Q} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0.0002 & 0 \\ 0 & 0 & 0 & 0.0002 \end{bmatrix} \tag{3.39}$$

$$\mathbf{R} = \begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix} \tag{3.40}$$

mostram tais matrizes.

Definidas todas as variáveis do filtro, basta apenas seguir a execução sequencial das Equações 3.2 à 3.6, tendo como estado inicial $\hat{\mathbf{x}}_0 = [0 \quad 0 \quad 0 \quad 0]$.

3.7 Estrutura do *software* embarcado

O *firmware*, isto é, o *software* embarcado na placa de desenvolvimento STM32F3Discovery foi desenvolvido em forma de módulos, na qual cada módulo trata de uma funcionalidade utilizada. Como mencionado anteriormente, o código fonte se baseia no pacote *STM32F3-Discovery Board Firmware Applications Package* versão 1.1.0 disponibilizado pela própria fabricante STMICROELECTRONICS (acesso em 11 de jun. 2013).

Este pacote fornece um suporte através de um *Hardware Abstraction Layer* (HAL), uma interface de programação (neste caso em linguagem C) que permite interagir com o dispositivo através de tal interface, não necessitando conhecer profundamente os seus detalhes. Assim, existem definições, estruturas de dados e funções para acessar e configurar periféricos e seus registradores, registradores do núcleo e arquivos de configuração do microcontrolador. Além disso, o pacote fornece uma interface para os recursos da placa como estruturas de dados e funções para comunicar e configurar os sensores, controlar os LEDs e ler o estado dos botões. Por fim, há um conjunto de exemplos e uma aplicação demonstrativa que já vem programada na placa.

O programa desenvolvido basicamente realiza as seguintes tarefas:

Comunicação com sensores: consiste na leitura dos dados fornecidos pelos sensores. As funções utilizadas para tal finalidade foram retiradas da aplicação demonstrativa do fabricante.

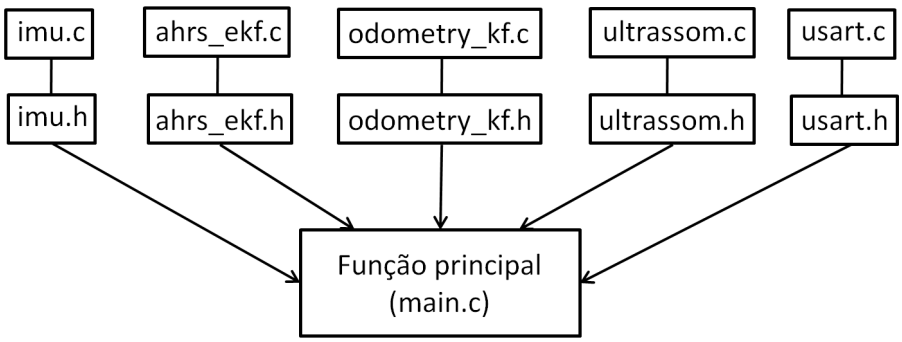
Processamento dos filtros: através da biblioteca *arm_math.h* fornecida pela ARM e contida no pacote do fabricante da placa, que contém funções matemáticas otimizadas para esta arquitetura, são realizados os cálculos dos filtros para atitude e velocidades.

Comunicação com o ultrassom: através de um temporizador configurado para a função de *input capture* e uma saída digital, executa-se o protocolo de comunicação do sensor, obtendo a distância medida.

Comunicação com o computador: consiste em usar um periférico USART para realizar uma comunicação serial entre a placa e o computador de bordo, enviando os dados obtidos.

Cada módulo é, basicamente, uma biblioteca com funções e variáveis internas. A Figura 22 ilustra a arquitetura do programa.

Figura 22: Esquemático da estrutura do *software* embarcado



Fonte: Próprio autor.

A biblioteca *imu.h* contém as funções para leitura dos dados inerciais, enquanto que a *ahrs_ekf.h* possui as funcionalidades do filtro para orientação. A *odometry_kf.h* tem as funções para o filtro que estima as velocidades lineares, a *ultrassom.h* proporciona a comunicação com o sensor de distância e, por fim, a biblioteca *usart.h* é usada para promover a comunicação serial com o computador de bordo.

A função *main.c* faz chamadas às funções das bibliotecas criadas em um período de amostragem estabelecido por uma interrupção periódica (denominada de *systick*). Desta forma, pode-se manter uma frequência de amostragem constante e de valor suficiente para o funcionamento das tarefas de controle. O Apêndice A contém o código utilizado nesta etapa do projeto.

4 VISÃO COMPUTACIONAL: FLUXO ÓPTICO

A utilização de técnicas de visão computacional tem apresentado grande eficiência no que tange à navegação e controle de robôs móveis. Além disso, com a evolução da capacidade de processamento dos sistemas computacionais, o uso de câmeras digitais para obtenção e posterior processamento de quadros de imagens tem sido um meio tecnicamente viável de sensoriamento.

A visão artificial é, assim como a visão humana, um dos meios de percepção mais poderosos, no que diz respeito à riqueza de informações que podem ser obtidas. Presença de obstáculos, objetos de interesse e conteúdo do ambiente ao redor do robô são exemplos de tais informações. Desta forma, tarefas como monitoramento, inspeção de objetos, mapeamento de terrenos e *tracking* (consiste na localização de um objeto em movimento) são possíveis, pois se baseiam em processamento de imagens.

Um sistema de visão geralmente possui baixo custo e pouco peso associado, facilitando o seu uso em robôs com carga limitada, como os veículos aéreos. Ademais, as câmeras digitais necessitam, como fonte de sinal, apenas a luz natural, além de constituírem um meio de sensoriamento não intrusivo (CAI; CHEN; LEE, 2011, p. 223).

Em robótica móvel, tais câmeras podem ser usadas para a navegação e controle dos robôs em ambientes complexos. As imagens fornecidas por estes dispositivos podem ser utilizadas para estimar a localização do robô, isto é, sua posição e orientação no espaço tridimensional, cujo nome dado às técnicas com tal finalidade é odometria (métodos de estimar a localização ao longo do tempo) visual (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011, p. 187). Pode-se ainda realizar tarefas de cooperação entre vários robôs, uma vez que se é capaz de estimar a posição de cada um deles.

Este capítulo tem a finalidade de apresentar a técnica denominada de fluxo óptico, um método de detecção de movimento através de uma sequência de quadros de imagens obtidas através de uma câmera digital. Também serão descritos a construção de um algoritmo utilizando a biblioteca de visão computacional OpenCV e a obtenção de informações a partir dos resultados do processamento das imagens.

4.1 Representação digital de um quadro de imagem

Antes de apresentar os conceitos relacionados à técnica de detecção de movimento por fluxo óptico, é importante conhecer como uma imagem é representada em meios digitais, tais como computadores e câmeras. Tal informação é fundamental para que se compreenda o funcionamento do algoritmo associado.

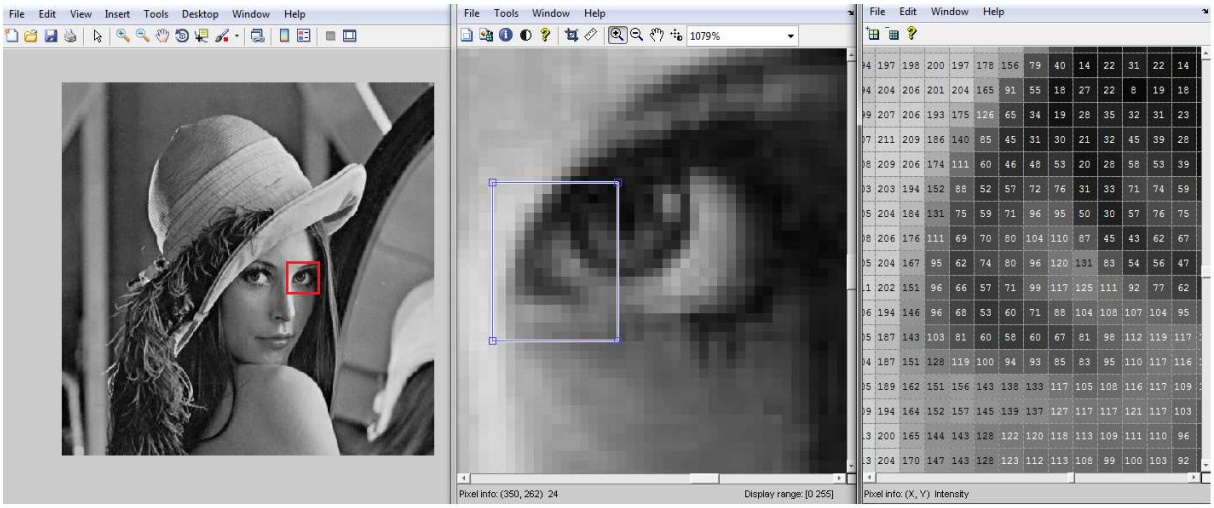
Em dispositivos de captura de imagens, como as câmeras digitais, a formação de uma imagem é realizada através da incidência de luz em uma matriz composta por um número finito de elementos sensores. Assim, uma imagem é capturada por um número discreto de pontos em um arranjo bidimensional. Do mesmo modo, em dispositivos de exibição de imagens, como monitores e telas, a formação também ocorre por meio de uma matriz de elementos.

Desta forma, uma imagem digital é representada por uma matriz bidimensional, na qual cada elemento é denominado de *pixel* (acrônimo para o termo em inglês *Picture Element*). A cada um destes elementos estão associados valores de intensidade de brilho e sua posição (horizontal e vertical) na matriz. O valor de intensidade de brilho, chamada de escala de cinza, depende de quantos *bits* são usados para codificá-lo e qual o espaço de representação de cores, como o RGB (sigla em inglês para Red, Green, Blue) (GONZALEZ; WOODS; EDDINGS, 2009, p. 13–14).

Por exemplo, um pixel no espaço de cores RGB em 8 *bits* é representado pelos valores de sua posição x e y e por três bytes, cada um associado a uma cor do espaço, na qual o valor indica a intensidade do brilho referente a tal cor, o que possibilita gerar diversas cores alterando a intensidade de brilho dos três bytes. Neste caso, a imagem é formada por três matrizes (intercaladas), cada uma referente a uma cor do espaço de representação.

A Figura 23 apresenta um quadro de imagem em escala de cinza e uma região de *pixels* com os seus valores de intensidade, obtidos através do *software* Matlab.

Figura 23: Quadro de imagem e uma região de *pixels*



Fonte: Próprio autor.

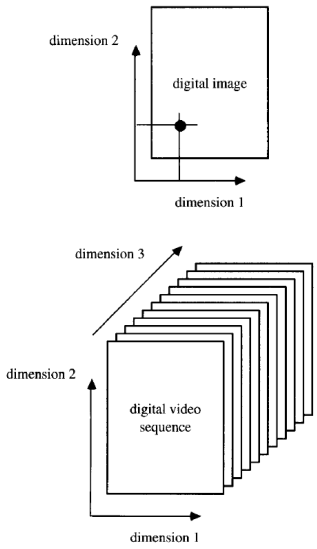
Formalmente, uma imagem é definida por uma função $I(x, y)$, na qual (x, y) são as coordenadas espaciais do pixel e I indica a intensidade do brilho ou a escala de cinza neste determinado ponto. É importante destacar que os valores de x, y e I são finitos e discretos, o que caracteriza um quadro de imagem como digital (GONZALEZ; WOODS; EDDINGS, 2009, p. 13–14).

Um vídeo digital, por sua vez, é nada mais que uma sequência de quadros de imagens, de tal modo que a imagem é definida por uma função $I(x, y, t)$, isto é, há uma coordenada temporal que indica qual a posição do quadro na sequência de imagens. Assim, um pixel localizado em (x, y) possui uma intensidade I no quadro de imagem referente ao instante t. A Figura 24 ilustra as dimensões de um quadro de imagem e de um vídeo.

4.2 Fluxo óptico

Em problemas de navegação e controle de robôs móveis, a detecção de movimento é fundamental para a realização de tais tarefas. Através da visão computacional é possível estimar o movimento de objetos ou da própria câmera (a detecção do movimento da câmera é chamada de *egomotion*), determinando assim a localização e velocidade do robô ou de obstáculos no ambiente. Com esta finalidade, existem técnicas envolvendo fluxo

Figura 24: Dimensões de um quadro de imagem e de um video



Fonte: BOVIK (2009).

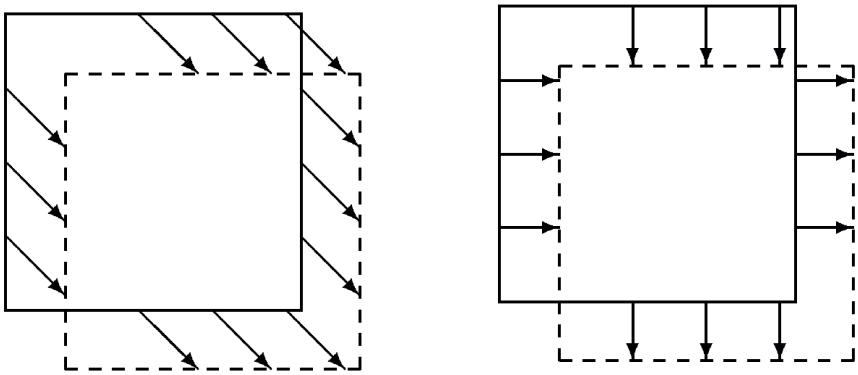
óptico que permitem estimar tais movimentos utilizando apenas uma câmera.

O fluxo óptico é definido como a distribuição de velocidades aparentes do movimento do padrão de brilho através do plano da imagem. Isto é, a formação de um campo vetorial de velocidades de cada pixel de um quadro de imagem. O fluxo óptico normalmente é tomado como resultado de um movimento relativo entre objetos e a câmera, considerando uma sequência de quadros de imagem ao longo do tempo (CALDEIRA, 2002, p. 19). A Figura 25 ilustra o fluxo óptico.

Na Figura 25(a) observa-se o movimento real do objeto (quadrado) na diagonal, enquanto que na Figura 25(b) mostram-se os vetores de fluxo óptico, sendo suas componentes vertical e horizontal. Ademais, o fluxo óptico utiliza pelo menos dois quadros de imagem sucessivos para estimar o deslocamento, mas podendo até requerer mais, dependendo do método empregado.

O campo de movimento é um campo vetorial que representa o movimento de pontos pertencentes a objetos no mundo real, projetados no plano da imagem. O campo de movimento, de fato, representa o deslocamento verdadeiro dos objetos. Campo de fluxo óptico e o campo de movimento podem ser iguais ou até mesmo diferentes, dependendo do tipo de movimento efetuado pelo objeto em questão. Entretanto, diferente do campo

Figura 25: Fluxo óptico do movimento de um objeto



(a) Movimento do objeto (b) Vetores do fluxo óptico

Fonte: MALLOT (2000).

de movimento, o fluxo óptico é observável (CALDEIRA, 2002, p. 19–20).

Existem diversos tipos de métodos que resultam no fluxo óptico. Há os baseados no gradiente, nos quais o fluxo é calculado a partir das derivadas espaciais e temporal do brilho da imagem, e os baseados em correlação, que obtêm o fluxo óptico pelo deslocamento de uma região entre duas imagens consecutivas. Os métodos baseados em energia de saída de filtros (também chamados de métodos baseados em frequência) usam a Transformada de Fourier, enquanto que os métodos baseados na fase analisam o comportamento da fase da saída de fitlros passa-banda (CALDEIRA, 2002, p. 20–21).

Neste trabalho será considerado apenas o algoritmo de Lucas e Kanade (1981), que se baseia no cálculo do gradiente.

4.2.1 Métodos baseados no gradiente

O métodos baseados no gradiente utilizam os gradientes espacial e temporal calculados de pelo menos dois quadros de imagem consecutivos. Partem da premissa que o brilho de um ponto em uma sequência de imagens é constante ao longo do tempo (a iluminância do ambiente não muda), sendo que o intervalo de tempo dt é suficientemente pequeno (CALDEIRA, 2002, p. 21–22). Assim, considerando-se $I(x, y, t)$ a função de uma imagem que especifica o brilho de um pixel na posição (x, y) no instante t de captura do quadro,

tem-se que

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \tag{4.1}$$

Portanto, assume-se que ocorre um pequeno deslocamento do padrão de brilho em (x, y, t) para a posição $(x + dx, y + dy, t + dt)$, sendo $I(x, y, t)$ constante em um período de tempo pequeno (o intervalo entre a captura das duas imagens é dt). Nos métodos baseados no gradiente supõe-se que os movimentos são pequenos ou locais (CALDEIRA, 2002, p. 21–22).

Pela expansão em uma Série de Taylor, obtém-se

$$I(x + dx, y + dy, t + dt) = I(x, y, t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt + \ldots \tag{4.2}$$

onde as derivadas de ordem superior são desconsideradas, por se tratar de uma vizinhança pequena de (x, y, t).

Das Equações 4.1 e 4.2 pode-se concluir que

$$\frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt = 0 \tag{4.3}$$

Derivando-se a Equação 4.3 em relação à t

$$\frac{\partial I(x, y, t)}{\partial t} = 0 \tag{4.4}$$

A Equação 4.4 pode ser reescrita como

$$I_x u + I_y v + I_t = 0, \tag{4.5}$$

onde $I_x = \frac{\partial I}{\partial x}$, $I_y = \frac{\partial I}{\partial y}$ e $I_t = \frac{\partial I}{\partial t}$. A variável u representa a velocidade do padrão de brilho no eixo x ($u = \frac{\partial x}{\partial t}$), e v denota a velocidade no eixo y ($v = \frac{\partial y}{\partial t}$).

Dessa forma, pode-se escrever que

$$\begin{aligned} [I_x \quad I_y] \begin{bmatrix} u \\ v \end{bmatrix} &= -I_t \\ \nabla I \cdot \vec{\mathbf{v}} &= -I_t \end{aligned} \tag{4.6}$$

As Equações 4.4, 4.5 e 4.6 representam a denominada equação de restrição de fluxo óptico. Observe-se que existem duas variáveis e uma equação, de modo que existem infinitas soluções. A este fato denomina-se de problema de abertura do fluxo óptico (CALDEIRA, 2002, p. 21–22).

A restrição do fluxo óptico não é suficiente para determinar u e v. Portanto, cada método baseado neste procedimento adiciona uma segunda restrição de forma a tornar tal equação com solução única. Um deste métodos é o de Lucas e Kanade (1981), utilizado neste projeto, que é apresentado a seguir (CALDEIRA, 2002, p. 21–22).

4.3 Algoritmo de Lucas e Kanade

O método de Lucas e Kanade, baseado no cálculo do gradiente, considera a hipótese de que as velocidades de pixels vizinhos de um pixel na posição (x, y) são assumidas iguais, isto é, o fluxo óptico de uma região é constante. Desta forma, a uma região de N por N pixels associa-se apenas um vetor de fluxo óptico, partindo do conjunto de equações

$$\begin{aligned} I_{x_1}u + I_{y_1}v + I_{t_1} &= 0 \\ I_{x_2}u + I_{y_2}v + I_{t_2} &= 0 \\ \vdots \qquad \vdots \quad \vdots \quad &\vdots \\ I_{x_{NxN}}u + I_{y_{NxN}}v + I_{t_{NxN}} &= 0. \end{aligned}$$

O conjunto de equações anterior pode ser arranjado em um formato matricial, como

$$\underbrace{\begin{bmatrix} I_{x_1} & I_{y_1} \\ \vdots & \vdots \\ I_{x_{NxN}} & I_{y_{NxN}} \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{\vec{\mathbf{v}}} = \underbrace{\begin{bmatrix} I_{t_1} \\ \vdots \\ I_{t_{NxN}} \end{bmatrix}}_{\mathbf{b}}. \tag{4.7}$$

Desta forma, como se tem mais equações do que variáveis, o sistema é considerado sobre-determinado. O cálculo do fluxo óptico de uma região se torna, então, um problema de estimativa de Mínimos Quadrados, na qual se busca uma solução (vetor de velocidade) que minimiza a soma do erro quadrático de cada equação do conjunto. As Equações 4.8, 4.9 e 4.10 mostram o desenvolvimento da solução através do método dos Mínimos Quadrados.

$$\mathbf{A} \times \vec{\mathbf{v}} = \mathbf{b} \tag{4.8}$$

$$(\mathbf{A}^T\mathbf{A}) \times \vec{\mathbf{v}} = \mathbf{A}^T\mathbf{b} \tag{4.9}$$

$$\vec{\mathbf{v}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{4.10}$$

A Equação 4.10 pode ser reescrita como

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{NxN} I_{x_i}^2 & \sum_{i=1}^{NxN} I_{x_i} I_{y_i} \\ \sum_{i=1}^{NxN} I_{y_i} I_{x_i} & \sum_{i=1}^{NxN} I_{y_i}^2 \end{bmatrix} \begin{bmatrix} -\sum_{i=1}^{NxN} I_{x_i} I_{t_i} \\ -\sum_{i=1}^{NxN} I_{y_i} I_{t_i} \end{bmatrix}. \tag{4.11}$$

É importante destacar que o algoritmo de Lucas e Kanade pode apresentar uma matriz adicional **W** que atribui pesos aos valores dos pixels da vizinhança de modo que a solução passa a ser (CALDEIRA, 2002, p. 28):

$$\vec{\mathbf{v}} = (\mathbf{A}^T\mathbf{W}^2\mathbf{A})^{-1}\mathbf{A}^T\mathbf{W}^2\mathbf{b} \tag{4.12}$$

Existem bibliotecas com funções que executam o algoritmo apresentado. Entre elas está a biblioteca OpenCV, que será apresentado a seguir.

4.3.1 Biblioteca OpenCV

A OpenCV é um acrônimo para o termo em inglês *Open Source Computer Vision library* ou biblioteca de código livre para visão computacional. Esta biblioteca provê estruturas de dados e funções nas áreas de processamento de imagens e visão computacional, e foi escrita em linguagem de programação C e C++, além de ter a capacidade de ser executada em diversas plataformas (Linux, Windows, entre outras), sendo bastante utilizada no meio acadêmico (BRADSKI; KAEHLER, 2008, p. 1–3).

A biblioteca possui funções para o cálculo do fluxo óptico e dentre tais funções está o algoritmo de Lucas e Kanade. O protótipo da função é *cvCalcOpticalFlowLK()* e recebe como argumentos dois quadros de imagem sucessivos, o tamanho da região de agrupamento de pixels (janela) e retorna as componentes horizontal e vertical do fluxo óptico.

Assim, OpenCV permite que se capture quadros de imagem de uma câmera conectada a um computador e se efetue o processamento das imagens.

Neste projeto, propõe-se utilizar uma câmera USB conectada ao computador de bordo, que realizará o processamento das imagens. A câmera será acoplada ao helicóptero e estará orientada de modo a apontar para o solo. O objetivo é calcular o fluxo óptico obtido através do movimento relativo entre o helimodelo (com a câmera acoplada) e o solo de tal forma a estimar as velocidades lineares do helicóptero.

Entretanto, as velocidades resultantes do fluxo óptico estão na unidade de medida de pixels por segundo. Desta forma, é necessário realizar a calibração da câmera e converter a unidade de medida para metros por segundo, o que está fora do espoco deste projeto, sendo uma sugestão para trabalhos futuros.

A utilização deste sistema de visão computacional busca promover um aprimoramento dos dados de navegação obtidos dos sensores inerciais da IMU e do sensor de ultrassom. Pode-se realizar uma fusão de dados, realizada a partir dos dados provenientes dos vários sensores mencionados. Uma vez que tais dispositivos possuem limitações e problemas, a redundância de informações sensoriais tende a minimizar os erros de medição e, consequentemente, aumentar a confiabilidade e pecisão do sistema de voo autônomo (BRANDÃO, 2013, p. 75).

5 COMPUTADOR EMBARCADO

Uma vez definidos o meio de acionamento dos atuadores e a instrumentação, deve-se então especificar o sistema embarcado que deverá executar as tarefas básicas de processamento dos dados sensoriais, realização dos cálculos envolvidos nas operações do controlador e a comunicação com o sistema de atuação, a fim de promover a navegação autônoma do helicóptero.

O computador de bordo deve ter capacidade de processamento e recursos de *hardware* que permitam executar os algoritmos das tais tarefas necessárias à navegação do robô, além de fornecer um ambiente de programação que facilite o desenvolvimento das aplicações. Ademais, o computador deverá atender a requisitos de tempo de resposta, já que em aplicações de controle e robótica a manutenção do tempo de amostragem é necessária para garantir o funcionamento correto do controlador e da instrumentação.

Neste capítulo será apresentada uma plataforma de *hardware* e *sotfware* capaz de realizar as tarefas de processamento de dados sensoriais e execução de algoritmos de controle. A plataforma computacional terá embarcado um sistema operacional baseado em Linux com um suporte de tempo real, que visa aumentar a confiabilidade do sistema proposto.

5.1 Recursos de *hardware* e *software*

O *hardware* utilizado como computador de bordo é uma SBC (sigla do termo em inglês *Single Board Computer*), isto é, um sistema computacional básico (processador, memória e dispositivos de entrada/saída) em uma única placa de circuito impresso, sendo possível portar um sistema operacional e utilizá-lo como um computador convencional.

O sistema embarcado é o modelo BD-SL-i.MX6 (também chamado de *SABRE Lite board*), da fabricante norte-americana *Boundary Devices*, apresentado na Figura 26.

Figura 26: Imagem do computador *SABRE Lite board*

Fonte: *Boundary Devices.*

A plataforma possui como elemento principal um SOC (do termo inglês *System on Chip*), circuito integrado que contém diversos componentes de um sistema eletrônico (como um computador), modelo i.MX6Quad fabricado pela Freescale. É um processador com quatro núcleos da arquitetura Cortex-A9 da desenvolvedora ARM operando à $1,0GHz$ (podendo chegar a $1,2GHz$).

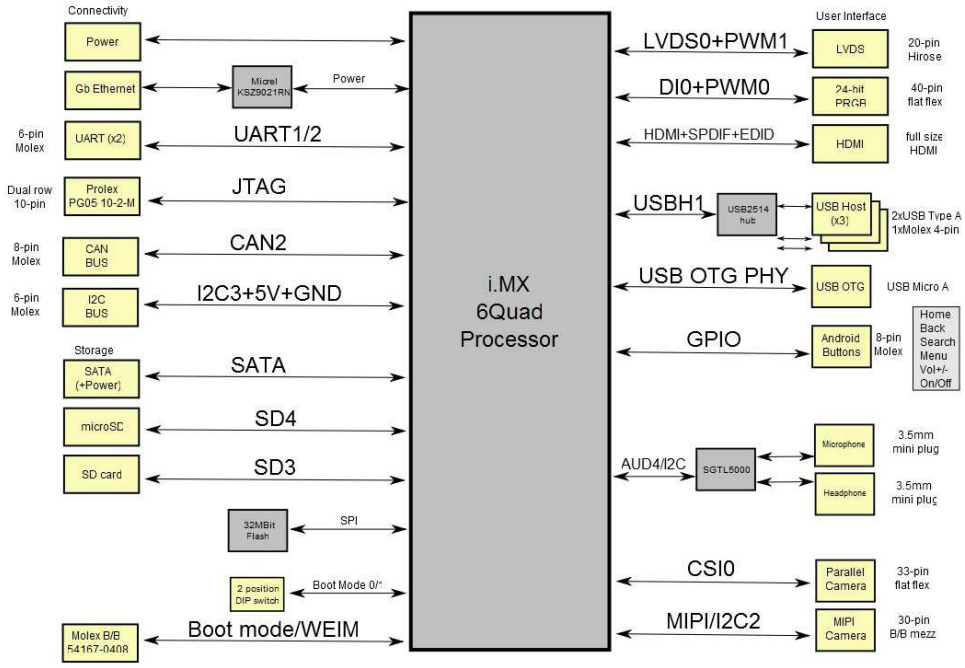
Este processador contém $32KB$ de memória cache L1 para dados e $32KB$ de memória cache L1 para dados, ambas para cada núcleo e $1MB$ de memória cache L2 compartilhada entre os núcleos. Possui também diversos periféricos integrados aos núcleos através de um barramento. Além de outros recursos, o processador contém ainda:

- Memória ROM de $96KB$ e RAM de $256KB$.
- Uma unidade de ponto flutuante, três unidades de processamento gráfico, duas unidades de processamento de imagem, uma unidade de processamento de vídeo e o co-processador NEON da ARM para multimídia.
- Interfaces para câmeras (sensores), monitores e telas.
- Interfaces para alguns tipos de memórias (como RAM e NAND-*flash*), disco rígido (SATA) e cartão de memória.

- Interfaces de comunicação: USB, serial assíncrona (cinco UARTs), I2C, SPI, Ethernet, CAN, entre outras.
- Possui diversas portas de entrada e saída digitais e quatro canais de sinais PWM.

Os recursos do computador de bordo incluem $1GB$ de memória RAM DDR3, conectores para monitores e câmeras, conector para porta SATA, Ethernet, USB, interface para cartões de memória, interface I2C, portas de entrada e saída digitais e dois conectores para portas seriais RS-232 através das UARTs, entre outras. O *bootloader* U-Boot, que promove a inicialização do processador e carrega o sistema operacional para memória, está armazenado em uma memória serial EEPROM SPI-NOR. A Figura 27 ilustra um diagrama esquemático dos dispositivos no computador.

Figura 27: Diagrama de blocos do computador *SABRE Lite board*



Fonte: *Boundary Devices.*

Um sistema operacional pode ser portado por meio de um cartão microSD que o contenha. O sistema operacional é a distribuição Linux Ubuntu com interface LXDE (Lubuntu), fornecida pela Linaro, uma organização sem fins lucrativos que desenvolve ferramentas e *software* de código live (como Linux) para a arquitetura ARM. A versão do SO (sistema operacional) é a Linaro ALIP 13.09 (sigla em inglês para *ARM Linaro Internet Platform*, ou plataforma ARM de Internet da Linaro, em português) que se baseia na versão 3.0.35-4.1.0 do *kernel* Linux, modificado pela fabricante da placa *Boundary Devices*, a qual

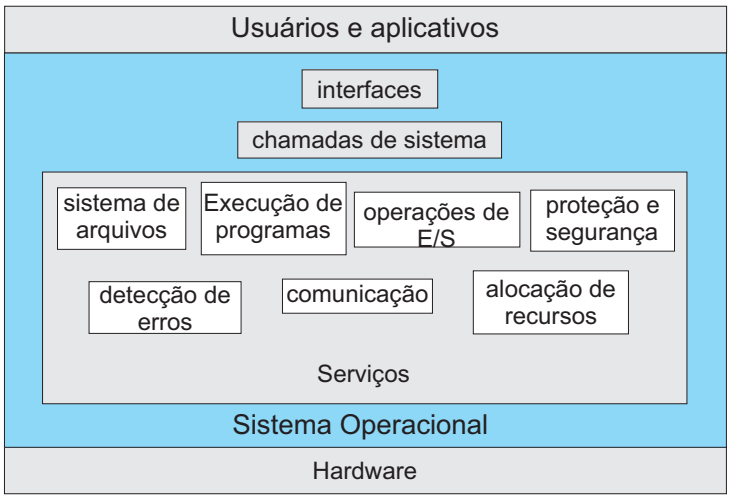
também disponibiliza a imagem para o cartão microSD já pronta, isto é, todo o sistema operacional.

5.2 Conceitos de sistemas operacionais

A programação direta em microprocessadores e microcontroladores, isto é, em baixo nível, se tornou extremamente complexa e pouco flexível, causada pela evolução da microeletrônica. A exigência das aplicações, em termos de funcionalidades, resultou em um aumento na complexidade e nos recursos presentes nestes dispositivos, que passaram a ter mais capacidade de processamento e a disponibilizar mais serviços aos seus usuários. Assim, o projeto de *software* passou a requerer bastante tempo de desenvolvimento. A solução de tal problema reside em utilizar um sistema operacional, o que permite ao desenvolvedor se preocupar apenas em trabalhar na aplicação em si (TANENBAUM, 2009, p. 1).

Um sistema operacional (sigla SO) é um *software* que gerencia o *hardware* do computador, isto é, atua como um alocador de recursos como memória, uso do processador e de acesso a dispositivos de entrada/saída para as aplicações (SILBERSCHATZ; GALVIN; GAGNE, 2013, p. 3–5). Também age como um intermediário entre o usuário e estes recursos de *hardware*, promovendo um modelo abstrato de computador mais adequado para o operador da máquina. O SO fornece uma interface de programação (abstração) mais flexível do *hardware* e controla a execução dos vários possíveis programas (TANENBAUM, 2009, p. 1–2). A Figura 28 ilustra a estruturação de um computador.

Figura 28: Estrutura de um computador do ponto de vista de sistema operacional



Fonte: Próprio autor.

Um SO é constituído basicamente pelos seguintes componentes (STALLINGS, 2012, p. 48–52):

Núcleo (*kernel*): é o responsável pela gerência do processador, controle, comunicação e sincronização entre processos e tratamento de interrupções. Pode-se considerá-lo o principal componente de um SO.

Gerente de memória: atua no controle da memória primária e na sua alocação aos programas em execução.

Sistema de entrada/saída (E/S): responsável pela execução das operações de E/S e controle dos dispositivos periféricos.

Sistema de Arquivos: responsável pelo controle e acesso dos dados armazenados na memória secundária. Promove a abstração dos dados armazenados para o conceito de arquivos.

Processador de comandos (*shell*): promove a interface entre o usuário e o sistema operacional.

A multiprogramação é uma característica importante de um SO atual. É definida como a capacidade de um SO permitir a execução de vários programas, compartilhando o tempo de uso do *hardware* e criando a ideia de execução simultânea das aplicações. Assim, tais programas competem pelos recursos do sistema, tornando mais eficiente o uso do computador (SILBERSCHATZ; GALVIN; GAGNE, 2013, p. 19).

A partir de tal característica, um conceito básico associado aos sistemas operacionais é o de processo. Um processo é uma abstração de um programa em execução. Processos, além de conter o código do programa que representam, contam com espaços de memória para pilha, dados (variáveis) e estruturas de controle do SO (TANENBAUM, 2009, p.50–51).

Tais estruturas de dados contêm informações a respeito do estado do processo (em execução, bloqueado, suspenso, entre outros), valores dos registradores do processador (contexto), recursos utilizados, informações de gerência de memória e outros. Tais estruturas servem para o SO controlar e gerenciar os processos. Assim, o sistema operacional possui esta estrutura de dados chamada de PCB (sigla do termo inglês *Process Control Block*) ou Bloco de Controle de Processo, na qual estão as informações com tal finalidade. O SO também promove segurança e proteção entre processos, de forma que nenhum processo

pode acessar o espaço de endereçamento de outro (SILBERSCHATZ; GALVIN; GAGNE, 2013, p. 106–109).

Para compartilhar os recursos de *hardware* entre os processos, o *kernel* do SO possui um programa denominado escalonador, responsável por determinar qual o próximo processo a ser executado pelo processador. O escalonador se baseia em um algoritmo que executa uma política de escolha do próximo processo chamado de algoritmo de escalonamento. Em uma política preemptiva, o processo pode perder a posse do processador na ocorrência de certos eventos; já em uma política não-preemptiva o processo só perde a posse quando termina ou devolve deliberadamente.

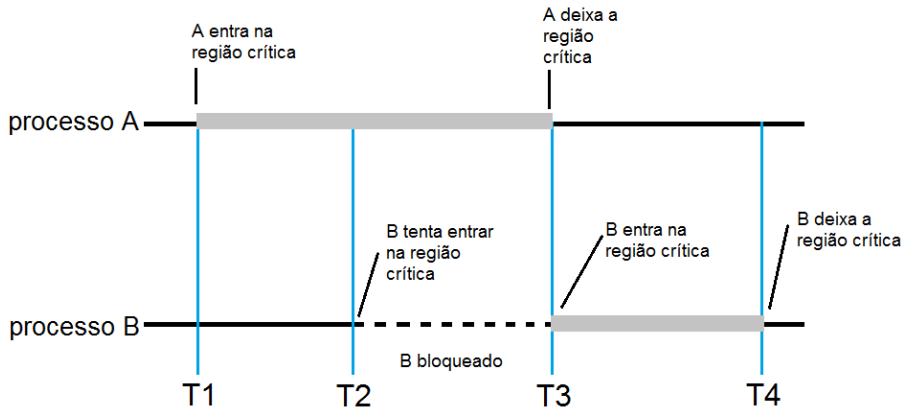
Uma vez determinado o próximo processo, ocorre a mudança de contexto. O contexto é o conjunto de informações necessárias para que o SO possa restaurar a execução do processo a partir do ponto interrompido. Por exemplo, o estado do processador (seus registradores) é um tipo de informação do contexto. Assim, o *kernel* salva em uma estrutura de dados o contexto do processo atual, e restaura o referente ao processo selecionado para ser executado (SILBERSCHATZ; GALVIN; GAGNE, 2013, p. 114). Para o processo em execução, é como se este detivesse todos os recursos do *hardware*, de forma que cada processo se vê como único no computador.

Outro serviço importante que o *kernel* oferece é a sincronização entre processos. A sincronização visa proteger os dados compartilhados e promover o acesso ordenado aos recursos. Assim, quando se deseja que um recurso seja acessado em uma determinada ordem de processos, ou evitar que dois processos acessem um mesmo recurso, deve-se usar algum mecanismo de sincronização.

A sincronização é um serviço importante quando se trata de programação concorrente. A um trecho de código no qual se acessam recursos compartilhados, como arquivos, e outras operações, dá-se o nome de região crítica. Assim, deve-se evitar que dois processos, que trabalham em conjunto, acessem as suas regiões críticas ao mesmo tempo. Um meio de evitar tal problema é através de semáforos, que são exemplos de métodos de sincronização. Semáforo é uma variável inteira usada para bloquear um processo quando outro está em sua região crítica. De forma similar, existe o mutex (acrônimo de *Mutual Exclusion*) que realiza a mesma função de um semáforo. Entretanto, o mutex pode ter apenas os valores

zero e um (SILBERSCHATZ; GALVIN; GAGNE, 2013, p. 212–217). A Figura 29 ilustra a ação de um mutex ou um semáforo.

Figura 29: Mecanismo de sincronização de processos



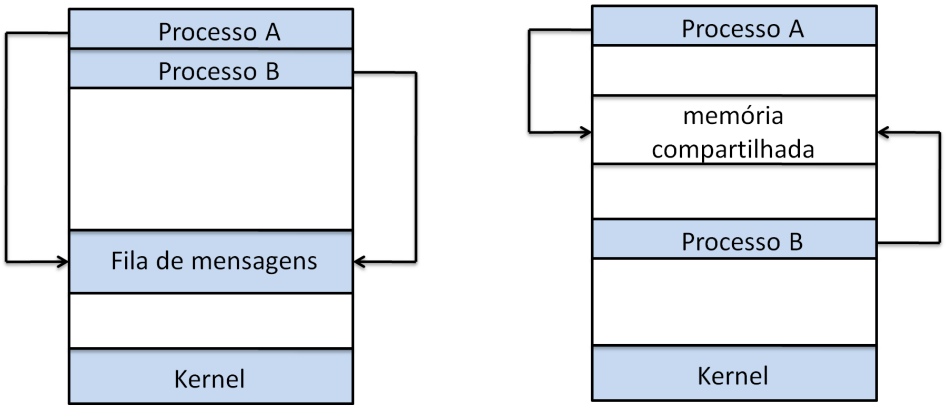
Fonte: Próprio autor.

O *kernel* também fornece mecanismos para comunicação entre processos (IPC - *Inter Process Communication*), isto é, a troca de informação entre tais processos. Como o sistema operacional assegura a independência entre processos, isto é, garante que um processo não afete outro, estes mecanismos são o meio de cooperação entre processos. Desta forma, processos que usam estes mecanismos são chamados de processos cooperativos (SILBERSCHATZ; GALVIN; GAGNE, 2013, p. 122).

Os mecanismos de IPC permitem que atividades ocorram mais rapidamente, uma vez que há a divisão de tarefas, além de fornecer um ambiente modular, onde cada processo tem um função especifica. Existem duas abordagens de comunicação: memória compartilhada e a passagem de mensagens (*pipes e filas)*. A memória compartilhada é uma área de memória comum aos processos cooperativos. As *pipes* são estruturas do tipo FIFO (do termo inglês *First-In First-Out*) criadas pelo *kernel* em tempo de execução e acessadas por descritores de arquivo. As filas são também estruturas FIFO, mas estão mantidas no sistema de arquivos (SILBERSCHATZ; GALVIN; GAGNE, 2013, p. 124–129). A Figura 30 ilustra as duas abordagens de IPC.

Os conceitos apresentados são importantes para a compreensão e desenvolvimento de programas utilizando-se dos serviços de um sistema operacional. Com tal conhecimento,

Figura 30: Mecanismos de comunicação entre processos



(a) Passagem de mensagens (b) Memória compartilhada

Fonte: Próprio autor.

pode-se criar de forma mais eficiente algoritmos para diversas aplicações. No caso deste trabalho, o *software* proposto a seguir utiliza tais recursos.

5.2.1 Linux embarcado

Como abordado anteriormente, o uso de um SO em sistemas embarcados fornece um meio ágil, flexível e dinâmico para o desenvolvimento de aplicações à medida que se necessita de mais funcionalidades. Uma alternativa como sistema operacional embarcado com grande desenvolvimento e larga utilização comercial e acadêmica são os sistemas operacionais baseados no *kernel* denominado Linux (YAGHMOUR, 2009, p. 2–3).

O Linux foi desenvolvido para ser utilizado em sistemas operacionais de propósitos gerais, como computadores pessoais e até servidores. Entretanto, apesar de não ser projetado (nem otimizado) para aplicações embarcadas, devido às suas qualidades técnicas e econômicas, além do desenvolvimento da eletrônica embarcada que o permitiu ser portável, o Linux passou a ser usado em diversos sistemas (YAGHMOUR, 2009, p. 3). Deve-se mencionar que, diferentemente de um computador pessoal, um sistema embarcado é um sistema computacional cujo objetivo é executar tarefas de aplicações específicas, como o computador de bordo.

Dentre as qualidades dos sistemas operacionais Linux que motivam o seu uso, podem-se destacar as seguintes características (KARIM, 2009, p. 8–11; HALLINAN, 2011, p. 2–3):

Confiabilidade de código: pela utilização do Linux, o *kernel* apresenta um código flexível, robusto e de qualidade no que diz respeito ao desempenho.

Suporte a *hardware*: o Linux fornece amplo suporte para diversos tipos de arquitetura como x86, ARM, Power PC, MIPS, entre outras. Além disso, um grande número de *drivers* para dispositivos é mantido pela comunidade Linux.

Modularidade e escalabilidade: o código do Linux é estruturado em forma de módulos para cada tipo de funcionalidade, de forma que o torna mais fácil de trabalhar. Além disso, é possível adicionar novos aspectos ao código e realizar modificações, sendo adaptável para diversas plataformas.

Configurabilidade pode-se selecionar quais recursos do código farão parte da aplicação final, isto é, o Linux permite um alto grau de personalização.

Suporte à conectividade: o Linux proporciona um amplo suporte a padrões de protocolos de rede e *softwares* de comunicação.

Uma característica muito importante sobre o Linux é o fato de seu código fonte ser aberto, o que o torna um *software* livre (*Open Source*). Isto implica que o desenvolvedor tem total acesso a qualquer linha de código, podendo otimizá-lo ou conhecê-lo melhor, além de ter acesso a várias ferramentas disponíveis. Pelo fato do Linux ser de código livre, formou-se uma comunidade constituída por entusiastas, acadêmicos e grandes empresas com a finalidade de desenvolver ferramentas e fornecer amplo suporte ao Linux, além de produzir um grande volume de informações sobre este *software*. Esta talvez seja a maior vantagem do Linux (YAGHMOUR, 2009, p. 11).

Outro ponto importante é a respeito do custo econômico. Uma vez que o Linux é de código livre, o seu licenciamento é gratuito, além de existirem diversas ferramentas também gratuitas para o desenvolvimento de um projeto. As próprias fabricantes do computador de bordo e do processador, Boundary Devices e a Freescale, respectivamente, fornecem código aberto e suporte no desenvolvimento do Linux em seus produtos (HALLINAN, 2011, p. 4–5).

Apresentadas tais características, neste projeto, então, decidiu-se basear a construção do *software* em um ambiente Linux, como o descrito anteriormente. Entretanto, para garantir a confiabilidade da execução das atividades de controle e instrumentação do helicóptero

no ambiente Linux, optou-se por integrar um suporte de tempo real com tal finalidade. Esta estrutura de tempo real é apresentada a seguir.

5.3 Suporte de tempo real para Linux

Existem aplicações nas quais o processamento de eventos em tempo hábil se faz necessário, isto é, o sistema que executa as tarefas deve responder tão rápido quanto o desejável. Para tais aplicações, deve-se utilizar um sistema de tempo real (*RTS*, sigla do termo em inglês *real-time systems*), de tal forma que se garantam resultados satisfatórios de suas funções (LAPLANTE, 2004, p. 1; BUTTAZZO, 2011, p. 1).

5.3.1 Sistemas de tempo real

Um sistema de tempo real é um sistema computacional (conjunto de *hardware* e *software*) cuja resposta deve ser entregue dentro de um limite de tempo estabelecido. Deste modo, deve-se obedecer a restrições de tempo entre o início e o final da tarefa a ser executada. Tal sistema deve ser capaz de realizar seus processos em um intervalo específico de tempo, caso contrário os atrasos podem ocasionar falhas graves ou redução do desempenho (LAPLANTE, 2004, p. 4).

Atividades como as relacionadas à indústria aeronáutica necessitam do uso de sistemas de tempo real para o seu desenvolvimento. O controle da propulsão de aeronaves, leitura dos sensores e comunicação são exemplos de processos nos quais softwares devem ser capazes de gerenciar várias tarefas de diversos estados críticos. O atraso no processamento de uma destas atividades pode resultar em falhas graves, ou até mesmo em catástrofes (YAGHMOUR, 2009, p. 311).

Em sistemas de automação industrial e robótica, o uso de recursos de tempo real é necessário para a realização de aplicações envolvendo controle de processos e instrumentação. Garantia dos tempos de amostragem dos dados e o processamento de cálculos de controle no ritmo certo são tarefas importantes para o bom desempenho de um sistema de automação. Outro fator que justifica o uso de plataformas de tempo real é a interação com o ambiente externo, como comunicação e acionamento de outros dispositivos, que devem ser realizadas com rapidez (YAGHMOUR, 2009, p. 311).

Sistemas de tempo real podem ser classificados quanto à necessidade de atendimento às restrições de tempo em sistemas *soft*, *hard* e *firm* (LAPLANTE, 2004, p. 5–6).

Em um *soft RTS* o não atendimento dos prazos de tempo resulta em degradação do desempenho da aplicação, como em sistemas de multimídia. Por outro lado, em um *hard RTS* o não cumprimento de uma restrição causa falha grave ou desastre, como em sistemas aeronáuticos. Por fim, em um *firm RTS* a perda de alguns prazos provoca a redução de desempenho do sistema e, caso isto se prolongue, pode resultar em falha, como um meio termo entre *soft* e *hard*. Como exemplo para esta última classe, pode-se citar o controle de robôs autônomos (LAPLANTE, 2004, p. 5–6).

Sistemas de tempo real devem ser determinísticos, isto é, para cada conjunto de sinais de entrada em certo estado do sistema, um único conjunto de saídas é possível, e o próximo estado pode ser determinado. Se o tempo de resposta de cada saída é conhecido, então o sistema possui determinismo temporal. Isto é necessário, pois um sistema de tempo real deve garantir que sempre será capaz de responder em qualquer instante e quando irá responder (determinismo temporal) (LAPLANTE, 2004, p. 9–10).

Neste projeto, a finalidade do computador de bordo é executar tarefas de controle do helicóptero e comunicação com os sistemas de instrumentação e acionamento. Portanto, devido ao tipo de aplicação, deve-se utilizar um sistema de tempo real de forma a garantir a integridade de tais tarefas.

Entretanto, o sistema portará um sistema operacional baseado em Linux, que não possui qualidades de um sistema de tempo real. O escalonador do Linux busca compartilhar os recursos de uma forma mais justa entre os processos, em contraste com o escalonador de um sistema de tempo real, que deve priorizar a execução dos processos mais críticos. Além disso, o *kernel* do Linux pode mascarar interrupções e executar também processos em segundo plano, que podem interferir no funcionamento das aplicações.

Para preencher tal lacuna, existem diversos projetos cuja finalidade é promover um suporte de tempo real para o Linux, tornando-o um sistema operacional de tempo real (RTOS - *Real Time Operating System*). Entre estes projetos se encontra o Xenomai, que foi selecionado para este trabalho, uma vez que fornece suporte à arquitetura utilizada.

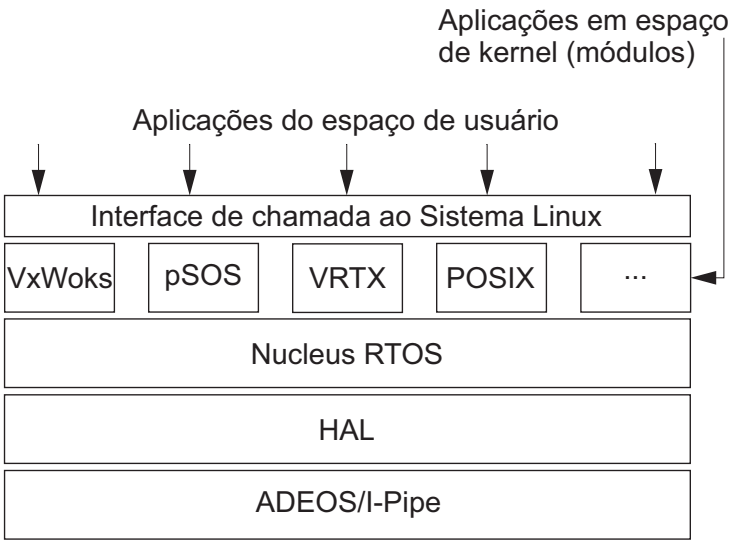
5.3.2 Projeto Xenomai

O Xenomai é um subsistema de tempo real que pode ser fortemente integrado ao *kernel* do Linux. O projeto se baseia na abordagem de *kernel* dual, na qual um *nanokernel* é executado ao lado do *kernel* do Linux, sobre o mesmo hardware. O *co-kernel* é o responsável pelo gerenciamento das tarefas de tempo real que são executadas no ambiente Linux (YAGHMOUR, 2009, p. 317).

O escalonamento é baseado em prioridades, e cada tarefa possui um número inteiro associado, indicando a sua prioridade. Assim, os recursos de *hardware* são alocados à tarefa de maior prioridade que está em espera para ser executada. No Xenomai, o usuário define a prioridade de cada tarefa desenvolvida (GERUM, 2004, p. 7–8).

A estrutura completa do Xenomai pode ser vista como um conjunto de camadas. A Figura 31 apresenta a arquitetura do Xenomai.

Figura 31: Arquitetura do suporte de tempo real Xenomai



Fonte: Próprio autor.

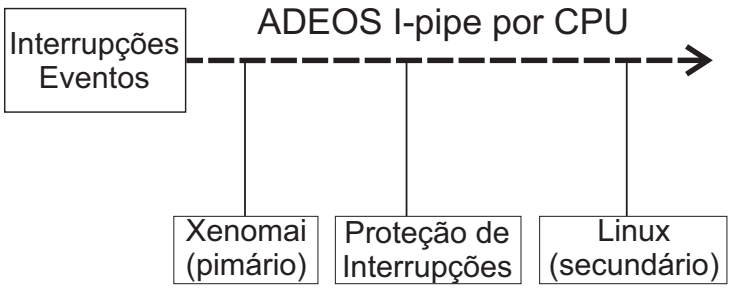
Acima do hardware se encontra uma estrutura de software chamada de I-*pipe*. A I-*pipe* deriva do sistema ADEOS (sigla em inglês de *Adaptive Domain Environment for Operating Systems*), cuja função é disponibilizar os recursos de hardware para múltiplos sistemas operacionais como um meio de virtualização (GERUM, 2005, p. 4–7). A I-*pipe*, por sua vez, é uma forma simplificada do ADEOS, sendo uma camada de virtualização com função de um controlador de interrupções programável (I-*pipe* deriva de *interruption pipeline*).

Como o Linux pode adiar interrupções externas, o que não pode ocorrer em sistemas críticos, a I-*pipe* cria máscaras de interrupção separadas (YAGHMOUR, 2009, p. 320–321).

Nesta abordagem de *kernel* dual, o *co-kernel* possui total controle das interrupções e recursos do processador. Assim, a estrutura I-*pipe* trabalha com o conceito de domínios, no qual o sistema completo é organizado como um conjunto de tais entidades conectadas formando uma *pipeline*. Estes domínios podem ser sistemas operacionais, módulos do *kernel* ou componentes de software baseados em *kernel*. Desta forma, a I-*pipe* promove os serviços de despacho de interrupções, exceções e chamadas de sistema para cada domínio, de acordo com a sua prioridade (YAGHMOUR, 2009, p. 320–321).

O Xenomai é o domínio de mais alta prioridade na *pipeline*, enquanto que o *kernel* Linux padrão é o de menor prioridade. Desta forma, o Linux é tratado pelo escalonador de tempo real como um processo em estado inativo, de tal modo que o Linux padrão somente é escalonado quando não há tarefas de tempo real a serem executadas. Os eventos enviados pela I-*pipe* são analisados pelos tratadores do Xenomai antes de serem transferidos ao domínio Linux, caso estes eventos sejam do domínio do *kernel* padrão. Os eventos têm um fluxo do domínio de maior prioridade para o de menor prioridade (GERUM, 2005, p. 4–7). A Figura 32 apresenta um esquema da I-*pipe* e os tais domínios.

Figura 32: Estrutura da I-*pipe*



Fonte: Próprio autor.

A I-*pipe* permite que o Xenomai trate todas as interrupções antes do *kernel* Linux, impedindo-o de mascará-las. Isto garante que o Xenomai tenha latências de interrupções previsíveis em níveis de microssegundos, independente do que está em execução no Linux. Assim, a I-*pipe* conta com uma estrutura que armazena as interrupções do Linux, denominada de proteção de interrupções (*Interrupt shield*) (GERUM, 2005, p. 4–7).

Acima da camada I-*pipe* se encontra a *Hardware Abstraction Layer* (HAL, camada de abstração de hardware, em português). Esta camada compreende todo o código dependente da arquitetura utilizada, promovendo acesso direto ao hardware. Assim, os serviços oferecidos permitem que os códigos de camadas superiores sejam independentes da plataforma (YAGHMOUR, 2009, p. 321).

Na camada seguinte se encontra o *co-kernel* do Xenomai, chamado de Nucleus, que fornece os serviços de um RTOS para as aplicações de tempo real da camada superior. O Nucleus do Xenomai é um *RTOS* abstrato, formado por componentes genéricos que podem ser executados sob qualquer API fornecida pela camada acima (YAGHMOUR, 2009, p. 321–322).

A camada superior é formada pelas API's do Xenomai, chamadas de *skins* (peles, em português). O Xenomai proporciona a emulação dos sistemas de tempo real tradicionais através da criação de APIs (do termo inglês *Application Programming Interface*) de diversos RTOS tradicionais de tal forma que permite o suporte de tempo real em um ambiente Linux para uma aplicação desenvolvida particularmente em outro RTOS. Esta é uma grande qualidade do Xenomai: a busca pela portabilidade de código. Cada *RTOS* possui peculiaridades em seus serviços, isto é, uma determinada funcionalidade em dois sistemas diferentes pode ter pequenas diferenças de comportamento que, por sua vez, podem ser relevantes para o funcionamento da aplicação (YAGHMOUR, 2009, p. 325–326).

Além das APIs dos sistemas tradicionais, o Xenomai conta com uma API própria, denominada de nativa (GERUM, 2006, p. 5–6). Esta API oferece suporte à criação de um ambiente multitarefas de tempo real com os seguintes recursos:

Gerenciamento de Tarefas: proporciona a criação de tarefas com um valor de prioridade associado e o controle da execução de tais tarefas.

Sincronização de tarefas: provê mecanismos de IPC como semáforos, mutexes e *flags* de eventos.

Comunicação entre tarefas: permite a comunicação entre tarefas através de mecanismos como filas de mensagens, pilha de memória (podem ser usadas como memória compartilhada) e *pipes*.

Gerenciamento de interrupções: proporciona o tratamento de interrupções, incluindo o tratamento de interrupções por dispositivos de E/S.

Gerenciamento de tempo: permite a contagem de intervalos de tempo e uso de temporizadores, denominados de alarmes.

Uma qualidade interessante do Xenomai é que este permite a criação de tarefas de tempo real no espaço de usuário, uma vez que os projetos somente permitiam criar tarefas no espaço de sistema, isto é em módulos do Linux. Isto torna mais flexível a criação de uma tarefa de tempo real (GERUM, 2004, p. 6).

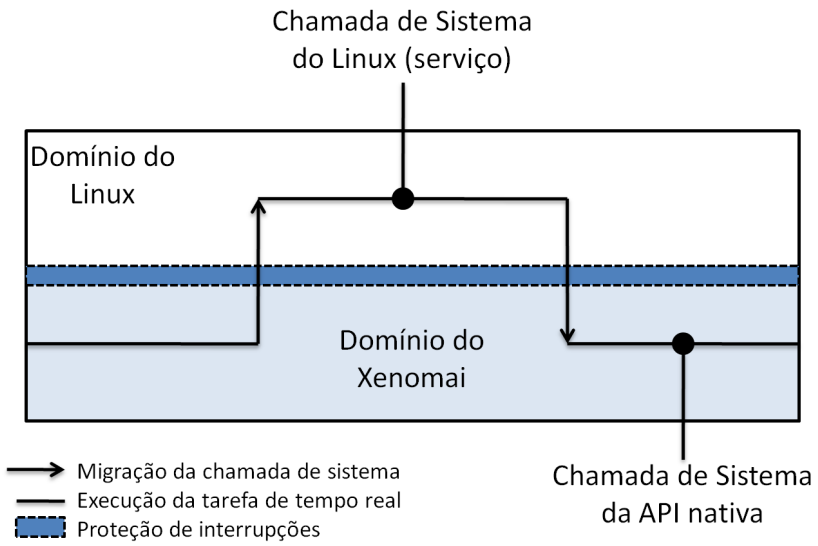
Neste aspecto, uma grande característica do Xenomai é o seu alto grau de integração com o ambiente nativo do Linux, que o diferencia de outros sistemas de kernel dual. Ademais, apesar dos domínios serem executados de forma independente, na qual o Xenomai executa as tarefas de tempo real enquanto o Linux executa os processos normais, é possível existir compartilhamento de contexto entre ambos os *kernels*. Isto é, as tarefas de tempo real podem requisitar serviços comuns do Linux padrão (YAGHMOUR, 2009, p. 326).

Assim, tal tarefa de tempo real pode ser transferida para o ambiente Linux quando esta necessitar de chamadas de sistema comum (do *kernel* padrão) e pode retornar ao contexto de tempo real quando requisitar serviços do Xenomai. Entretanto, tais serviços do *kernel* padrão, por não estarem em um domínio de tempo real, podem ter latências imprevisíveis, o que pode até ser aceitável em casos não críticos. Uma forma de evitar tal problema é bloquear as interrupções para o Linux enquanto uma tarefa de tempo real estiver neste contexto, de modo que estas interrupções não afetem a execução da tarefa. Isto é feito através da proteção de interrupções (YAGHMOUR, 2009, p. 326–327).

Estes dois domínios são chamados de primário (Xenomai), de alta prioridade, e secundário (Linux padrão), de baixa prioridade (GERUM, 2005, p. 4–6). A Figura 33 ilustra a tranferência de uma tarefa entre domínios.

Por fim, o projeto Xenomai fornece uma interface comum para a construção de *drivers* de dispositivos para aplicações de tempo real, chamada de *RTDM* (sigla em inglês de *Real Time Driver Model* ou modelo de driver em tempo real, em português). Ela promove uma interface entre o *co-kernel* e o *driver* do dispositivo, como um mediador entre a aplicação e os serviços oferecidos pelo *driver* do dispositivo (YAGHMOUR, 2009, p. 330–331).

Figura 33: Migração de uma tarefa entre domínios



Fonte: Próprio. autor.

5.4 Estrutura do *software* proposto

O *software* executado no computador de bordo pode ser estruturado em forma de tarefas de tempo real, na qual cada tarefa se comporta como um processo descrito anteriormente. Entretanto, quem gerencia estas tarefas é o Xenomai e não o Linux padrão. O *software*, então, pode ser divido nas seguintes tarefas:

Controlador: nesta tarefa são executados os cálculos da estratégia de controle envolvida. Ao fim, as ações de controle são transmitidas através de uma porta serial para o módulo de atuação. Esta tarefa deve ser periódica, de tal modo que o período de amostragem seja compatível com o sistema dinâmico do helicóptero.

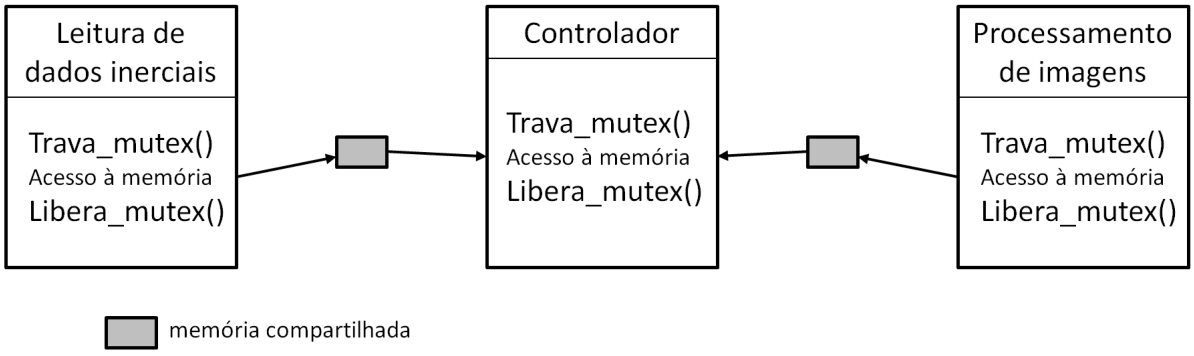
Processamento de imagens: as imagens obtidas através de uma câmera USB devem ser processadas a fim de determinar o fluxo óptico. Nesta tarefa, um único valor de velocidade deve ser associado aos eixos x e y, uma vez que tais velocidades devem representar o movimento relativo entre o helicóptero e o solo. Assim, faz-se o cálculo da média dos vetores obtidos de todos os pixels do quadro de imagem.

Leitura dos dados da odometria: as informações do módulo de instrumentação, que são a orientação, velocidades lineares e a altitude são lidas de uma porta serial e podem passar por um novo processo de filtragem e mudança de sistema referência.

As tarefas relacionadas ao sensoriamento (processamento de imagem e leitura dos dados inerciais) devem enviar os seus dados para a tarefa do controlador de modo a formar o *feedback* (realimentação, em português) da malha de controle. Determinou-se como forma de comunicação entre tais tarefas o uso de memória compartilhada, pois é de acesso rápido e não necessita da intervenção do *kernel*, no caso das *pipes* e filas.

Como descrito anteriormente, a memória compartilhada é uma área de memória que faz parte do espaço de endereçamento das tarefas associadas. Portanto, pode-se armazenar valores nestas áreas, de tal modo que as tarefas de sensoriamento escrevem seus dados na memória compartilhada e a tarefa de controle faz a leitura. Cada tarefa de sensoriamento terá uma memória compartilhada com a tarefa de controle. A Figura 34 ilustra a comunicação entre as tarefas e a ação dos *mutexes*.

Figura 34: Comunicação entre as tarefas do computador de bordo



Fonte: Próprio autor.

Outro recurso interessante do Xenomai é possibilidade de escolher quais núcleos executarão uma determinada tarefa, de modo semelhante à um sistema de multiprocessadores. Assim, decidiu-se que a tarefa de processamento de imagens será executada por dois núcleos, enquanto que as outras tarefas serão executadas pelos outros dois núcleos restantes.

O trecho de código na qual se faz o acesso à memória compartilhada deve ser protegido, de modo que, uma tarefa não escreva na área de memória ao mesmo tempo que outra tarefa faz a leitura de tal área, isto é, deve-se evitar o acesso simultâneo de tarefas. Para tal finalidade, usam-se *mutexes* que bloqueiam uma tarefa enquanto a outra não liberar o *mutex*. Assim, enquanto uma tarefa escreve na memória compartilhada, a tarefa que faz a leitura será bloqueada se a primeira não liberar o *mutex*. Isto é, a tarefa que entra no

trecho de código trava o *mutex* e o libera assim que sair do trecho.

Desta forma, se tem um sistema de multiprogramação em tempo real com comunicação entre as tarefas. Pode-se incluir mais tarefas à medida que for necessário, como a comunicação com uma estação em terra.

6 RESULTADOS EXPERIMENTAIS

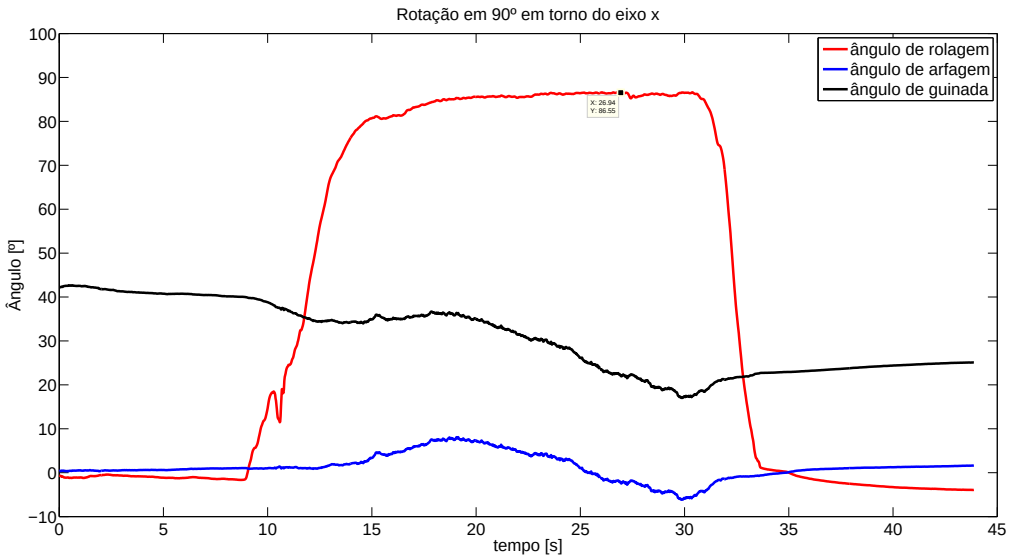
Neste capítulo serão apresentados resultados experimentais que avaliem o desempenho de cada componente do sistema de navegação. Foram realizados experimentos com os filtros de Kalman para orientação e velocidades lineares, fluxo óptico e com o computador de bordo. Além disso, cada experimento apresenta uma discussão a cerca do resultado obtido.

6.1 Resultados do filtro de Kalman estendido para orientação

Para verificar o desempenho do filtro de Kalman estendido que estima a orientação no espaço, buscou-se realizar rotações nos três eixos coordenados em valores fáceis de aferir. Assim, a placa de desenvolvimento STM32F3Discovery foi rotacionada em ângulos de 90º manualmente. A leitura dos dados ocorre a cada $20ms$ (frequência de $50Hz$).

A Figura 35 mostra um rotação de 90º em torno do eixo x da placa com o ângulo de guinada próximo de 42º e ângulo de arfagem em 0º. A rotação inicia-se próximo dos 9 segundos e a rotação foi realizada gradualmente, uma vez que foi feita de forma manual.

Figura 35: Rotação de 90º em torno do eixo x



Fonte: Próprio autor.

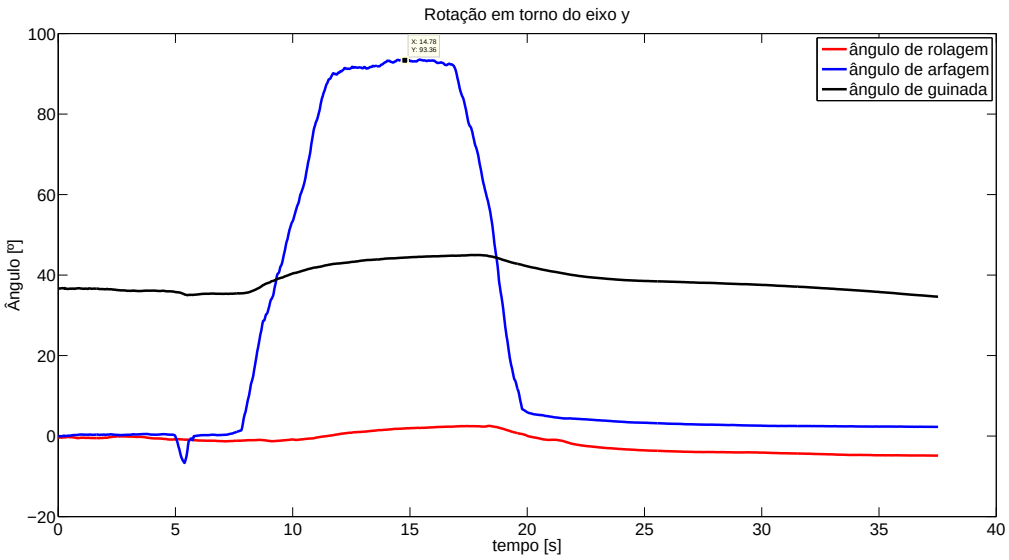
A partir da Figura 35, pode-se observar que o filtro possui certa precisão, de tal modo que é suficiente para o projeto em questão. Isto é, o erro é considerado pequeno, em torno de 5º.

Observa-se ainda uma variação nos outros dois ângulos. O ângulo de arfagem varia em torno de 6º enquanto o de guinada varia 25º, o que é um valor alto e deve ser considerado. Este comportamento pode ser verificado pela conversão do quatérnio em ângulos de Euler, uma vez que os três ângulos são definidos apenas por um quatérnio. Através dos ajustes das matrizes do filtro de Kalman estendido, pode-se obter um melhor resultado.

Além disso, o uso da visão computacional auxiliaria justamente na determinação do ângulo de guinada através de uma fusão sensorial entre o fluxo óptico e os dados inerciais. Entretanto, o projeto do algoritmo de fusão sensorial está fora do escopo deste projeto, sendo esta uma sugestão para trabalhos futuros.

A Figura 36 mostra a rotação de 90º em torno do eixo y da placa e que se inicia aos 8 segundos. Novamente, por se tratar de um movimento manual, este foi realizado de forma gradual.

Figura 36: Rotação de 90º em torno do eixo y

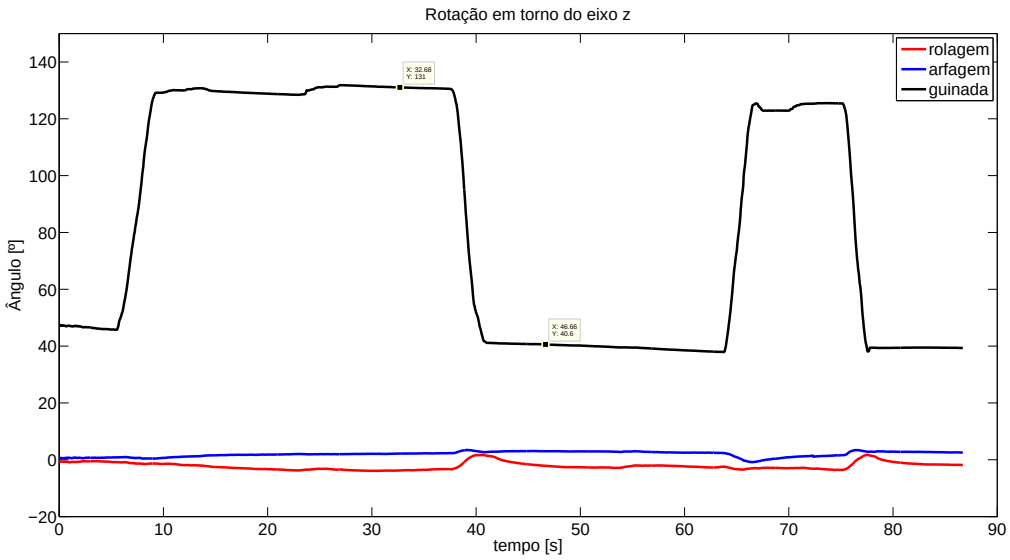


Fonte: Próprio autor.

Diferente do caso anterior, na Figura 36, os ângulos de arfagem e guinada sofrem variações bem mais suaves, de 8º para o ângulo de guinada e 3º para o ângulo de rolagem. Mais uma vez, observa-se um erro próximo de 5 em relação à rotação, também considerado pequeno.

A Figura 37 mostra duas rotações no eixo z da placa, a primeira iniciando aos 6 segundos e a segunda aos 64 segundos, partindo do ângulo próximo de 40º até o ângulo de 130º. Da mesma forma que nos casos anteriores, a rotação foi realizada de forma gradual.

Figura 37: Duas rotações de 90º em torno do eixo z



Fonte: Próprio autor.

Observa-se pequena variação nos ângulos de arfagem e rolagem. A precisão obtida é considerada suficiente para o projeto, uma vez que o erro é em torno de 3º.

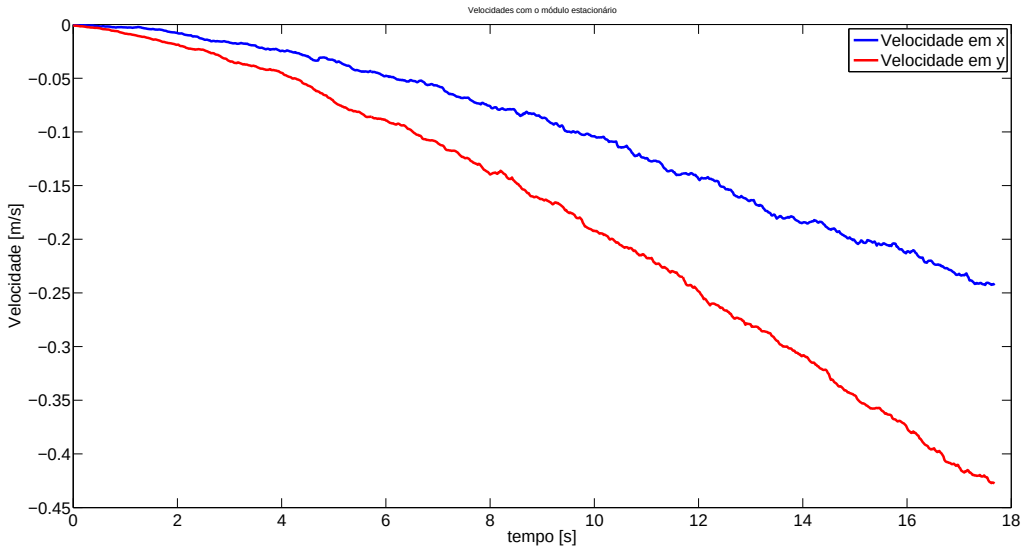
Por fim, uma das causas da presença de erro pode ser pelo fato de se realizar a rotação da placa de forma manual. Além disso, se pode ajustar as matrizes de covariância no filtro de Kalman estendido para obter resultados mais precisos caso necessário.

6.2 Resultados do filtro de Kalman para velocidades lineares

Neste experimento, faz-se a leitura das velocidades lineares da placa de desenvolvimento STM32F3Discovery a fim de verificar o desempenho do filtro de Kalman utilizado para realizar as estimativas das variáveis em questão.

Foram realizados testes em duas condições, uma com a placa estacionária e outra com a placa em um movimento oscilatório manual. A Figura 38 mostra o resultado obtido com a placa estacionária.

Figura 38: Velocidades com o módulo estacionário



Fonte: Próprio autor.

Da Figura 38 pode-se observar que o filtro não é capaz de filtrar totalmente os ruídos presentes. Assim, pode-se verificar a presença de um valor de *bias*, isto é, a diferença entre o valor real e o esperado de uma estimativa. Entretanto, o filtro de Kalman parte da hipótese que os ruídos associados ao sistema possuem valor médio nulo, o que não é o caso.

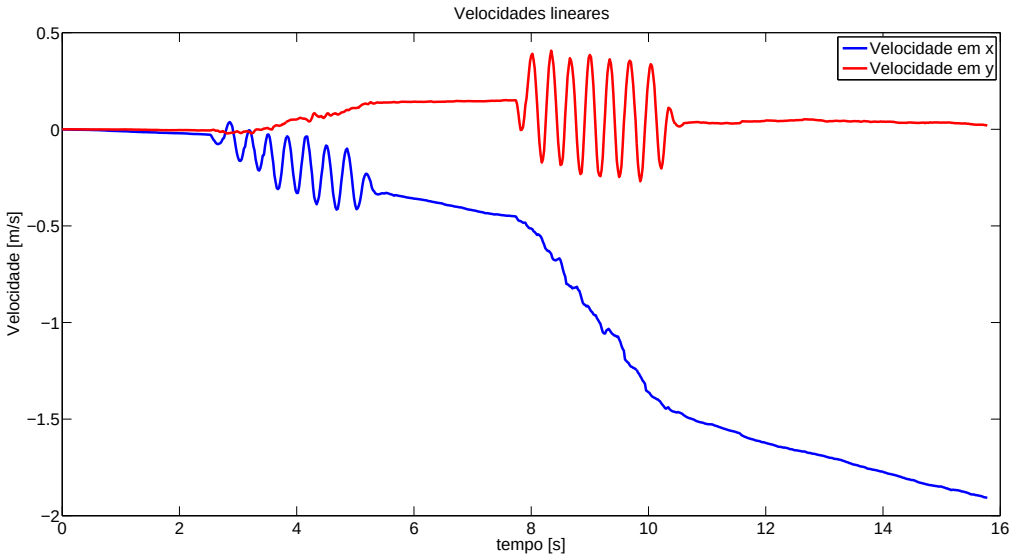
A presença deste valor de *bias* provoca um efeito de integração infinita. Uma vez que a placa está parada, mas os valores de aceleração não são nulos, a velocidade (integral da aceleração) não é nula e seu valor aumenta em módulo ao longo do tempo.

Para buscar um desempenho satisfatório do algoritmo, deve-se realizar os processos de calibração e compensação de erro do sensor. A própria fabricante do acelerômetro (ST-Microelectronics) fornece um método de calibração do acelerômetro, apesar do sensor ser calibrado ainda em fábrica. Assim, através da nota de aplicação AN3192 (título *Using LSM303DLH for a tilt compensated electronic compass*), a STMicroelecronics descreve um método de calibração para o acelerômetro e para o magnetômetro. Entretanto, apesar de se executar a calibração, os resultados não demostraram diferenças.

A Figura 39 mostra os resultados obtidos com a placa oscilando manualmente, em um

teste no eixo x e em outro no eixo y.

Figura 39: Velocidades com o módulo em oscilação



Fonte: Próprio autor.

Pela Figura 39 verifica-se que o filtro permite detectar movimentos, uma vez que foi possível observar o perfil de velocidade de um movimento oscilatório teórico.

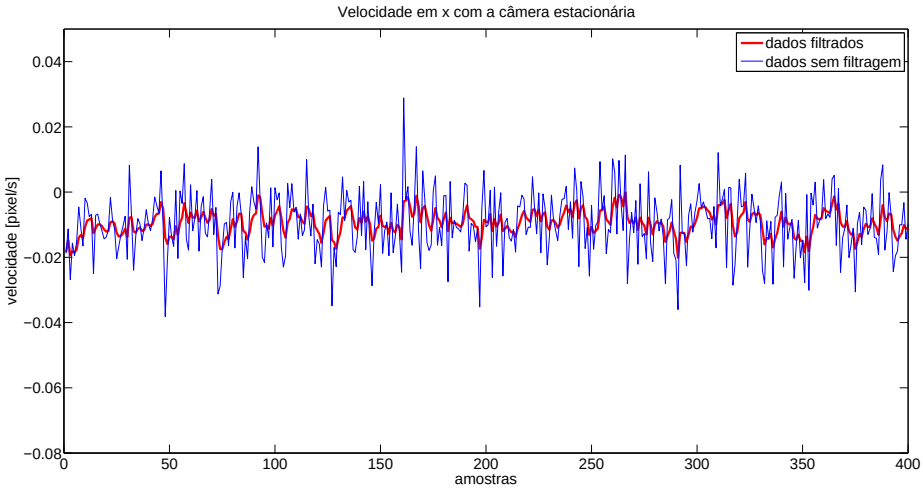
Mais uma vez, pode-se observar a presença do *bias*, provocando novamente a integração da aceleração mesmo no estado estacionário. Ademais, o valor do *bias* no eixo x é maior que no eixo y em módulo, por isso, o valor resultante da integração cresce (em módulo) mais rapidamente.

Uma vez que o desempenho do filtro não foi satisfatório, se faz necessário desenvolver um método de calibração mais robusto, além de buscar melhorar o modelo do sistema do filtro de Kalman, assim como o ajuste dos valores das matrizes de covariância. Além disso, pode-se desenvolver um método de fusão sensorial com as informações do sistema de visão computacional para obter dados mais confiáveis.

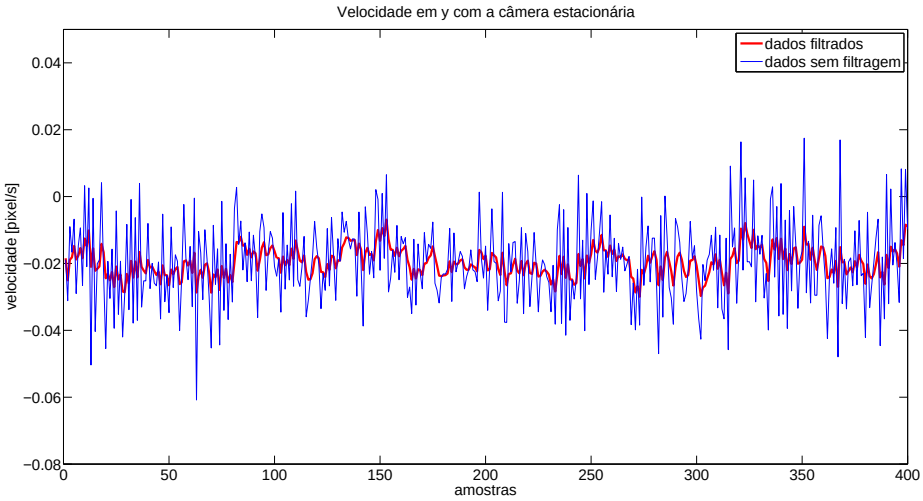
6.3 Resultados do fluxo óptico

Para testar o desempenho do algoritmo de fluxo óptico, utilizou-se um plano quadriculado preto e branco, como um tabuleiro de xadrez por exemplo. Primeiro, manteve-se o plano estacionário em relação à câmera a fim de verificar a presença de ruídos. Segundo, foram realizados movimentos horizontais e verticais oscilatórios a fim de verificar a detecção de movimento. A figura 40 mostra os resultados obtidos com o plano estacionário em relação à câmera.

Figura 40: Velocidades com o plano estacionário



(a) Velocidade em x



(b) Velocidade em y

Fonte: Próprio autor.

Observa-se da Figura 40 que há a presença de ruídos no conjunto de dados. Além disso, existe um valor de *offset* nas velocidades dos dois eixos coordenados. Para reduzir o efeito do ruído, pode-se utilizar um filtro de Kalman baseado no conjunto de equações

$$\mathbf{F} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6.1}$$

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6.2}$$

$$\mathbf{P} = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix} \tag{6.3}$$

$$\mathbf{Q} = \begin{bmatrix} 0.2 & 0 \\ 0 & 0.2 \end{bmatrix} \tag{6.4}$$

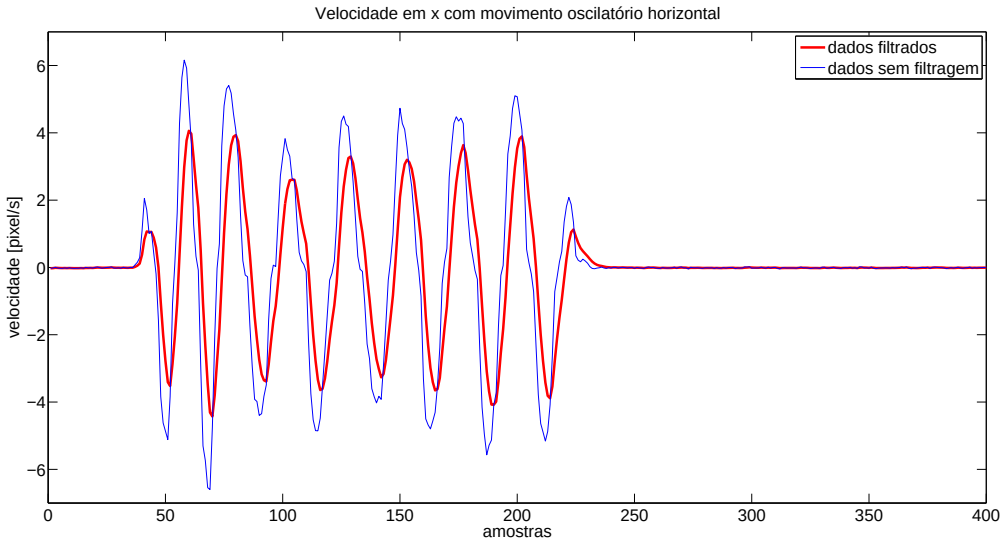
$$\mathbf{R} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}. \tag{6.5}$$

Na própria Figura 40 estão os dados já filtrados utilizando o filtro de Kalman. Observa-se que há uma redução considerável do ruído. Entretanto, ainda se deve remover o valor de *offset*.

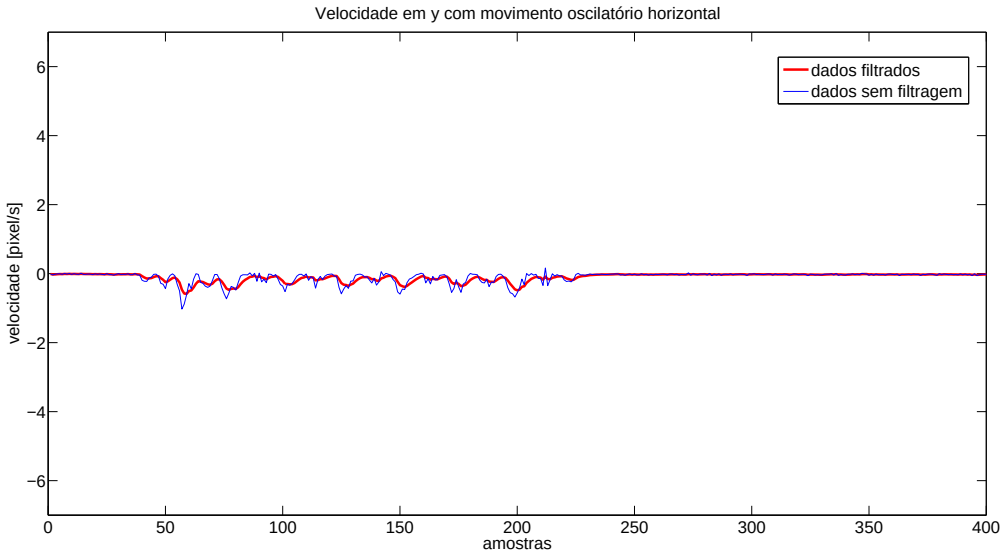
O segundo experimento se baseou em produzir movimentos oscilatórios horizontais e verticais no plano de forma a verificar a detecção de movimento por parte do algoritmo de fluxo óptico. Deve-se destacar que neste experimento não busca-se verificar se os valores obtidos de velocidades condizem com os valores reais do movimento, uma vez que é necessário realizar a calibração da câmera e realizar movimentos baseados em uma velocidade conhecida. O interesse é somente em observar se o algoritmo detecta os movimentos realizados.

A Figura 41 mostra os resultados obtidos com a oscilação horizontal também com o filtro de Kalman.

Figura 41: Velocidades com o plano oscilando horizontalmente



(a) Velocidade em x

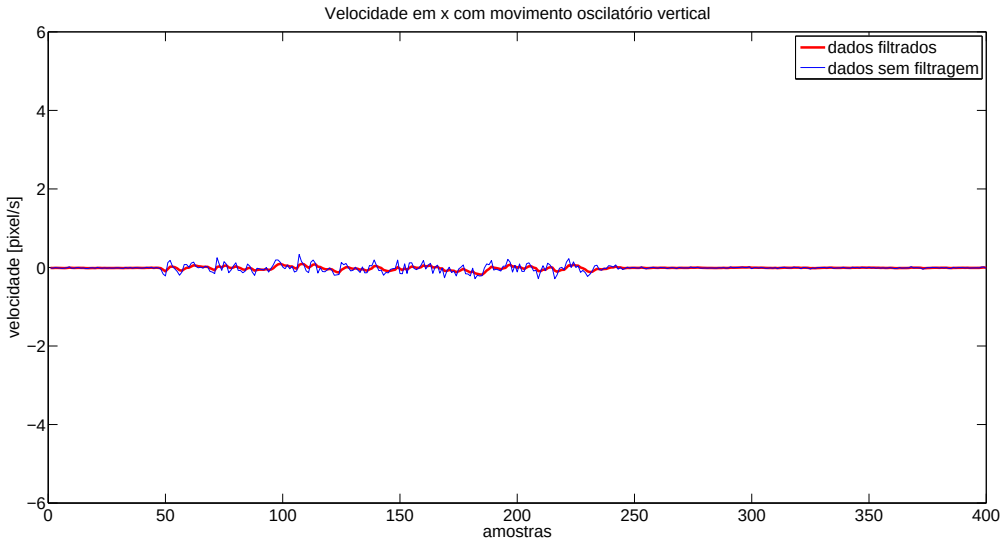


(b) Velocidade em y

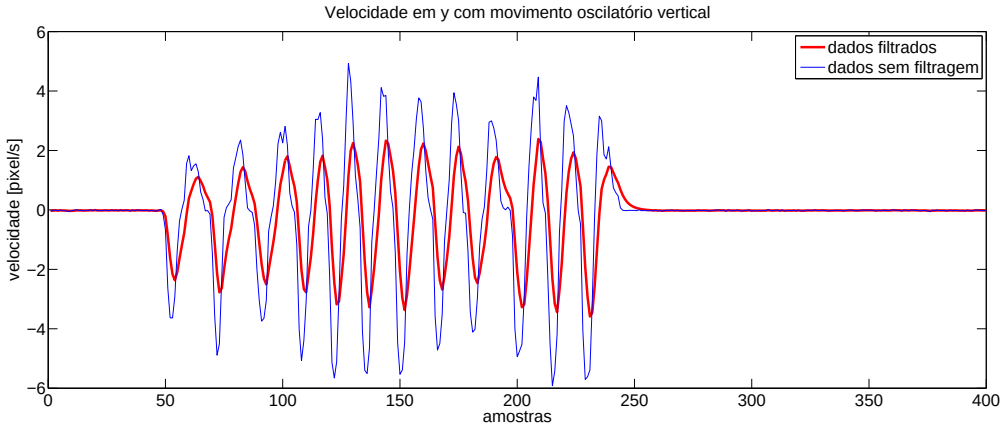
Fonte: Próprio autor.

Da Figura 41 observa-se que o algoritmo tem a capacidade de detectar o movimento realizado. Verifica-se também que ocorrem pequenas oscilações no eixo em que não há movimento. Do mesmo modo, a Figura 42 apresenta os resultados obtidos com a oscilação vertical também com o filtro de Kalman.

Figura 42: Velocidades com o plano oscilando verticalmente



(a) Velocidade em x



(b) Velocidade em y

Fonte: Próprio autor.

Para analisar melhor o desempenho do fluxo óptico, pode-se calibrar a câmera, o que permite inferir as velocidades em metros por segundos. Além disso, pode-se realizar a fusão sensorial com os dados inerciais de forma a obter informações mais precisas e confiáveis.

6.4 Resultados do computador embarcado

O teste para verificação do desempenho do computador de bordo consiste em analisar a capacidade de cumprir as restrições de tempo real das tarefas. Como apresentado no

Capítulo 5, as tarefas são: leitura dos dados de odometria, execução dos cálculos do controlador e o processamento de imagens para obtenção do fluxo óptico.

As tarefas de tempo real foram todas executadas de uma vez, simulando o computador de bordo em operação. Os valores de tempo de duração da execução de cada *loop* das tarefas foram armazenadas em arquivos. Os resultados foram utilizados para gerar histogramas com os valores obtidos.

A tarefa de controle se baseia nos trabalhos de Brandão (2013) e Santana (2011). Consiste no projeto de um controlador de altitude, dado por

$$f_3 = m\left[\ddot{z}_d + K_{dz1}tanh(K_{dz2}\dot{\tilde{z}}) + K_{pz1}tanh(K_{pz2}\tilde{z}) + g\right] \tag{6.6}$$

onde f_3 é a força de propulsão do helicóptero, z_d é a referência de altitude, m a massa do veículo e $\tilde{z} = z_d - z$. K_{dz1}, K_{dz2}, K_{pz1} e K_{pz2} são os ganhos do controlador. O canal do rádio controle que altera a velocidade do rotor principal e, consequentemente, provoca a mudança de altitude é o canal 1, que é dado por

$$CH1 = K_{dz1}tanh(K_{dz2}\dot{\tilde{z}}) + K_{pz1}tanh(K_{pz2}\tilde{z}) + CH1_{min} \tag{6.7}$$

onde $CH1_{min}$ é o valor do período do sinal PWM que mantém o helicóptero com a altitude constante.

O controlador do ângulo de guinada é dado pela Equação 6.8.

$$f_4 = \frac{I_{zz}}{l_t}\left[\ddot{\psi}_d + K_{d\psi1}tanh(K_{d\psi2}\dot{\tilde{\psi}}) + K_{p\psi1}tanh(K_{p\psi2}\tilde{\psi})\right] \tag{6.8}$$

onde f_4 é a força de propulsão do rotor de cauda, l_t é a distância da cauda para o centro de massa do helimodelo, ψ_d é a referência do ângulo, I_{zz} momento de inércia do veículo no eixo z e $\tilde{\psi} = \psi_d - \psi$. $K_{d\psi1}$, $K_{d\psi2}$, $K_{p\psi1}$ e $K_{p\psi2}$ são os ganhos do controlador. O canal do rádio controle que altera o coletivo do rotor principal e, consequentemente, provoca movimento de guinada é o canal 4, que é dado por

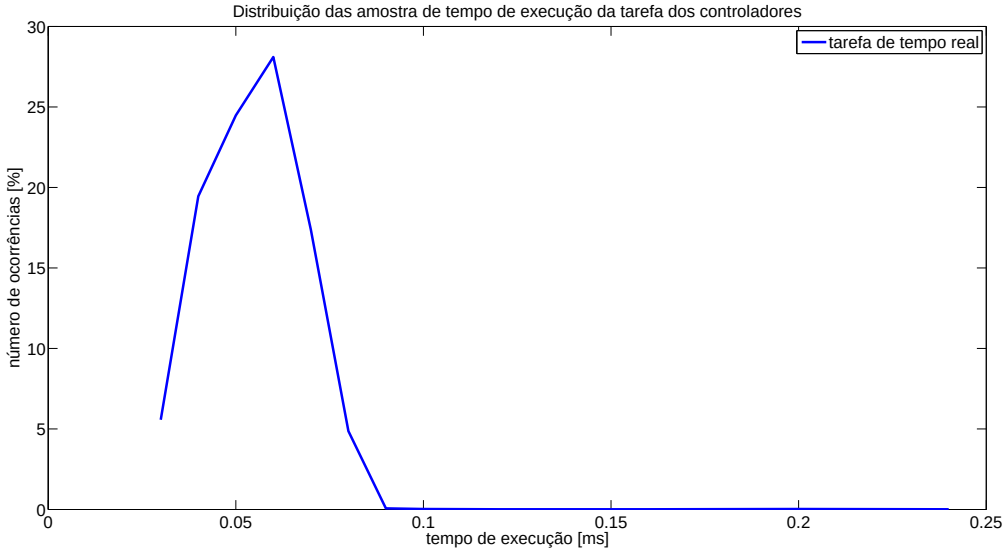
$$CH4 = K_{d\psi1}tanh(K_{d\psi2}\dot{\tilde{\psi}}) + K_{p\psi1}tanh(K_{p\psi2}\tilde{\psi}) + CH4_{min} \tag{6.9}$$

onde $CH4_{min}$ é o valor do período do sinal PWM que mantém o helicóptero sem rotacionar no eixo z.

A Figura 43 mostra o histograma gerado a partir dos resultados obtidos da execução da

tarefa do controlador.

Figura 43: Distribuição das amostras de tempo de execução do controlador



Fonte: Próprio autor.

A Tabela 1 apresenta as medidas estatísticas da execução da tarefa do controlador.

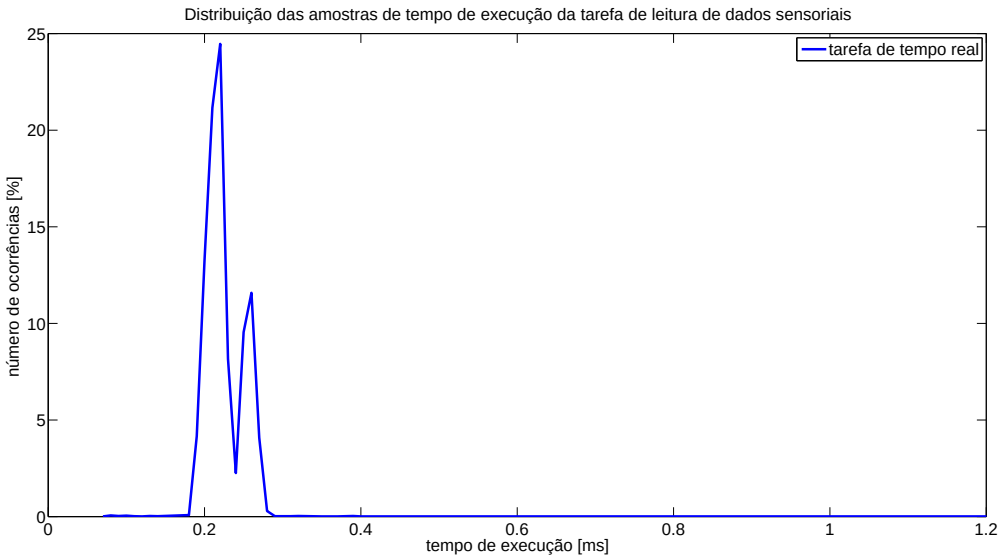
Tabela 1: Medidas estatísticas obtidas da tarefa do controlador

Medida estatística	Valor [ms]
média	0,0548
desvio padrão	0,0132
moda	0,06
mínimo	0,03
máximo	0,24

Observa-se que o valor médio é bem mais próximo do valor mínimo que do máximo. Além disso, o valor médio é próximo também da moda do conjunto. Verifica-se que o gráfico é bem centrado, apresentando pouca dispersão.

A Figura 44 mostra o histograma gerado a partir dos resultados obtidos da execução da tarefa de leitura dos dados sensoriais.

Figura 44: Distribuição das amostras de tempo de execução da leitura dos dados sensoriais



Fonte: Próprio autor.

A Tabela 2 apresenta as estatística descritiva da execução da tarefa de leitura da odometria. De forma similar ao caso da tarefa de controle, o gráfico é bem centrado, tendo as

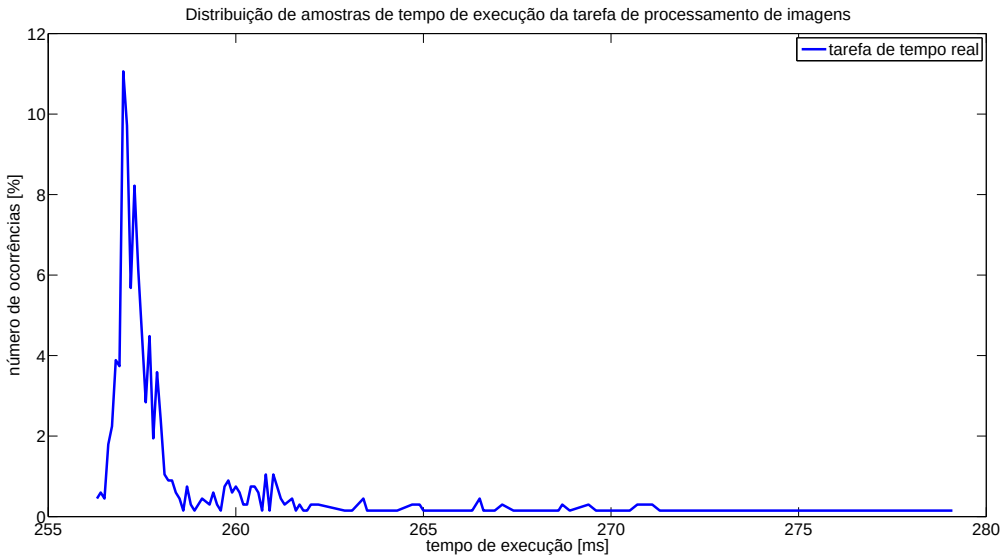
Tabela 2: Medidas estatísticas obtidas da tarefa de leitura da odometria

Medida estatística	Valor [ms]
média	0,2665
desvio padrão	0,6678
moda	0,22
mínimo	0,07
máximo	19,71

suas medidas bem próximas umas das outras. Entretanto, observa-se um valor máximo grande que pode ter sido gerado pois a tarefa faz a migração de domínios, do Xenomai para o Linux padrão. Isto ocorre pois se faz chamadas de sistema através de funções do Linux para acessar a porta serial.

Por último, a Figura 45 mostra o histograma gerado a partir dos resultados obtidos da execução da tarefa de processamento de imagens.

Figura 45: Distribuição das amostras de tempo de execução do processamento de imagens



Fonte: Próprio autor.

A Tabela 1 apresenta as medidas estatísticas da execução da tarefa de processamento de imagens.

Tabela 3: Medidas estatísticas obtidas da tarefa de processamento de imagens

Medida estatística	Valor [ms]
média	258,4952
desvio padrão	2,9365
moda	257
mínimo	256,3
máximo	279,1

Neste último caso, por ser uma tarefa que exige mais recursos computacionais que as anteriores, o desempenho do sistema de tempo real se torna mais evidente. Observa-se o baixo valor de desvio padrão comparado com o valor médio, o que mostra um histograma bem centrado, isto é, pouco disperso. Além disso, verifica-se que o valor médio está bem próximo dos valores mínimo e de moda. Também verifica-se que o valor máximo não é muito maior que o valor médio calculado.

Através deste simples experimento, pode-se observar que o sistema de tempo real busca

manter constante o tempo de execução das tarefas de tempo real, obtendo medidas estatísticas satisfatórias. O Apêndice B contém o código fonte referente à este experimento.

7 Conclusões e trabalhos futuros

Neste trabalho foi proposto um sistema de navegação autônoma para um helicóptero miniatura. O projeto é constituído por um módulo microcontrolado para leitura e filtragem de dados de sensores inerciais, um sistema de visão computacional com a finalidade de detectar movimentos lineares do helicóptero e um computador de bordo executando um sistema operacional baseado em Linux com capacidades de tempo real.

O módulo de sensoriamento realiza a leitura dos sensores de uma unidade de medição inercial contendo um acelerômetro, giroscópio e magnetômetro. Os dados obtidos são filtrados através de um filtro de Kalman estendido utilizando-se uma representação em quatérnios. Através de simples experimentos, pôde-se comprovar a precisão e eficiência do filtro em estimar a orientação de um objeto no espaço tridimensional.

Além da estimativa da orientação, o módulo de sensoriamento executa um filtro de Kalman para estimar as velocidades lineares do veículo. Pelos testes realizados, verificou-se que é necessário aprimorar o método de filtragem, uma vez que não foi possível atenuar um ruído com valor médio diferente de zero.

No sistema de visão computacional, o método para a detecção de movimento é o algoritmo de Lucas e Kanade (1981) e se baseia no fluxo óptico obtido através de um cálculo de gradiente de intensidade de brilho dos pixels. As respostas dos experimentos também se mostraram com ruído e um valor de *offset* pequeno, que puderam ser minimizados com um filtro de Kalman. Verificou-se ainda que o algoritmo é capaz de detectar movimentos nos dois eixos coordenados.

O computador de bordo é uma placa simples com um microprocessador ARM contendo quatro núcleos que operam em $1,0GHz$. O computador executa uma distribuição Linux

contendo o suporte de tempo real Xenomai. Através de experimentos simples, observou-se a capacidade do Xenomai em manter o tempo de execução das tarefas o mais constante possível. Assim, os valores médio, de moda e mínimo da distribuição de amostras de tempo de execução são bem próximas, o que pode ser confirmado através do baixo valor de desvio padrão.

Como trabalhos futuros, pode-se buscar tornar o módulo de sensoriamento mais robusto e preciso, através da utilização de métodos de filtragem com melhores desempenhos para estimar as velocidades e da melhoria do filtro de Kalman estendido proposto para a orientação. Deve-se testar o módulo embarcado no helicóptero, pois assim estará sujeito aos distúrbios produzidos pelo veículo, como vibração mecânica e a geração de campo magnético pelo motor principal.

Deve-se também aprimorar o algoritmo de visão computacional para que possa não só detectar o movimento do veículo, mas também determinar os valores de velocidade em metros por segundo, o que pode ser obtido através da calibração da câmera e da medida de altitude proveniente do sensor de ultrassom. Além disso, o algoritmo poderá estimar não só deslocamentos lineares, mas também inferir sobre movimentos de rotação e estimar o ângulo da rotação.

Pode-se também desenvolver um algoritmo de fusão sensorial envolvendo tanto os dados inerciais quanto as informações do fluxo óptico, de modo a obter resultados mais confiáveis. Uma vez que o filtro para estimar as velocidades não teve um desempenho satisfatório, a fusão de dados do acelerômetro e do sistema de visão computacional poderia gerar dados mais precisos sobre as velocidades do veículo.

O computador embarcado poderá conter mais tarefas como comunicação com uma estação em terra, para a qual serão enviados os dados de orientação e posição, além de enviar os quadros de imagem da câmera. Assim, um modelo em simulação na estação em tera poderá ser executado com os dados recebidos, mostrando assim a trajetória do helicóptero.

Por último, com o sistema de navegação realizando mais testes e verificando-se a sua robustez, poderá então ser embarcado no helicóptero e realizar os experimentos de navegação autônoma.

Este trabalho foi importante, pois é um primeiro passo no desenvolvimento de um sistema embarcado de tempo real para a navegação autônoma de um helimodelo. Através deste sistema, os controladores propostos pelo grupo de pesquisa em *Controle não linear aplicado a VANTs* do Departamento de Engenharia Elétrica da UFES poderão futuramente ser testados nesta plataforma.

Referências

ANDERSON, D. F.; EBERHARDT, S. **Understanding flight**. 2nd. ed. USA: McGraw-Hill, 2010.

ARM. **Documentação da biblioteca de processamento digital de sinais da ARM**. Disponível em: <http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html>. Acesso em: 02 de Agosto de 2014.

BOVIK, A. **Handbook of image and video processing**. 2nd. ed. Burlington, MA, USA: Elsevier Academic Press, 2009.

BRADSKI, G. R.; KAEHLER, A. **Learning OpenCV - computer vision with OpenCV library**. Sebastopol, CA. USA: O'Reilly Media, 2008.

BRANDÃO, A. **Projeto de Controladores Não Lineares para Voo Autônomo de Veículos Aéreos de Pás Rotativas**. Tese (Doutorado em Engenharia Elétrica) — Programa de Pós Graduação em Engenharia Elétrica, Universidade Federal do Espírito Santo, Vitória, ES, Abril 2013.

BUTTAZZO, G. C. **Hard real-time computing systems: predictable scheduling algorithms and applications**. 3rd. ed. New York, USA: Elsevier Academic Press, 2011.

CAI, G.; CHEN, B. M.; LEE, T. H. **Unmanned Rotorcraft Systems**. London, UK: Springer, 2011.

CALDEIRA, E. M. de O. **Navegação reativa de robôs móveis com base no fluxo óptico**. Tese (Doutorado em Engenharia Elétrica) — Programa de Pós Graduação em Engenharia Elétrica, Universidade Federal do Espírito Santo, Vitória, ES, Dezembro 2002.

CASTILLO, P.; LOZANO, R.; DZUL, A. **Modelling and control of mini-flying machines**. London: Springer, 2005.

DAVIES, E. R. **Computer and machine vision: theory, algorithms, praticalities**. 4th. ed. USA: Elsevier Academic Press, 2012.

DEVICES, B. **Single Board Computer SABRE Lite board**. Disponível em: <http://boundarydevices.com/products/sabre-lite-imx6-sbc/>. Acesso em: 09 de Agosto de 2013.

FARAGHER, R. Understanding the basis of the kalman filter via a simple and intuitive derivation [lecture notes]. **Signal Processing Magazine, IEEE**, v. 29, n. 5, p. 128–132, Sept 2012. ISSN 1053-5888.

GERUM, P. **Xenomai – Implementing a RTOS emulation framework on GNU/Linux**. Abril 2004. Disponível em: <http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf>. Acesso em: 11 de Janeiro de 2014.

GERUM, P. **Life with Adeos**. Outubro 2005. Disponível em: <http://www.xenomai.org/documentation/xenomai-2.3/pdf/Life-with-Adeos-rev-B.pdf>. Acesso em: 11 de Janeiro de 2014.

GERUM, P. **Native API Tour**. Março 2006. Disponível em: <http://www.xenomai.org/documentation/xenomai-2.3/pdf/Native-API-Tour-rev-C.pdf>. Acesso em: 11 de Janeiro de 2014.

GONZALEZ, R. C.; WOODS, R. E.; EDDINGS, S. L. **Digital image processing using MATLAB**. 2nd. ed. USA: Gatesmark Publishing, 2009.

GREWAL, M. S.; WEILL, L. R.; ANDREWS, A. P. **Global positioning systems, inertial navigation and integration**. 2nd. ed. New Jersey, USA: John Wiley & Sons, 2007.

GROVES, P. D. **Principles of GNSS, inertial and multisensor integrated navigation systems**. USA: Artech House, 2008.

HALLINAN, C. **Embedded Linux primer: a practical real-world approach**. 2nd. ed. USA: Prentice Hall, 2011.

KUIPERS, J. B. **Quaternions and rotations sequences: a primer with applications to orbits, aerospace and virtual reality.** [S.l.]: Princeton University Press, 1998.

LAPLANTE, P. A. **Real-time systems design and analysis**. 3rd. ed. [S.l.]: IEEE Press, 2004.

LI, M.; MOURIKIS, A. 3-d motion estimation and online temporal calibration for camera-imu systems. In: **Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2013**. Karlsruhe, Germany: [s.n.], 2013. p. 5709–5716.

LUCAS, B.; KANADE, T. An iterative image registration technique with an application to stereo vision. In: **Proceedings of the 7th International Joint Conference on Artificial Intelligence - IJCAI 81**. Vancouver, Canada: [s.n.], 1981. p. 674–679.

MADGWICK, S. O. H.; VAIDYANATHAN, R.; HARRISON, A. **An Efficient Orientation Filter for IMU and MARG Sensor Arrays**. Department of Mechanical Engineering, University of Bristol: [s.n.], 2010.

MALLOT, H. A. **Computational vision: information processing in perception and visual behavior**. USA: The MIT Press, 2000.

MOHAMMAD-ZAHERI, M.; CHEN, L. Design of an intelligent controller for a model helicopter using neuro-predictive method with fuzzy compensation. In: **Proceedings of the World Congress on Engineering, WCE 2007**. London, UK: [s.n.], 2007. p. 19–24.

MOREIRA, M. A. G.; FREITAS, E. J. R.; PEREIRA, G. A. S.; TORRES, L. A. B.; ISCOLD, P. Modelagem e controle de um helimodelo em ambientes internos. In: **Anais do XVIII Congresso Brasileiro de Automática - CBA2010**. Bonito - MS, Brasil: [s.n.], 2010. p. 3218–3224.

NOURELDIN, A.; KARAMAT, T. B.; GEORGY, J. **Fundamentals of inertial navigation, satellite-based positioning and their integration**. Berlin, Germany: Springer, 2013.

PADFIELD, G. D. **Helicopter flight dynamics: the theory and applicaion of flying qualities and simulation modeling**. 2nd. ed. Oxford, UK: Blackwell Publising, 2007.

PHANG, S. K.; ONG, J. J.; YEO, R. T. C.; CHEN, B. M.; LEE, T. H. Autonomous mini-uav for indoor flight with embedded on-board vision processing as navigation system. In: **Proceedings of the IEEE Region 8 International Conference on Computational Technologies in Electrical and Electronics Engineering (SIBIRCON), 2010**. Irkutsk Listvyanka, Russia: [s.n.], 2010. p. 722–727.

PIZETTA, I. H. B. **Uma plataforma para suporte à navegação autônoma de veículos aéreos da pás rotativas**. Dissertação (Mestrado em Engenharia Elétrica) — Programa de Pós Graduação em Engenharia Elétrica, Universidade Federal do Espírito Santo, Vitória, ES, Abril 2013.

SANTANA, L. V. **Controle de altitude e ângulo de guinada de um mini-helicóptero**. Dissertação (Mestrado em Engenharia Elétrica) — Programa de Pós Graduação em Engenharia Elétrica, Universidade Federal do Espírito Santo, Vitória, ES, Março 2011.

SANTANA, L. V.; BRANDÃO, A. S.; SARCINELLI-FILHO, M.; CARELLI, R. Uma estrutura sensorial e de comunicação para o controle de altitude e guinada de um mini-helicóptero autônomo. In: **Anais do XVIII Congresso Brasileiro de Automática - CBA2010**. Bonito - MS, Brasil: [s.n.], 2010. p. 1840–1845.

SANTANA, L. V.; BRANDÃO, A. S.; SARCINELLI-FILHO, M.; CARELLI, R. Hovering control of a miniature helicopter attached to a platform. In: **Proceedings of the 20th IEEE International Symposium on Industrial Electronics - ISIE2011**. Gdansk, Poland: [s.n.], 2011. p. 2231–2236.

SANTANA, L. V.; SARCINELLI-FILHO, M.; CARELLI, R. Estimation and control of the 3d position of a quadrotor in indoor environments. In: **Proceedings of the 16th International Conference on Advanced Robotics (ICAR) 2013**. Montevideo, Uruguai: [s.n.], 2013. p. 1–6.

SEDDON, J.; NEWMAN, S. **Basic helicopter aerodynamics**. 3rd. ed. UK: John Wiley & Sons, 2011.

SIEGWART, R.; NOURBAKHSH, I. R.; SCARAMUZZA, D. **Introduction to autonomous mobile robots**. 2nd. ed. USA: The MIT Press, 2011.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating system concepts**. 9th. ed. New Jersey, USA: John Wiley & Sons, 2013.

STALLINGS, W. **Operating systems: internals and design principles**. 7th. ed. New Jersey, USA: Prentice Hall, 2012.

STMICROELECTRONICS. **Placa de Desenvolvimento STM32F3Discovery**. Disponível em: <http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF254044>. Acesso em: 11 de Junho de 2013.

TANENBAUM, A. S. **Sistemas operacionais modernos**. 3. ed. São Paulo: Pearson Prentice Hall, 2009.

WATSON, M. **The design and implementation of a robus AHRS for integration into a quadrotor platform**. Department of Electronic and Electrical Engineering, University of Sheffield, UK: [s.n.], 2013. Master of Engineering Report.

WELCH, G.; BISHOP, G. **An introduction to the Kalman filter**. 2001. Curso apresentado no *Special Interest Group on Graphics and Interactive Techniques* 2001. Disponível em: http://www.cs.unc.edu/~tracker/media/pdf/SIGGRAPH2001_CoursePack_08.pdf. Acesso em: 02 de Março de 2014.

XENOMAI. **Xenomai: Real-Time Framework for Linux**. `http://www.xenomai.org`.

YAGHMOUR, K. **Construindo sistemas Linux embarcados**. 2. ed. Rio de janeiro: Alta Books, 2009.

APÊNDICE A – CÓDIGO FONTE DO MÓDULO DE INSTRUMENTAÇÃO

Neste apêndice serão apresentados os códigos fonte embarcado no módulo de instrumentação. O código referente à IMU e à USART foi retirado do programa de demonstração fornecido pela STMicroelectronics e parcialmente modificados para o uso no módulo.

A.0.1 main.c

```
/* Includes ------------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include "stm32f30x.h"
#include "stm32f3_discovery.h"
#include "imu.h"
#include "usart.h"
#include "ultrassom.h"
#include "ahrs_ekf.h"
#include "odometry_kf.h"

#define D2R             (float)  0.0174532925f
#define SampleTime (float) 0.02
/******************* Functions Prototypes *******************/
void TIM3_Init(void);
void TIM3_IRQHandler(void);
void USART_PutF(float fdata);
/************************************************************/


/****************** STM32F3 Library Structs *******************/
static RCC_ClocksTypeDef RCC_Clocks;
//static GPIO_InitTypeDef GPIO_InitStructure;
/************************************************************/


/*********************** User variables ***********************/
float MagBuffer[3]={0,0,0}, AccBuffer[3]={0,0,0}, GyrBuffer[3]={0,0,0};
char SerialBuffer[200];
uint8_t Wait;
float pitch=0, roll=0, yaw=0;
```

```
float Velx=0, Vely=0;
float timeval = SampleTime;
/*************************************************************/



/*****************************************************************************
*****************************************************************************
*****************************************************************************
                                                          MAIN
*****************************************************************************
*****************************************************************************
*****************************************************************************/
int main(void){

        SystemInit();

        /* SysTick end of count event each 1ms - divide by 1000 for 1 x ms*/
        RCC_GetClocksFreq(&RCC_Clocks);
        SysTick_Config(RCC_Clocks.HCLK_Frequency / 1000);

        /* Initialize LEDs and User Button available on STM32F3-Discovery board */
        STM_EVAL_LEDInit(LED3);
        STM_EVAL_LEDInit(LED4);
        STM_EVAL_LEDInit(LED5);
        STM_EVAL_LEDInit(LED6);
        STM_EVAL_LEDInit(LED7);
        STM_EVAL_LEDInit(LED8);
        STM_EVAL_LEDInit(LED9);
        STM_EVAL_LEDInit(LED10); //PE13

        /* Gyroscope configuration*/
        GyroConfig();
        /* Compass and Accelerometer configuration*/
        CompassConfig();

        /* Start USART2 - PIN A14 (TX) e PIN A15 (RX)*/
        USART2_Init(115200);

        Ultrassom_TIM_Init();
        //PD4 - Echo PD3 - Trigger
        Ultrassom_InitIO();
        AHRS_EKFInit();
        Odometry_KFInit();

        /* Set unbuffered mode for stdout (newlib) */
        setvbuf( stdout, 0, _IONBF, 0 );

        //TIM3_Init();

        /* Infinite loop */
        while(1){

                Wait = 0;
                //STM_EVAL_LEDOn(LED10);
                // Read Gyro Angular data
                GyroReadAngRate(GyrBuffer);
```

```
                // Read Compass data
                CompassReadMag(MagBuffer);
                CompassReadAcc(AccBuffer);

                UltrassomTrigger();

                AHRS_EKFUpdate(AccBuffer, GyrBuffer, MagBuffer, &timeval, &roll, &pitch, &yaw);
                Odometry_KFUpdate(AccBuffer, &timeval, &Velx, &Vely);

                //sprintf(SerialBuffer, "%f\t%f\t%f\t%f\t%f\t%f\r\n", roll, pitch, yaw, Altitude, Ve

                while(!Wait){
                }

        }

        return 0;
}




/*****************************************************************************
*****************************************************************************
*****************************************************************************
                                        INTERRUPT & OTHERS
*****************************************************************************
*****************************************************************************
*****************************************************************************/
/**
  * @brief  Interrupt routine for with 1 mSecond window
  */
void SysTick_Handler(void)
{
        static int i=0;
        i++;
        /* 1ms Interrupt */
        if(i % 20 == 0){
                USART_PutF(roll);
                USART_PutF(pitch);
                USART_PutF(yaw);
                USART_PutF(Altitude);
                USART_PutF(Velx);
                USART_PutF(Vely);

                //USART2_PutS(SerialBuffer);
                STM_EVAL_LEDToggle(LED10);
                Wait = 1;
        }


}


void USART_PutF(float fdata){

        char *ptr;
        ptr = (char*)(&fdata);
```

```
        USART2_PutC(ptr[0]);
        USART2_PutC(ptr[1]);
        USART2_PutC(ptr[2]);
        USART2_PutC(ptr[3]);

/*
        // Teste Linux
        USART2_PutC(ptr[3]);
        USART2_PutC(ptr[2]);
        USART2_PutC(ptr[1]);
        USART2_PutC(ptr[0]);
*/
}
```

A.0.2 ahrs_ekf.h

```
#include <stdio.h>
#include <math.h>
#include "stm32f30x.h"
#include "stm32f3_discovery.h"
#include "stm32f3_discovery_lsm303dlhc.h"
#include "stm32f3_discovery_l3gd20.h"

/* Biblioteca da ARM para operar com DSP */
#include "arm_math.h"


#define R2D                         (float) 57.2957795131f
#define D2R                         (float)  0.0174532925f

void AHRS_EKFInit();
void AHRS_EKFUpdate(float *Acc, float *Gyr, float *Mag, float *sampletime, float *Roll, float *Pitch
```

A.0.3 ahrs_ekf.c

```
#include "ahrs_ekf.h"

/* Biblioteca da ARM para operar com DSP:
        #include "arm_math.h"

        typedef struct
        {
                uint16_t numRows;       < number of rows of the matrix.
                uint16_t numCols;       < number of columns of the matrix.
                float32_t *pData;       < points to the data of the matrix.
        } arm_matrix_instance_f32;



        void arm_mat_init_f32(
                                arm_matrix_instance_f32 * S,
                                uint16_t nRows,
                                uint16_t nColumns,
                                float32_t    *pData);
```

```
        arm_status arm_mat_add_f32(
                                        const arm_matrix_instance_f32 * pSrcA,
                                        const arm_matrix_instance_f32 * pSrcB,
                                        arm_matrix_instance_f32 * pDst);



        arm_status arm_mat_trans_f32(
                                        const arm_matrix_instance_f32 * pSrc,
                                        arm_matrix_instance_f32 * pDst);



        arm_status arm_mat_mult_f32(
                                        const arm_matrix_instance_f32 * pSrcA,
                                        const arm_matrix_instance_f32 * pSrcB,
                                        arm_matrix_instance_f32 * pDst);



        arm_status arm_mat_sub_f32(
                                        const arm_matrix_instance_f32 * pSrcA,
                                        const arm_matrix_instance_f32 * pSrcB,
                                        arm_matrix_instance_f32 * pDst);
*/


/* Extended Kalman Filter algorithm ------------------------------------------------------*/

/* Variaveis Vetores e matrizes*/
float   X_data[7];
float   Z_data[6];
float   P_data[7*7];
float   F_data[7*7];
float   Ft_data[7*7];
float   FP_data[7*7];
float   FPFt_data[7*7];
float   Q_data[7*7];
float   h_data[6];
float   Y_data[6];
float   H_data[6*7];
float   Ht_data[7*6];
float   S_data[6*6];
float   Sinv_data[6*6];
float   PHt_data[7*6];
float   HPHt_data[6*6];
float   R_data[6*6];
float   K_data[7*6];
float   KY_data[7];
float   KH_data[7*7];
float   KHP_data[7*7];

float ax, ay, az, gx, gy, gz, mx, my, mz;
float accnorm = 0;
float magnorm = 0;
float m_x = 0;
float m_y = 0;
float m_z = 0;
float bx=0;
```

```
float bz=0;
float q[4]={1,0,0,0};
float qnorm = 0;
float wxb = 0;
float wyb = 0;
float wzb = 0;
float dt;

/* Estruturas ARM para operacao com Matrizes */
arm_matrix_instance_f32          X;
arm_matrix_instance_f32          Z;
arm_matrix_instance_f32          P;
arm_matrix_instance_f32          F;
arm_matrix_instance_f32          Ft;
arm_matrix_instance_f32          FP;
arm_matrix_instance_f32          FPFt;
arm_matrix_instance_f32          Q;
arm_matrix_instance_f32          h;
arm_matrix_instance_f32          Y;
arm_matrix_instance_f32          H;
arm_matrix_instance_f32          Ht;
arm_matrix_instance_f32          S;
arm_matrix_instance_f32          Sinv;
arm_matrix_instance_f32          PHt;
arm_matrix_instance_f32          HPHt;
arm_matrix_instance_f32          R;
arm_matrix_instance_f32          K;
arm_matrix_instance_f32          KY;
arm_matrix_instance_f32          KH;
arm_matrix_instance_f32          KHP;


void AHRS_EKFInit(){

        arm_mat_init_f32(&X, 7, 1, X_data);
        arm_mat_init_f32(&F, 7, 7, F_data);
        arm_mat_init_f32(&Ft, 7, 7, Ft_data);
        arm_mat_init_f32(&Z, 6, 1, Z_data);
        arm_mat_init_f32(&FP, 7, 7, FP_data);
        arm_mat_init_f32(&FPFt, 7, 7, FPFt_data);
        arm_mat_init_f32(&h, 6, 1, h_data);
        arm_mat_init_f32(&Y, 6, 1, Y_data);
        arm_mat_init_f32(&H, 6, 7, H_data);
        arm_mat_init_f32(&Ht, 7, 6, Ht_data);
        arm_mat_init_f32(&S, 6, 6, S_data);
        arm_mat_init_f32(&Sinv, 6, 6, Sinv_data);
        arm_mat_init_f32(&PHt, 7, 6, PHt_data);
        arm_mat_init_f32(&HPHt, 6, 6, HPHt_data);
        arm_mat_init_f32(&K, 7, 6, K_data);
        arm_mat_init_f32(&KY, 7, 1, KY_data);
        arm_mat_init_f32(&KH, 7, 7, KH_data);
        arm_mat_init_f32(&KHP, 7, 7, KHP_data);

        arm_mat_init_f32(&P, 7, 7, P_data);
        arm_fill_f32(0, P.pData, 7*7);
        P.pData[0] = 100000000;
        P.pData[8] = 100000000;
        P.pData[16] = 100000000;
        P.pData[24] = 100000000;
```

```
        P.pData[32] = 100000000;
        P.pData[40] = 100000000;
        P.pData[48] = 100000000;

        arm_mat_init_f32(&Q, 7, 7, Q_data);
        arm_fill_f32(0, Q.pData, 7*7);
        Q.pData[32] = 0.2;
        Q.pData[40] = 0.2;
        Q.pData[48] = 0.2;

        arm_mat_init_f32(&R, 6, 6, R_data);
        arm_fill_f32(0, R.pData, 6*6);
        R.pData[0] = 500000;
        R.pData[7] = 500000;
        R.pData[14] = 500000;
        R.pData[21] = 10000000;
        R.pData[28] = 10000000;
        R.pData[35] = 10000000;
}

void AHRS_EKFUpdate(float *Acc, float *Gyr, float *Mag, float *sampletime, float *Roll, float *Pitch

        ax = Acc[0];
        ay = Acc[1];
        az = Acc[2];
        gx = Gyr[0]*D2R;
        gy = Gyr[1]*D2R;
        gz = Gyr[2]*D2R;
        mx = Mag[0];
        my = Mag[1];
        mz = Mag[2];
        dt = *sampletime;

        dt = dt/2;

        // Predicao do estado
        X.pData[0] = q[0] + (dt)*(-q[1]*(gx-wxb) - q[2]*(gy-wyb) - q[3]*(gz-wzb));
        X.pData[1] = q[1] + (dt)*(q[0]*(gx-wxb) - q[3]*(gy-wyb) + q[2]*(gz-wzb));
        X.pData[2] = q[2] + (dt)*(q[3]*(gx-wxb) + q[0]*(gy-wyb) - q[1]*(gz-wzb));
        X.pData[3] = q[3] + (dt)*(-q[2]*(gx-wxb) + q[1]*(gy-wyb) + q[0]*(gz-wzb));
        X.pData[4] = wxb;
        X.pData[5] = wyb;
        X.pData[6] = wzb;

        // normaliza o quaternio
        qnorm = sqrt((X.pData[0])*(X.pData[0]) + (X.pData[1])*(X.pData[1]) + (X.pData[2])*(X.pData[2
        X.pData[0] = X.pData[0]/qnorm;
        X.pData[1] = X.pData[1]/qnorm;
        X.pData[2] = X.pData[2]/qnorm;
        X.pData[3] = X.pData[3]/qnorm;
        q[0] = X.pData[0];
        q[1] = X.pData[1];
        q[2] = X.pData[2];
        q[3] = X.pData[3];

        // constroi a matriz Jacobiana F
        F.pData[0] = 1;
        F.pData[1] = -(dt)*(gx-wxb);
        F.pData[2] = -(dt)*(gy-wyb);
```

```
F.pData[3] = -(dt)*(gz-wzb);
F.pData[4] = (dt)*q[1];
F.pData[5] = (dt)*q[2];
F.pData[6] = (dt)*q[3];

F.pData[7] = (dt)*(gx-wxb);
F.pData[8] = 1;
F.pData[9] = (dt)*(gz-wzb);
F.pData[10] = -(dt)*(gy-wyb);
F.pData[11] = -(dt)*q[0];
F.pData[12] = (dt)*q[3];
F.pData[13] = -(dt)*(q[2]);

F.pData[14] = (dt)*(gy-wyb);
F.pData[15] = -(dt)*(gz-wzb);
F.pData[16] = 1;
F.pData[17] = (dt)*(gx-wxb);
F.pData[18] = -(dt)*q[3];
F.pData[19] = -(dt)*q[0];
F.pData[20] = (dt)*q[1];

F.pData[21] = (dt)*(gz-wzb);
F.pData[22] = (dt)*(gy-wyb);
F.pData[23] = -(dt)*(gx-wxb);
F.pData[24] = 1;
F.pData[25] = (dt)*q[2];
F.pData[26] = -(dt)*q[1];
F.pData[27] = -(dt)*q[0];

F.pData[28] = 0;
F.pData[29] = 0;
F.pData[30] = 0;
F.pData[31] = 0;
F.pData[32] = 1;
F.pData[33] = 0;
F.pData[34] = 0;

F.pData[35] = 0;
F.pData[36] = 0;
F.pData[37] = 0;
F.pData[38] = 0;
F.pData[39] = 0;
F.pData[40] = 1;
F.pData[41] = 0;

F.pData[42] = 0;
F.pData[43] = 0;
F.pData[44] = 0;
F.pData[45] = 0;
F.pData[46] = 0;
F.pData[47] = 0;
F.pData[48] = 1;

// estima a covariancia P
arm_mat_trans_f32(&F, &Ft);
arm_mat_mult_f32(&F, &P, &FP);
arm_mat_mult_f32(&FP, &Ft, &FPFt);
arm_mat_add_f32(&FPFt, &Q, &P);
```

```
// normaliza o vetor Z
accnorm = sqrt(ax*ax + ay*ay + az*az);
Z.pData[0] = ax/accnorm;
Z.pData[1] = ay/accnorm;
Z.pData[2] = az/accnorm;
magnorm = sqrt(mx*mx + my*my + mz*mz);
Z.pData[3] = mx/magnorm;
Z.pData[4] = my/magnorm;
Z.pData[5] = mz/magnorm;

// constroi a matriz de rotacao de quaternio
m_x =   (Z.pData[3])*(q[0]*q[0] + q[1]*q[1] - q[2]*q[2] - q[3]*q[3])
                + 2*(Z.pData[4])*(q[1]*q[2] + q[0]*q[3])
                + 2*(Z.pData[5])*(q[1]*q[3] - q[0]*q[2]);

m_y =   2*(Z.pData[3])*(q[1]*q[2] - q[0]*q[3])
                + (Z.pData[4])*(q[0]*q[0] - q[1]*q[1] + q[2]*q[2] - q[3]*q[3])
                - 2*(Z.pData[5])*(q[2]*q[3] + q[0]*q[1]);

m_z =   2*(Z.pData[3])*(q[1]*q[3] + q[0]*q[2])
                + 2*(Z.pData[4])*(q[2]*q[3] - q[0]*q[1])
                + (Z.pData[5])*(q[0]*q[0] - q[1]*q[1] - q[2]*q[2] + q[3]*q[3]);

bx = sqrt(m_x*m_x + m_y*m_y);
bz = m_z;


h.pData[0] = -2*(q[1]*q[3] - q[0]*q[2]);
h.pData[1] = -2*(q[2]*q[3] + q[0]*q[1]);
h.pData[2] = -q[0]*q[0] + q[1]*q[1] + q[2]*q[2] - q[3]*q[3];
h.pData[3] = bx*(q[0]*q[0] + q[1]*q[1] - q[2]*q[2] - q[3]*q[3])
                        + 2*bz*(q[1]*q[3] - q[0]*q[2]);
h.pData[4] = 2*bx*(q[1]*q[2] - q[0]*q[3]) + 2*bz*(q[2]*q[3] + q[0]*q[1]);
h.pData[5] = 2*bx*(q[1]*q[3] + q[0]*q[2])
                        + bz*(q[0]*q[0] - q[1]*q[1] - q[2]*q[2] + q[3]*q[3]);

//residual de medicao
arm_mat_sub_f32(&Z, &h, &Y);


H.pData[0] = 2*q[2];
H.pData[1] = -2*q[3];
H.pData[2] = 2*q[0];
H.pData[3] = -2*q[1];
H.pData[4] = 0;
H.pData[5] = 0;
H.pData[6] = 0;

H.pData[7] = -2*q[1];
H.pData[8] = -2*q[0];
H.pData[9] = -2*q[3];
H.pData[10] = -2*q[2];
H.pData[11] = 0;
H.pData[12] = 0;
H.pData[13] = 0;

H.pData[14] = -2*q[0];
H.pData[15] = 2*q[1];
H.pData[16] = 2*q[2];
```

```
H.pData[17] = -2*q[3];
H.pData[18] = 0;
H.pData[19] = 0;
H.pData[20] = 0;

H.pData[21] = 2*(q[0]*bx-q[2]*bz);
H.pData[22] = 2*(q[1]*bx+q[3]*bz);
H.pData[23] = 2*(-q[2]*bx-q[0]*bz);
H.pData[24] = 2*(-q[3]*bx+q[1]*bz);
H.pData[25] = 0;
H.pData[26] = 0;
H.pData[27] = 0;

H.pData[28] = 2*(-q[3]*bx+q[1]*bz);
H.pData[29] = 2*(q[2]*bx+q[0]*bz);
H.pData[30] = 2*(q[1]*bx+q[3]*bz);
H.pData[31] = 2*(-q[0]*bx+q[2]*bz);
H.pData[32] = 0;
H.pData[33] = 0;
H.pData[34] = 0;

H.pData[35] = 2*(q[2]*bx+q[0]*bz);
H.pData[36] = 2*(q[3]*bx-q[1]*bz);
H.pData[37] = 2*(q[0]*bx-q[2]*bz);
H.pData[38] = 2*(q[1]*bx+q[3]*bz);
H.pData[39] = 0;
H.pData[40] = 0;
H.pData[41] = 0;


arm_mat_trans_f32(&H, &Ht);
arm_mat_mult_f32(&P, &Ht, &PHt);
arm_mat_mult_f32(&H, &PHt, &HPHt);
arm_mat_add_f32(&HPHt, &R, &S);


arm_mat_inverse_f32(&S, &Sinv);
arm_mat_mult_f32(&PHt, &Sinv, &K);


arm_mat_mult_f32(&K, &Y, &KY);
arm_mat_add_f32(&X, &KY, &X);


qnorm = sqrt((X.pData[0])*(X.pData[0]) + (X.pData[1])*(X.pData[1]) + (X.pData[2])*(X.pData[2
X.pData[0] = X.pData[0]/qnorm;
X.pData[1] = X.pData[1]/qnorm;
X.pData[2] = X.pData[2]/qnorm;
X.pData[3] = X.pData[3]/qnorm;
q[0] = X.pData[0];
q[1] = X.pData[1];
q[2] = X.pData[2];
q[3] = X.pData[3];


arm_mat_mult_f32(&K, &H, &KH);
arm_mat_mult_f32(&KH, &P, &KHP);
arm_mat_sub_f32(&P, &KHP, &P);
```

```
        *Pitch = (double)(R2D*atan2(2*(q[0]*q[1] + q[2]*q[3]), 1-2*(q[1]*q[1]+q[2]*q[2])));
        *Roll = (double)(R2D*asin(2*(q[0]*q[2]-q[3]*q[1])));
        *Yaw = (double)(R2D*atan2(2*(q[0]*q[3] + q[1]*q[2]), 1-2*(q[2]*q[2] + q[3]*q[3])));
}
```

A.0.4 odometry_kf.h

```
#include <stdio.h>
#include <math.h>
#include "stm32f30x.h"
#include "stm32f3_discovery.h"
#include "stm32f3_discovery_lsm303dlhc.h"
#include "stm32f3_discovery_l3gd20.h"

/* Biblioteca da ARM para operar com DSP */
#include "arm_math.h"


#define R2D                         (float) 57.2957795131f
#define D2R                         (float)  0.0174532925f

void Odometry_KFInit();
void Odometry_KFUpdate(float *Acc, float *sampletime, float *velx, float *vely);
```

A.0.5 odometry_kf.c

```
#include "odometry_kf.h"

/* Biblioteca da ARM para operar com DSP:
        #include "arm_math.h"

        typedef struct
        {
                uint16_t numRows;       < number of rows of the matrix.
                uint16_t numCols;       < number of columns of the matrix.
                float32_t *pData;       < points to the data of the matrix.
        } arm_matrix_instance_f32;



        void arm_mat_init_f32(
                                arm_matrix_instance_f32 * S,
                                uint16_t nRows,
                                uint16_t nColumns,
                                float32_t    *pData);



        arm_status arm_mat_add_f32(
                                        const arm_matrix_instance_f32 * pSrcA,
                                        const arm_matrix_instance_f32 * pSrcB,
                                        arm_matrix_instance_f32 * pDst);



        arm_status arm_mat_trans_f32(
                                        const arm_matrix_instance_f32 * pSrc,
                                        arm_matrix_instance_f32 * pDst);
```

```
        arm_status arm_mat_mult_f32(
                                        const arm_matrix_instance_f32 * pSrcA,
                                        const arm_matrix_instance_f32 * pSrcB,
                                        arm_matrix_instance_f32 * pDst);


        arm_status arm_mat_sub_f32(
                                        const arm_matrix_instance_f32 * pSrcA,
                                        const arm_matrix_instance_f32 * pSrcB,
                                        arm_matrix_instance_f32 * pDst);
*/

/* Extended Kalman Filter algorithm ------------------------------------------------------*/

/* Variaveis Vetores e matrizes*/
float   Xv_data[4];
float   Zv_data[2];
float   Pv_data[4*4];
float   Fv_data[4*4];
float   Fvt_data[4*4];
float   FPv_data[4*4];
float   FPFvt_data[4*4];
float   Qv_data[4*4];
float   hv_data[2];
float   Yv_data[2];
float   Hv_data[2*4];
float   Hvt_data[4*2];
float   Sv_data[2*2];
float   Sinvv_data[2*2];
float   PHvt_data[4*2];
float   HPHvt_data[2*2];
float   Rv_data[2*2];
float   Kv_data[4*2];
float   KYv_data[4];
float   KHv_data[4*4];
float   KHPv_data[4*4];

float ax, ay;
float dt;

/* Estruturas ARM para operacao com Matrizes */
arm_matrix_instance_f32         Xv;
arm_matrix_instance_f32         Zv;
arm_matrix_instance_f32         Pv;
arm_matrix_instance_f32         Fv;
arm_matrix_instance_f32         Fvt;
arm_matrix_instance_f32         FPv;
arm_matrix_instance_f32         FPFvt;
arm_matrix_instance_f32         Qv;
arm_matrix_instance_f32         hv;
arm_matrix_instance_f32         Yv;
arm_matrix_instance_f32         Hv;
arm_matrix_instance_f32         Hvt;
arm_matrix_instance_f32         Sv;
```

```
arm_matrix_instance_f32          Sinvv;
arm_matrix_instance_f32          PHvt;
arm_matrix_instance_f32          HPHvt;
arm_matrix_instance_f32          Rv;
arm_matrix_instance_f32          Kv;
arm_matrix_instance_f32          KYv;
arm_matrix_instance_f32          KHv;
arm_matrix_instance_f32          KHPv;


void Odometry_KFInit(){

        arm_mat_init_f32(&Xv, 4, 1, Xv_data);
        Xv.pData[0] = 0;
        Xv.pData[1] = 0;
        Xv.pData[2] = 0;
        Xv.pData[3] = 0;

        arm_mat_init_f32(&Fv, 4, 4, Fv_data);
        arm_mat_init_f32(&Fvt, 4, 4, Fvt_data);
        arm_mat_init_f32(&Zv, 2, 1, Zv_data);
        arm_mat_init_f32(&FPv, 4, 4, FPv_data);
        arm_mat_init_f32(&FPFvt, 4, 4, FPFvt_data);
        arm_mat_init_f32(&hv, 2, 1, hv_data);
        arm_mat_init_f32(&Yv, 2, 1, Yv_data);
        arm_mat_init_f32(&Hv, 2, 4, Hv_data);
        arm_mat_init_f32(&Hvt, 4, 2, Hvt_data);
        arm_mat_init_f32(&Sv, 2, 2, Sv_data);
        arm_mat_init_f32(&Sinvv, 2, 2, Sinvv_data);
        arm_mat_init_f32(&PHvt, 4, 2, PHvt_data);
        arm_mat_init_f32(&HPHvt, 2, 2, HPHvt_data);
        arm_mat_init_f32(&Kv, 4, 2, Kv_data);
        arm_mat_init_f32(&KYv, 4, 1, KYv_data);
        arm_mat_init_f32(&KHv, 4, 4, KHv_data);
        arm_mat_init_f32(&KHPv, 4, 4, KHPv_data);

        arm_mat_init_f32(&Pv, 4, 4, Pv_data);
        arm_fill_f32(0, Pv.pData, 4*4);
        Pv.pData[0] = 1;
        Pv.pData[5] = 1;
        Pv.pData[10] = 1;
        Pv.pData[15] = 1;

        arm_mat_init_f32(&Qv, 4, 4, Qv_data);
        arm_fill_f32(0, Qv.pData, 4*4);
        Qv.pData[0] = 0;
        Qv.pData[5] = 0;
        Qv.pData[10] = 0.0002;
        Qv.pData[15] = 0.0002;

        arm_mat_init_f32(&Rv, 2, 2, Rv_data);
        arm_fill_f32(0, Rv.pData, 2*2);
        Rv.pData[0] = 1000;
        Rv.pData[3] = 1000;
}

void Odometry_KFUpdate(float *Acc, float *sampletime, float *velocx, float *velocy){

        ax = -Acc[0]*9.8;
```

```
ay = -Acc[1]*9.8;
dt = *sampletime;

// Predicao do estado
Xv.pData[0] += dt*Xv.pData[2];
Xv.pData[1] += dt*Xv.pData[3];
//X.pData[2]
//X.pData[3]


// constroi a matriz Jacobiana F
Fv.pData[0] = 1;
Fv.pData[1] = 0;
Fv.pData[2] = dt;
Fv.pData[3] = 0;

Fv.pData[4] = 0;
Fv.pData[5] = 1;
Fv.pData[6] = 0;
Fv.pData[7] = dt;

Fv.pData[8] = 0;
Fv.pData[9] = 0;
Fv.pData[10] = 1;
Fv.pData[11] = 0;

Fv.pData[12] = 0;
Fv.pData[13] = 0;
Fv.pData[14] = 0;
Fv.pData[15] = 1;


// estima a covariancia P
arm_mat_trans_f32(&Fv, &Fvt);
arm_mat_mult_f32(&Fv, &Pv, &FPv);
arm_mat_mult_f32(&FPv, &Fvt, &FPFvt);
arm_mat_add_f32(&FPFvt, &Qv, &Pv);


hv.pData[0] = Xv.pData[2];
hv.pData[1] = Xv.pData[3];

Zv.pData[0] = ax;
Zv.pData[1] = ay;
//residual de medicao
arm_mat_sub_f32(&Zv, &hv, &Yv);


Hv.pData[0] = 0;
Hv.pData[1] = 0;
Hv.pData[2] = 1;
Hv.pData[3] = 0;

Hv.pData[4] = 0;
Hv.pData[5] = 0;
Hv.pData[6] = 0;
Hv.pData[7] = 1;
```

```
        arm_mat_trans_f32(&Hv, &Hvt);
        arm_mat_mult_f32(&Pv, &Hvt, &PHvt);
        arm_mat_mult_f32(&Hv, &PHvt, &HPHvt);
        arm_mat_add_f32(&HPHvt, &Rv, &Sv);


        arm_mat_inverse_f32(&Sv, &Sinvv);
        arm_mat_mult_f32(&PHvt, &Sinvv, &Kv);


        arm_mat_mult_f32(&Kv, &Yv, &KYv);
        arm_mat_add_f32(&Xv, &KYv, &Xv);


        arm_mat_mult_f32(&Kv, &Hv, &KHv);
        arm_mat_mult_f32(&KHv, &Pv, &KHPv);
        arm_mat_sub_f32(&Pv, &KHPv, &Pv);

        *velocx = Xv.pData[0];
        *velocy = Xv.pData[1];
}
```

A.0.6 ultrassom.h

```
#include <stdio.h>
#include <math.h>
#include "stm32f30x.h"
#include "stm32f3_discovery.h"
#include "usart.h"

extern float Altitude;

void Ultrassom_TIM_Init(void);
void TIM2_IRQHandler(void);

void Ultrassom_InitIO(void);
void UltrassomTrigger(void);
```

A.0.7 ultrassom.c

```
#include "ultrassom.h"

int ICValue1 = 0;
int ICValue2 = 0;
int ICPeriod = 0;
float Altitude=0;

/* ------------------------------------------------------------------------
        TIM3 Configuration: Output Compare Timing Mode:

        In this example TIM3 input clock (TIM3CLK) is set to 2 * APB1 clock (PCLK1),
        since APB1 prescaler is different from 1.
          TIM3CLK = 2 * PCLK1
          PCLK1 = HCLK / 2
          => TIM3CLK = HCLK = SystemCoreClock

        To get TIM3 counter clock at 72 MHz, the prescaler is computed as follows:
           Prescaler = (TIM3CLK / TIM3 counter clock) - 1
```

```
            Prescaler = ((SystemCoreClock) /72 MHz) - 1

        Note:
         SystemCoreClock variable holds HCLK frequency and is defined in system_stm32f3xx.c file.
         Each time the core clock (HCLK) changes, user had to call SystemCoreClockUpdate()
         function to update SystemCoreClock variable value. Otherwise, any configuration
         based on this variable will be incorrect.
------------------------------------------------------------------------------ */
#define TIM2_Frequency          4000000 // 4MHz
#define TIM2_Period                     ((uint32_t)0xFFFFFFFF)
/*****************************************************************************
                                                                  FUNCOES DO TIMER 2
*****************************************************************************/
void Ultrassom_TIM_Init(void){

        TIM_TimeBaseInitTypeDef         TIM_TimeBaseStructure;
        TIM_ICInitTypeDef                       TIM_ICInitStructure;
        NVIC_InitTypeDef                        NVIC_InitStructure;
        uint16_t                                        PrescalerValue;

        // TIM2 clock enable
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

        /* Compute the prescaler value */
        PrescalerValue = (uint16_t) ((SystemCoreClock) / TIM2_Frequency) - 1;

        /* Time base configuration */
        TIM_TimeBaseStructure.TIM_Period = TIM2_Period;
        TIM_TimeBaseStructure.TIM_Prescaler = 0;
        TIM_TimeBaseStructure.TIM_ClockDivision = 0;
        TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

        TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
        /* Prescaler configuration */
        TIM_PrescalerConfig(TIM2, PrescalerValue, TIM_PSCReloadMode_Immediate);



        /* Input Capture configuration */
        TIM_ICInitStructure.TIM_Channel = TIM_Channel_2;
        TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_BothEdge;
        TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
        TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
        TIM_ICInitStructure.TIM_ICFilter = 0;

        TIM_ICInit(TIM2, &TIM_ICInitStructure);


        NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

        NVIC_Init(&NVIC_InitStructure);

        TIM_ITConfig(TIM2, TIM_IT_CC2, ENABLE);
}
```

```
void TIM2_IRQHandler(void){

        static uint8_t flag = 0;

        //STM_EVAL_LEDToggle(LED3);

        if (TIM_GetITStatus(TIM2, TIM_IT_CC2) != RESET){

                TIM_ClearITPendingBit(TIM2, TIM_IT_CC2);

                if(flag == 0){

                        ICValue1 = TIM_GetCapture2(TIM2);
                        flag = 1;
                }
                else{

                        ICValue2 = TIM_GetCapture2(TIM2);

                        if(ICValue1 < ICValue2){
                                ICPeriod = ICValue2 - ICValue1;
                        }
                        else{
                                ICPeriod = TIM2_Period - ICValue1 + ICValue2;

                        }
                        flag = 0;

                }

                Altitude = ((float)ICPeriod/(float)TIM2_Frequency)*17000;
        } // if externo

}



/*****************************************************************************
*****************************************************************************/


void Ultrassom_InitIO(){

        // Pin PD4 - Echo (Input Capture)
        // Pin PD3 - Trigger

        GPIO_InitTypeDef        GPIO_InitStructure;

        RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOD, ENABLE);


        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
```

```
        GPIO_Init(GPIOD, &GPIO_InitStructure);
        GPIO_PinAFConfig(GPIOD, GPIO_PinSource4, GPIO_AF_2);


        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;

        GPIO_Init(GPIOD, &GPIO_InitStructure);
        GPIO_ResetBits(GPIOD, GPIO_Pin_3);
}



void UltrassomTrigger(){

                TIM_Cmd(TIM2, DISABLE);
                TIM_SetCounter(TIM2, 0);

                GPIO_SetBits(GPIOD, GPIO_Pin_3);
                TIM_Cmd(TIM2, ENABLE);
                while(TIM_GetCounter(TIM2)<40){
                }
                GPIO_ResetBits(GPIOD, GPIO_Pin_3);

}
```

A.0.8 imu.h

```
#include <stdio.h>
#include <math.h>
#include "stm32f30x.h"
#include "stm32f3_discovery.h"
#include "stm32f3_discovery_lsm303dlhc.h"
#include "stm32f3_discovery_l3gd20.h"

#define R2D                         (float) 57.2957795131f
#define D2R                         (float)  0.0174532925f

#define L3G_Sensitivity_250dps      (float)       114.285f        /*!< gyroscope sensitivity with 2
#define L3G_Sensitivity_500dps      (float)    57.1429f        /*!< gyroscope sensitivity with 500 dp
#define L3G_Sensitivity_2000dps     (float)    14.285f         /*!< gyroscope sensitivity with 2000 d
//#define PI                          (float)    3.14159265f

#define LSM_Acc_Sensitivity_2g      (float)     1.0f            /*!< accelerometer sensitivity with 2
#define LSM_Acc_Sensitivity_4g      (float)     0.5f            /*!< accelerometer sensitivity with 4
#define LSM_Acc_Sensitivity_8g      (float)     0.25f           /*!< accelerometer sensitivity with 8
#define LSM_Acc_Sensitivity_16g     (float)     0.0834f         /*!< accelerometer sensitivity with 1


void GyroConfig(void);
void CompassConfig(void);
void GyroReadAngRate (float* pfData);
void CompassReadMag (float* pfData);
void CompassReadAcc(float* pfData);
float InvSqrt(float x);
```

A.0.9 imu.c

```
#include "imu.h"



/* Sensor reading functions ------------------------------------------------------*/


/**
  * @brief  Configure the Mems to gyroscope application.
  * @param  None
  * @retval None
  */
void GyroConfig(void){

        L3GD20_InitTypeDef                      L3GD20_InitStructure;
        L3GD20_FilterConfigTypeDef      L3GD20_FilterStructure;

        /* Configure Mems L3GD20 */
        L3GD20_InitStructure.Power_Mode = L3GD20_MODE_ACTIVE;
        L3GD20_InitStructure.Output_DataRate = L3GD20_OUTPUT_DATARATE_1;
        L3GD20_InitStructure.Axes_Enable = L3GD20_AXES_ENABLE;
        L3GD20_InitStructure.Band_Width = L3GD20_BANDWIDTH_4;
        L3GD20_InitStructure.BlockData_Update = L3GD20_BlockDataUpdate_Continous;
        L3GD20_InitStructure.Endianness = L3GD20_BLE_LSB;
        L3GD20_InitStructure.Full_Scale = L3GD20_FULLSCALE_2000;
        L3GD20_Init(&L3GD20_InitStructure);

        L3GD20_FilterStructure.HighPassFilter_Mode_Selection =L3GD20_HPM_NORMAL_MODE_RES;
        L3GD20_FilterStructure.HighPassFilter_CutOff_Frequency = L3GD20_HPFCF_0;
        L3GD20_FilterConfig(&L3GD20_FilterStructure) ;

        L3GD20_FilterCmd(L3GD20_HIGHPASSFILTER_ENABLE);
}




/**
  * @brief  Calculate the angular Data rate Gyroscope.
  * @param  pfData : Data out pointer
  * @retval None
  */
void GyroReadAngRate (float* pfData){

        uint8_t tmpbuffer[6] ={0};
        int16_t RawData[3] = {0};
        uint8_t tmpreg = 0;
        float sensitivity = 0;
        int i =0;

        L3GD20_Read(&tmpreg,L3GD20_CTRL_REG4_ADDR,1);

        L3GD20_Read(tmpbuffer,L3GD20_OUT_X_L_ADDR,6);

        /* check in the control register 4 the data alignment (Big Endian or Little Endian)*/
        if(!(tmpreg & 0x40)){
                for(i=0; i<3; i++){
```

```
                        RawData[i]=(int16_t)(((uint16_t)tmpbuffer[2*i+1] << 8) + tmpbuffer[2*i]);
                }
        }
        else{
                for(i=0; i<3; i++){
                        RawData[i]=(int16_t)(((uint16_t)tmpbuffer[2*i] << 8) + tmpbuffer[2*i+1]);
                }
        }

        /* Switch the sensitivity value set in the CRTL4 */
        switch(tmpreg & 0x30){
                case 0x00:
                                sensitivity=L3G_Sensitivity_250dps;
                                break;

                case 0x10:
                                sensitivity=L3G_Sensitivity_500dps;
                                break;

                case 0x20:
                                sensitivity=L3G_Sensitivity_2000dps;
                                break;
        }
        /* divide by sensitivity */
        pfData[0]=(float)(-RawData[1]/sensitivity);//*D2R;
        pfData[1]=(float)(RawData[0]/sensitivity);//*D2R;
        pfData[2]=(float)(RawData[2]/sensitivity);//*D2R;
}




/**
  * @brief  Configure the Mems to compass application.
  * @param  None
  * @retval None
  */
void CompassConfig(void){

        LSM303DLHCMag_InitTypeDef                       LSM303DLHC_InitStructure;
        LSM303DLHCAcc_InitTypeDef                       LSM303DLHCAcc_InitStructure;
        LSM303DLHCAcc_FilterConfigTypeDef       LSM303DLHCFilter_InitStructure;

        /* Configure MEMS magnetometer main parameters: temp, working mode, full Scale and Data rate
        LSM303DLHC_InitStructure.Temperature_Sensor = LSM303DLHC_TEMPSENSOR_ENABLE;
        LSM303DLHC_InitStructure.MagOutput_DataRate =LSM303DLHC_ODR_220_HZ;
        LSM303DLHC_InitStructure.MagFull_Scale = LSM303DLHC_FS_1_3_GA;
        LSM303DLHC_InitStructure.Working_Mode = LSM303DLHC_CONTINUOS_CONVERSION;
        LSM303DLHC_MagInit(&LSM303DLHC_InitStructure);

        /* Fill the accelerometer structure */
        LSM303DLHCAcc_InitStructure.Power_Mode = LSM303DLHC_NORMAL_MODE;
        LSM303DLHCAcc_InitStructure.AccOutput_DataRate = LSM303DLHC_ODR_100_HZ;
        LSM303DLHCAcc_InitStructure.Axes_Enable= LSM303DLHC_AXES_ENABLE;
        LSM303DLHCAcc_InitStructure.AccFull_Scale = LSM303DLHC_FULLSCALE_4G;
        LSM303DLHCAcc_InitStructure.BlockData_Update = LSM303DLHC_BlockUpdate_Continous;
        LSM303DLHCAcc_InitStructure.Endianness=LSM303DLHC_BLE_LSB;
        LSM303DLHCAcc_InitStructure.High_Resolution=LSM303DLHC_HR_ENABLE;
        /* Configure the accelerometer main parameters */
```

```
        LSM303DLHC_AccInit(&LSM303DLHCAcc_InitStructure);

        /* Fill the accelerometer LPF structure */
        LSM303DLHCFilter_InitStructure.HighPassFilter_Mode_Selection =LSM303DLHC_HPM_NORMAL_MODE;
        LSM303DLHCFilter_InitStructure.HighPassFilter_CutOff_Frequency = LSM303DLHC_HPFCF_16;
        LSM303DLHCFilter_InitStructure.HighPassFilter_AOI1 = LSM303DLHC_HPF_AOI1_DISABLE;
        LSM303DLHCFilter_InitStructure.HighPassFilter_AOI2 = LSM303DLHC_HPF_AOI2_DISABLE;

        /* Configure the accelerometer LPF main parameters */
        LSM303DLHC_AccFilterConfig(&LSM303DLHCFilter_InitStructure);
}




/**
* @brief Read LSM303DLHC output register, and calculate the acceleration ACC=(1/SENSITIVITY)* (out_h
* @param pnData: pointer to float buffer where to store data
* @retval None
*/
void CompassReadAcc(float* pfData){

        int16_t pnRawData[3];
        uint8_t ctrlx[2];
        uint8_t buffer[6], cDivider;
        uint8_t i = 0;
        float LSM_Acc_Sensitivity = LSM_Acc_Sensitivity_4g;

        /* Read the register content */
        LSM303DLHC_Read(ACC_I2C_ADDRESS, LSM303DLHC_CTRL_REG4_A, ctrlx,2);
        LSM303DLHC_Read(ACC_I2C_ADDRESS, LSM303DLHC_OUT_X_L_A, buffer, 6);

        if(ctrlx[1]&0x40)
                cDivider=64;
        else
                cDivider=16;

        /* check in the control register4 the data alignment*/
        if(!(ctrlx[0] & 0x40) || (ctrlx[1] & 0x40)){ /* Little Endian Mode or FIFO mode */
                for(i=0; i<3; i++){
                        pnRawData[i]=((int16_t)((uint16_t)buffer[2*i+1] << 8) + buffer[2*i])/cDivide
                }
        }
        else{ /* Big Endian Mode */
                for(i=0; i<3; i++)
                        pnRawData[i]=((int16_t)((uint16_t)buffer[2*i] << 8) + buffer[2*i+1])/cDivide
                }
        /* Read the register content */
        LSM303DLHC_Read(ACC_I2C_ADDRESS, LSM303DLHC_CTRL_REG4_A, ctrlx,2);


        if(ctrlx[1]&0x40){
                /* FIFO mode */
                LSM_Acc_Sensitivity = 0.25;
        }
        else{
                /* normal mode */
                /* switch the sensitivity value set in the CRTL4*/
                switch(ctrlx[0] & 0x30){
```

```
                        case LSM303DLHC_FULLSCALE_2G:
                                        LSM_Acc_Sensitivity = LSM_Acc_Sensitivity_2g;
                                        break;
                        case LSM303DLHC_FULLSCALE_4G:
                                        LSM_Acc_Sensitivity = LSM_Acc_Sensitivity_4g;
                                        break;
                        case LSM303DLHC_FULLSCALE_8G:
                                        LSM_Acc_Sensitivity = LSM_Acc_Sensitivity_8g;
                                        break;
                        case LSM303DLHC_FULLSCALE_16G:
                                        LSM_Acc_Sensitivity = LSM_Acc_Sensitivity_16g;
                                        break;
                }
        }

  /* Obtain the g value for the three axis */
        for(i=0; i<3; i++){
                pfData[i]=(float)-pnRawData[i]*0.001/LSM_Acc_Sensitivity;
        }

}




/**
  * @brief  calculate the magnetic field Magn.
* @param  pfData: pointer to the data out
  * @retval None
  */
void CompassReadMag (float* pfData){

        static uint8_t buffer[6] = {0};
        uint8_t CTRLB = 0;
        uint16_t Magn_Sensitivity_XY = 0, Magn_Sensitivity_Z = 0;
        uint8_t i =0;
        LSM303DLHC_Read(MAG_I2C_ADDRESS, LSM303DLHC_CRB_REG_M, &CTRLB, 1);

        LSM303DLHC_Read(MAG_I2C_ADDRESS, LSM303DLHC_OUT_X_H_M, buffer, 1);
        LSM303DLHC_Read(MAG_I2C_ADDRESS, LSM303DLHC_OUT_X_L_M, buffer+1, 1);
        LSM303DLHC_Read(MAG_I2C_ADDRESS, LSM303DLHC_OUT_Y_H_M, buffer+2, 1);
        LSM303DLHC_Read(MAG_I2C_ADDRESS, LSM303DLHC_OUT_Y_L_M, buffer+3, 1);
        LSM303DLHC_Read(MAG_I2C_ADDRESS, LSM303DLHC_OUT_Z_H_M, buffer+4, 1);
        LSM303DLHC_Read(MAG_I2C_ADDRESS, LSM303DLHC_OUT_Z_L_M, buffer+5, 1);
        /* Switch the sensitivity set in the CRTLB*/

        switch(CTRLB & 0xE0){

        case LSM303DLHC_FS_1_3_GA:
                        Magn_Sensitivity_XY = LSM303DLHC_M_SENSITIVITY_XY_1_3Ga;
                        Magn_Sensitivity_Z = LSM303DLHC_M_SENSITIVITY_Z_1_3Ga;
                        break;
        case LSM303DLHC_FS_1_9_GA:
                        Magn_Sensitivity_XY = LSM303DLHC_M_SENSITIVITY_XY_1_9Ga;
                        Magn_Sensitivity_Z = LSM303DLHC_M_SENSITIVITY_Z_1_9Ga;
                        break;
        case LSM303DLHC_FS_2_5_GA:
                        Magn_Sensitivity_XY = LSM303DLHC_M_SENSITIVITY_XY_2_5Ga;
                        Magn_Sensitivity_Z = LSM303DLHC_M_SENSITIVITY_Z_2_5Ga;
```

```
                        break;
        case LSM303DLHC_FS_4_0_GA:
                        Magn_Sensitivity_XY = LSM303DLHC_M_SENSITIVITY_XY_4Ga;
                        Magn_Sensitivity_Z = LSM303DLHC_M_SENSITIVITY_Z_4Ga;
                        break;
        case LSM303DLHC_FS_4_7_GA:
                        Magn_Sensitivity_XY = LSM303DLHC_M_SENSITIVITY_XY_4_7Ga;
                        Magn_Sensitivity_Z = LSM303DLHC_M_SENSITIVITY_Z_4_7Ga;
                        break;
        case LSM303DLHC_FS_5_6_GA:
                        Magn_Sensitivity_XY = LSM303DLHC_M_SENSITIVITY_XY_5_6Ga;
                        Magn_Sensitivity_Z = LSM303DLHC_M_SENSITIVITY_Z_5_6Ga;
                        break;
        case LSM303DLHC_FS_8_1_GA:
                        Magn_Sensitivity_XY = LSM303DLHC_M_SENSITIVITY_XY_8_1Ga;
                        Magn_Sensitivity_Z = LSM303DLHC_M_SENSITIVITY_Z_8_1Ga;
                        break;
        }

        for(i=0; i<2; i++){
                pfData[i]=(float)((int16_t)(((uint16_t)buffer[2*i] << 8) + buffer[2*i+1])*1000)/Magn
        }
        pfData[2]=(float)((int16_t)(((uint16_t)buffer[4] << 8) + buffer[5])*1000)/Magn_Sensitivity_Z
/*
  // Normalise magnetometer measurement
  float recipNorm;
  recipNorm = InvSqrt(pfData[0] * pfData[0] + pfData[1] * pfData[1] + pfData[2] * pfData[2]);
  pfData[0] *= recipNorm;
  pfData[1] *= recipNorm;
  pfData[2] *= recipNorm;
  */
}




/**
  * @brief  Basic management of the timeout situation.
  * @param  None.
  * @retval None.
  */
uint32_t LSM303DLHC_TIMEOUT_UserCallback(void)
{
        return 0;
}




/**
  * @brief  Basic management of the timeout situation.
  * @param  None.
  * @retval None.
  */
uint32_t L3GD20_TIMEOUT_UserCallback(void)
{
        return 0;
}
```

```
#ifdef USE_FULL_ASSERT

/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t* file, uint32_t line)
{
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */

  /* Infinite loop */
        while (1)
        {
        }
}
#endif




float InvSqrt(float x) {
        float halfx = 0.5f * x;
        float y = x;
        long i = *(long*)&y;
        i = 0x5f3759df - (i>>1);
        y = *(float*)&i;
        y = y * (1.5f - (halfx * y * y));
        return y;
}
```

A.0.10 usart.h

```
#include <stdio.h>
#include <math.h>
#include "stm32f30x.h"


// Funcoes para USART1 - Informacoes do PIC
void USART1_Init(uint32_t speed);
void USART1_PutS(char *pcString);
void USART1_PutC(char ch);
char USART1_GetC(void);
void USART1_IRQHandler(void);
int USART1_kbhit(void);

// Funcoes para USART2 - Comunicacao PC
void USART2_Init(uint32_t speed);
void USART2_PutS(char *pcString);
void USART2_PutC(char ch);
char USART2_GetC(void);
void USART2_IRQHandler(void);
```

```
int USART2_kbhit(void);
```

A.0.11 usart.c

```
#include "usart.h"

/*****************************************************************************
                                                                FUNCOES DA USART1
*****************************************************************************/
void USART1_Init(uint32_t speed){

        USART_InitTypeDef                USART_InitStructure;            // estrutura para USART
        GPIO_InitTypeDef                 GPIO_InitStructure;                     // estrutura para GP
        NVIC_InitTypeDef                 NVIC_InitStructure;                     // estrutura para In


        USART_InitStructure.USART_BaudRate = speed;
        USART_InitStructure.USART_WordLength = USART_WordLength_8b;
        USART_InitStructure.USART_StopBits = USART_StopBits_1;
        USART_InitStructure.USART_Parity = USART_Parity_No;
        USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
        USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

        /* Enable GPIO clock */
        RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC, ENABLE);

        /* Enable USART clock */
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);

        /* Connect PXx to USARTx_Tx - PIN C4 - TX*/
        GPIO_PinAFConfig(GPIOC, GPIO_PinSource4, GPIO_AF_7);

        /* Connect PXx to USARTx_Rx - PIN C5 - RX*/
        GPIO_PinAFConfig(GPIOC, GPIO_PinSource5, GPIO_AF_7);

        /* Configure USART Tx as alternate function push-pull */
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
        GPIO_Init(GPIOC, &GPIO_InitStructure);

        /* Configure USART Rx as alternate function push-pull */
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
        GPIO_Init(GPIOC, &GPIO_InitStructure);

        /* USART configuration */
        USART_Init(USART1, &USART_InitStructure);

        /* Enable the USART1 Receive interrupt: this interrupt is generated when the
                    USART1 receive data register is not empty */
        USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);

        /* Enable the USART1 Interrupt */
        NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
        NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
```

```
        NVIC_Init(&NVIC_InitStructure);


        /* Enable USART */
        USART_Cmd(USART1, ENABLE);

}

char USART1_GetC(void){
        return USART_ReceiveData(USART1);
}

void USART1_PutC(char ch){
        /* Put character on the serial line */
        USART1->TDR = (ch & (uint16_t)0x01FF);

        /* Loop until transmit data register is empty */
        while ((USART1->ISR & USART_FLAG_TXE) == (uint16_t) RESET);
}

void USART1_PutS(char *pcString){
        int i = 0;
        while (pcString[i] != 0){
                USART1_PutC(pcString[i]);
                i++;
        }
}

void USART1_IRQHandler(void){
        static int rx1_index = 0;
        rx1_index++;
/*

*/
}

int USART1_kbhit(void){
        return USART_GetFlagStatus(USART1, USART_FLAG_RXNE);
}


/*****************************************************************************
                                                                FUNCOES DA USART2
*****************************************************************************/
void USART2_Init(uint32_t speed){

        USART_InitTypeDef       USART_InitStructure;            // estrutura para USART
        GPIO_InitTypeDef        GPIO_InitStructure;                     // estrutura para GPIO

        /*!< At this stage the microcontroller clock setting is already configured,
                                this is done through SystemInit() function which is called from startup
                                file (startup_stm32f37x.s) before to branch to application main.
                                To reconfigure the default setting of SystemInit() function, refer to
                                system_stm32f37x.c file
         */
        /* USARTx configured as follow:
                                - BaudRate = 115200 baud
                                - Word Length = 8 Bits
                                - One Stop Bit
```

```
                                    - No parity
                                    - Hardware flow control disabled (RTS and CTS signals)
                                    - Receive and transmit enabled
         */
        USART_InitStructure.USART_BaudRate = speed;
        USART_InitStructure.USART_WordLength = USART_WordLength_8b;
        USART_InitStructure.USART_StopBits = USART_StopBits_1;
        USART_InitStructure.USART_Parity = USART_Parity_No;
        USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
        USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

        /* Enable GPIO clock */
        RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);

        /* Enable USART clock */
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);

        /* Connect PXx to USARTx_Tx - PIN A14 - TX*/
        GPIO_PinAFConfig(GPIOA, GPIO_PinSource14, GPIO_AF_7);

        /* Connect PXx to USARTx_Rx - PIN A15 - RX*/
        GPIO_PinAFConfig(GPIOA, GPIO_PinSource15, GPIO_AF_7);

        /* Configure USART Tx as alternate function push-pull */
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
        GPIO_Init(GPIOA, &GPIO_InitStructure);

        /* Configure USART Rx as alternate function push-pull */
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
        GPIO_Init(GPIOA, &GPIO_InitStructure);

        /* USART configuration */
        USART_Init(USART2, &USART_InitStructure);

        /* Enable USART */
        USART_Cmd(USART2, ENABLE);
}


char USART2_GetC(void){
        return USART_ReceiveData(USART2);
}


void USART2_PutC(char ch){
        /* Put character on the serial line */
        USART2->TDR = (ch & (uint16_t)0x01FF);

        /* Loop until transmit data register is empty */
        while ((USART2->ISR & USART_FLAG_TXE) == (uint16_t) RESET);
}


void USART2_PutS(char *pcString){
        int i = 0;
```

```
        while (pcString[i] != 0){
                USART2_PutC(pcString[i]);
                i++;
        }
}


void USART2_IRQHandler(void){
        static int rx2_index = 0;
        rx2_index++;
/*

*/
}

int USART2_kbhit(void){
        return USART_GetFlagStatus(USART2, USART_FLAG_RXNE);
}
```

APÊNDICE B – CÓDIGO FONTE DO COMPUTADOR EMBARCADO

Neste apêndice serão apresentados os códigos fonte embarcado no computador de bordo para a realização dos testes experimentais. A versão do Xenomai utilizada neste projeto é a 2.6.3.

B.0.12 CompRT.c

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
#include <math.h>

#include<termios.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<fcntl.h>
#include<errno.h>
#include <signal.h>
#include <sys/mman.h>
#include<sys/time.h>

#include <cv.h>
#include <highgui.h>
#include <opencv2/legacy/legacy.hpp>

#include <native/task.h>
#include <native/heap.h>
#include <native/mutex.h>
#include <native/timer.h>

#define time_sample 0.02
/*********************************************************/
/*                                    Variaveis Globais
*/
/*********************************************************/
RT_TASK ControllerTask, OpticalFlowTask, IMUTask;
RT_HEAP ControlOptFlow_sharedmem, ControlIMU_sharedmem;
```

```
RT_MUTEX ControlOptFlow_mutex, ControlIMU_mutex;

void* ControlOptFlow_memory;
void* ControlIMU_memory;

FILE *of_fd;
FILE *imu_fd;
FILE *control_fd;
/*********************************************************/
/*                    Prototipo das Tarefas de Tempo Real                    */
/*********************************************************/
void ControllerThread(void *arg);
void OpticalFlowThread(void *arg);
void IMUThread(void *arg);


void catch_signal(int sig){

        fclose(of_fd);
        fclose(imu_fd);
        fclose(control_fd);

        rt_task_delete(&IMUTask);
        rt_task_delete(&ControllerTask);
        rt_task_delete(&OpticalFlowTask);
}



/*********************************************************/
/*                                                    MAIN
*/
/*********************************************************/
int main(int argc, char* argv[]){

        signal(SIGTERM, catch_signal);
        signal(SIGINT, catch_signal);

        of_fd = fopen("RTOpticalFlow.txt", "w");
        imu_fd = fopen("RTIMU.txt", "w");
        control_fd = fopen("RTControl.txt", "w");

        mlockall(MCL_CURRENT|MCL_FUTURE);

        rt_task_create(&IMUTask, "LeituraIMU", 1024, 99, T_FPU|T_CPU(0)|T_CPU(1));
        rt_task_create(&ControllerTask, "Controlador", 1024, 98, T_FPU|T_CPU(0)|T_CPU(1));
        rt_task_create(&OpticalFlowTask, "FluxoOptico", 0XFFFF, 99, T_FPU|T_CPU(2)|T_CPU(3));

        rt_heap_create(&ControlOptFlow_sharedmem, "ControlOptFlowHeap", 2*sizeof(float), H_SHARED);
        rt_heap_bind(&ControlOptFlow_sharedmem, "ControlOptFlowShm", TM_NONBLOCK);
        rt_heap_alloc(&ControlOptFlow_sharedmem, 0, TM_NONBLOCK, &ControlOptFlow_memory);

        rt_heap_create(&ControlIMU_sharedmem,  "ControlIMUHeap",6*sizeof(float), H_SHARED);
        rt_heap_bind(&ControlIMU_sharedmem, "ControlIMUShm", TM_NONBLOCK);
        rt_heap_alloc(&ControlIMU_sharedmem, 0, TM_NONBLOCK, &ControlIMU_memory);

        rt_mutex_create(&ControlOptFlow_mutex, "ControlOptFlowMutex");
        rt_mutex_create(&ControlIMU_mutex, "ControlIMUMutex");
```

```
        rt_task_start(&ControllerTask, &ControllerThread, NULL);
        rt_task_start(&OpticalFlowTask, &OpticalFlowThread, NULL);
        rt_task_start(&IMUTask, &IMUThread, NULL);

        pause();

        return 0;
}



/*********************************************************/
/*                                    Tarefas de Tempo Real
*/
/*********************************************************/

/* Tarefa do controlador*/
void ControllerThread(void *arg){

        struct timeval start, end;

        float* optflow_ptr;
        float* imu_ptr;
        float Roll=0, Pitch=0, Yaw=0, Altitude=0, Velx=0, Vely=0;
        float VelxOF=0, VelyOF=0;

        float ZRef = 2;
        float ZErro, ZDerivErro, ZErroAnt;
        float YawRef = 45;
        float YawErro, YawDerivErro, YawErroAnt;
        int Ch1, Ch4;
        float Kdz1=1, Kdz2=1, Kpz1=1, Kpz2=1;
        float Kdy1=1, Kdy2=1, Kpy1=1, Kpy2=1;

        rt_task_set_periodic(NULL, TM_NOW, 20000000);

        while(1){
                gettimeofday(&start, NULL);

                optflow_ptr = (float*)ControlOptFlow_memory;
                rt_mutex_acquire(&ControlOptFlow_mutex, TM_INFINITE);
                VelxOF = optflow_ptr[0];
                VelyOF = optflow_ptr[1];
                rt_mutex_release(&ControlOptFlow_mutex);

                imu_ptr = (float*)ControlIMU_memory;
                rt_mutex_acquire(&ControlIMU_mutex, TM_INFINITE);
                Roll = imu_ptr[0];
                Pitch = imu_ptr[1];
                Yaw = imu_ptr[2];
                Altitude = imu_ptr[3];
                Velx = imu_ptr[4];
                Vely = imu_ptr[5];
                rt_mutex_release(&ControlIMU_mutex);

                ZErro = ZRef - Altitude;
                ZDerivErro = (ZErro - ZErroAnt)/time_sample;
```

```
                ZErroAnt = ZErro;

                YawErro = YawRef - Yaw;
                YawDerivErro = (YawErro - YawErroAnt)/time_sample;
                YawErroAnt = YawErro;

                Ch1 = Kdz1*tanh(Kdz2*ZDerivErro) + Kpz1*tanh(Kpz2*ZErro);

                Ch4 = Kdy1*tanh(Kdy2*YawDerivErro) + Kpy1*tanh(Kpy2*YawErro);

                gettimeofday(&end, NULL);

                fprintf(control_fd,"%f\n",(((double)end.tv_sec-(double)start.tv_sec)*1000 +
                                ((double)end.tv_usec-(double)start.tv_usec)/1000));

                rt_task_wait_period(NULL);

        }
}



/*********************************************************/
/*********************************************************/

/* Tarefa do Fluxo Optico*/
void OpticalFlowThread(void *arg){

        struct timeval start, end;

        float *control_ptr;

        // ponteiro para captura de quadros de uma camera
        CvCapture *cam_video = cvCaptureFromCAM(0);
        cvQueryFrame(cam_video);

        // define o tamanho do quadro para a alocacao das imagens
        CvSize frame_tam;
        frame_tam.height = (int) cvGetCaptureProperty(cam_video, CV_CAP_PROP_FRAME_HEIGHT);
        frame_tam.width = (int) cvGetCaptureProperty(cam_video, CV_CAP_PROP_FRAME_WIDTH);

        // estrutura para obter quadro da camera
        IplImage* frame = NULL;

        // estruturas para armazenar o quadro anterior e o atual
        IplImage* frame1 = cvCreateImage(frame_tam, IPL_DEPTH_8U, 1); // quadro anterior
        IplImage* frame2 = cvCreateImage(frame_tam, IPL_DEPTH_8U, 1);  // quadro atual

        // estruturas auxiliares e de piramide
        IplImage* velx = cvCreateImage(frame_tam, IPL_DEPTH_32F, 1);
        IplImage* vely = cvCreateImage(frame_tam, IPL_DEPTH_32F, 1);

        CvScalar mediax, mediay;

        rt_task_set_periodic(NULL, TM_NOW, 300000000);

        while(1){
                gettimeofday(&start, NULL);
```

```
                frame = cvQueryFrame(cam_video); // captura o primeiro quadro
                cvConvertImage(frame, frame1, 0);

                frame = cvQueryFrame(cam_video); // captura o segundo quadro
                cvConvertImage(frame, frame2, 0);

                cvCalcOpticalFlowLK(
                        frame1,
                        frame2,
                        cvSize(3,3),
                        velx,
                        vely
                        );

                mediax = cvAvg(velx, NULL);
                mediay = cvAvg(vely, NULL);

                control_ptr = (float*)ControlOptFlow_memory;

                rt_mutex_acquire(&ControlOptFlow_mutex, TM_INFINITE);
                control_ptr[0] = mediax.val[0];
                control_ptr[1] = mediay.val[0];
                rt_mutex_release(&ControlOptFlow_mutex);

                gettimeofday(&end, NULL);
                fprintf(of_fd,"%f\n",(((double)end.tv_sec-(double)start.tv_sec)*1000 +
                                ((double)end.tv_usec-(double)start.tv_usec)/1000));

                rt_task_wait_period(NULL);
        }

}



/*********************************************************/
/*********************************************************/
/* Tarefa da Leitura da IMU*/
void IMUThread(void *arg){

        struct timeval start, end;

        float *control_ptr;
        float roll=0, pitch=0, yaw=0, altitude=0, velx=0, vely=0;
        struct termios uart1;
        int uart_fd;
        cfsetospeed(&uart1, B115200);
        cfsetispeed(&uart1, B115200);

        uart1.c_cflag = (uart1.c_cflag & ~CSIZE) | CS8;
        uart1.c_iflag &= ~IGNBRK;
        uart1.c_lflag = ~(ICANON | ISIG);
        uart1.c_oflag = 0;
        uart1.c_cc[VMIN] = 0;
        uart1.c_cc[VTIME]=0;
        uart1.c_iflag &= ~(IXON | IXOFF | IXANY);
        uart1.c_cflag |= (CLOCAL | CREAD);
```

```
        uart1.c_cflag &= ~(PARENB | PARODD);
        uart1.c_cflag |=0;
        uart1.c_cflag &= ~CSTOPB;
        uart1.c_cflag &= ~CRTSCTS;

        uart_fd = open("/dev/ttymxc0", O_RDWR);
        tcflush(uart_fd, TCIFLUSH);
        tcsetattr(uart_fd, TCSANOW, &uart1);

        rt_task_set_periodic(NULL, TM_NOW, 20000000);

        while(1){
                gettimeofday(&start, NULL);

                read(uart_fd, &roll, sizeof(float));
                read(uart_fd, &pitch, sizeof(float));
                read(uart_fd, &yaw, sizeof(float));
                read(uart_fd, &altitude, sizeof(float));
                read(uart_fd, &velx, sizeof(float));
                read(uart_fd, &vely, sizeof(float));

                control_ptr = (float*)ControlIMU_memory;
                rt_mutex_acquire(&ControlIMU_mutex, TM_INFINITE);
                control_ptr[0] = roll;
                control_ptr[1] = pitch;
                control_ptr[2] = yaw;
                control_ptr[3] = altitude;
                control_ptr[4] = velx;
                control_ptr[5] = vely;
                rt_mutex_release(&ControlIMU_mutex);

                gettimeofday(&end, NULL);
                fprintf(imu_fd,"%f\n",(((double)end.tv_sec-(double)start.tv_sec)*1000 +
                                ((double)end.tv_usec-(double)start.tv_usec)/1000));

                rt_task_wait_period(NULL);
        }

}
```