Universidade Federal de Juiz de Fora

Programa de Pós-Graduação em Engenharia Elétrica

João Pedro Carvalho de Souza

Pouso Autônomo de VANTs baseado em Rede Neural Artificial Supervisionada por Lógica Fuzzy

Juiz de Fora

2018

João Pedro Carvalho de Souza

Pouso Autônomo de VANTs baseado em Rede Neural Artificial Supervisionada por Lógica Fuzzy

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Juiz de Fora, como requisito para obtenção do título de Mestre em Engenharia Elétrica

Orientador: Prof. André Luís Marques Marcato, D.Sc.

Juiz de Fora

2018

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Carvalho, João Pedro.
Pouso Autônomo de VANTs baseado em Rede Neural Artificial Supervisionada por Lógica Fuzzy
/ João Pedro Carvalho de Souza. – 2018.
137 f. : il.

Orientador: Prof. André Luís Marques Marcato, D.Sc.
Dissertação – Universidade Federal de Juiz de Fora, Programa de Pós-Graduação em Engenharia Elétrica. , 2018.

1. VANT 2. RNA 3. Fuzzy 4. Pouso Autônomo por Visão 5. Sistema Embarcado

João Pedro Carvalho de Souza

Pouso Autônomo de VANTs baseado em Rede Neural Artificial Supervisionada por Lógica Fuzzy

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Juiz de Fora, como requisito para obtenção do título de Mestre em Engenharia Elétrica

Aprovada em: 08/02/2018

BANCA EXAMINADORA

Prof. André Luís Marques Marcato, D.Sc. - Orientador
Universidade Federal de Juiz de Fora

Prof. Bruno Henrique Groenner Barbosa, D.Sc.
Universidade Federal de Lavras

Prof. Eduardo Pestana de Aguiar, D.Sc.
Universidade Federal de Juiz de Fora

À Natureza
A Deus
Ao meu pai
À minha mãe
À minha namorada
Aos meus familiares
Aos meus amigos

Agradecimentos

Sou grato a Deus, à minha fé e à Natureza de me proporcionarem saúde e sanidade.

Agradeço aos meus familiares, principalmente a meus pais, João Carlos e Celimar, que depositaram em mim a confiança, investiram em meus estudos e me deram suporte nessa caminhada. Sem eles eu não estaria aqui. Agradeço a minha namorada, Lizandra, que conviveu comigo nessa jornada de ausências, foi compreensível e me trouxe serenidade.

Agradeço ao meu orientador, André Marcato, que depositou em mim a confiança e oportunidade de me unir a grupos de pesquisas fantásticos. Me proporcionou condições de trabalho e me guiou no caminho das pesquisas. Sou grato a oportunidade dada, além das propostas, discussões, críticas e sugestões dadas ao meu trabalho.

Agradeço aos meus colegas do laboratório Litel, alunos e professores, que sempre que possível me auxiliavam. Um agradecimento especial vai a Alexandre Menezes e Marco Aurélio Jucá, que foram meus principais companheiros na jornada de pesquisas, estudos e testes. Opniões e conselhos foram dados, além de suporte em experimentos práticos, que eram impossíveis de serem executados sozinho. Debaixo de chuva e sol estavam lá comigo.

T*he intelligence is the capacity of adaptation.*
(Stephen Hawking)

RESUMO

Os Veículos Aéreos Não Tripulados (VANTs) demonstram-se como tecnologia promissora visto sua alta aplicabilidade e custos reduzidos. Assim, esses veículos são estudados por engenheiros e pesquisadores que visam, além de aplicá-los, melhorar seu desempenho, segurança e torná-los autônomos e de fácil interação.

Etapas de voos como decolagem, subida, cruzeiro, descida e aterrissagem são objetos de estudos para melhoria de perfomance dessas aeronaves. A aterrissagem é uma etapa delicada para o veículo, cuja operação inadequada pode resultar em acidentes e perdas. Com esse intuito, a presente dissertação propõe uma técnica para o pouso autônomo/assistido de VANTs embarcado ao veículo, sem a necessidade de estações base de processamento. Para o sensoriamento, é utilizado o algoritmo de visão computacional denominado *Ar Track Alvar* para identificação de marcadores artificiais, utilizados como local de pouso. A configuração do local de pouso visa a aplicação da aterrissagem em alturas mais elevadas, pois são utilizados diferentes marcadores artificiais para a sua composição.

O algoritmo de pouso também é uma contribuição do presente trabalho, no qual a execução é realizada por uma Rede Neural Artificial (RNA), do tipo *Multilayer Perceptron*, cujo treinamento é supervisionado por uma lógica *fuzzy* que utiliza a inferência Mamdani. A utilização do *fuzzy* torna-se viável devido a sua característica não determinística, sendo menos susceptível a ruídos de sensoriamento. Outro ponto importante é a não necessidade de se ajustar ganhos para o procedimento para cada aeronave usada, tornando-se o processo perigoso e trabalhoso. Esse revés é visto em controladores clássicos como o PID. Apesar das vantagens da lógica *fuzzy*, essa se mostra computacionalmente custosa devido a seu processo Mamdani. Como uma RNA treinada é um conjunto de operações matriciais, é proposto o treinamento da mesma supervisionada pelo algoritmo *fuzzy* já funcional. Assim se reduz a complexidade computacional do algoritmo embarcado facilitando o processsamento de imagem.

O *firmware* de aterrissagem proposto é desenvolvido sobre o *framework Robot Operation System* (ROS) e focado para replicação em dispositivos reais e embarcados.

Os resultados são apresentados em *Software in the Loop* (SITL) e em experimentos reais em ambientes externos para locais de pouso estáticos e dinâmicos. A comparação de desempenhos dos algoritmos é mostrada. O desempenho atingido foi satisfatório e a capacidade da RNA, além da redução da complexidade computacional, foram verificadas.

Palavras-chave: VANT, RNA, Fuzzy, Pouso Autônomo por Visão, Sistema Embarcado.

ABSTRACT

Unmanned Aerial Vehicles (UAVs) are shown as promising technology because of their high applicability and low costs. Thus, these vehicles are engineers and researchers studies targets that aim, in addition to applying them, to improve their performance, safety and make them autonomous and easily interaction.

Flight stages such as takeoff, ascent, cruise, descent and landing are objects of studies to improve these aircrafts performance. Landing is a delicate stage for the vehicle, whose improper operation can result in accidents and losses. With this purpose, the present dissertation proposes a technique for the UAVs autonomous/assisted landing onboard the vehicle, without the use of ground control stations. As a sensing, the Ar Track Alvar computational vision algorithm is used to identify artificial markers used as a landing site. The landing site configuration aims the application of landing at higher altitudes, as different artificial markers are used for its composition.

The landing algorithm is also a contribution of the present work, in which the execution is performed by an Multilayer Perceptron Artificial Neural Network (ANN) whose training is supervised by a logic fuzzy that uses the Mamdani inference. The use of fuzzy becomes viable due to non-deterministic characteristic and is less susceptible to sensing noise. Another important point is the no need to adjust gains for the procedure for each aircraft used, making the process dangerous and laborious. This setback is seen in classic controllers like the PID. Despite the advantages of fuzzy logic, this is computationally costly due to its Mamdani process. As a trained RNA is a set of matrix operations, it is proposed to train it supervised by the already functional fuzzy algorithm. This reduces the computational complexity of the embedded algorithm, facilitating image processing.

The proposed landing firmware is developed on the Robot Operation System (ROS) and is focused on replication on real and embedded devices.

The results are presented in Software in the Loop (SITL) and in real experiments at outdoor environments for static and dynamic landing spots. Comparison of algorithm performances is also shown. The performance was satisfactory and the RNA capacity and computational complexity reduction were verified.

Key-words: UAV, ANN, Fuzzy, Autonomous Vision Landing, Embedded System

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

ABREVIATURAS E SIGLAS

VANT Veiculo Aéreo Não Tripulado

RPA *Remotely-Piloted Aircraft*

ANAC Agência Nacional de Aviação Civil

VTOL *Vertical Takeoff and Landing*

LOCUST *LOw-Cost Unmanned Aerial Vehicle Swarming Technology*

UAS *Unmanned Aircraft System*

FAB Força Aérea Brasileira

ROS *Robot Operation System*

FCU *Flight Control Unit*

SLAM *Simultaneous Localization and Mapping*

RNA Rede Neural Artificial

GPS *Global Positioning System*

SITL *Software in the Loop*

ORB *Object Request Broker*

SO Sistema Operacional

MAVLINK *Micro Air Vehicle Communication Protocol*

XML *eXtensible Markup Language*

TCP *Transmission Control Protocol*

IP *Internet Protocol*

UDP *User Datagram Protocol*

TCPROS *ROS Transmission Control Protocol*

UDPROS *ROS User Datagram Protocol*

IDE *Integrated Development Environment*

CAN *Controller Area Network*

MAVROS *MAVLink to ROS gateway with UDP proxy for Ground Control Station*

GCS *Ground Control Station*

FTP *File Transfer Protocol*

OBC *Onboard Computer*

YAML *YAML Ain't Markup Language*

RTOS *Real-time Operating System*

PWM *Pulse-width Modulation*

IMU *Inertial Measurement Unit*

UTM *Universal Transverse Mercator*

HUD *Head Up Display*

LISTA DE SÍMBOLOS

$\mathbf{U}$	Universo de discurso.
Θ	Conjunto vazio
$\mu_A(u)$	Função de pertinência do elemento $u \in \mathbf{U}$ para o subconjunto $A \in \mathbf{U}$.
$\mu_{\mathbf{U}}$	Função pertinência de $\mathbf{U}$.
μ_Θ	Função pertinência de Θ.
S	Área de uma figura geométrica.
$f(\cdot)$	Função de ativação.
$\sum$	Integrador de um neurônio artificial.
n	Número da iteração do processo de treinamento.
k	Número da camada de uma RNA.
K	Número máximo de camadas de uma RNA.
i	Número do neurônio dado uma camada k.
$I(k)$	Número máximo de neurônios na camada k.
j	Número do neurônio dado uma camada k-1.
x_i	Entrada do conjunto de treinamento. Entrada para o neurônio i da camada k=1.
d_i	Saída desejada do conjunto de treinamento. Saída desejada para o neurônio i da camada k=K.
$o_i(n)$	Saída estimada pela RNA durante a iteração n, para o neurônio i da camada k=K.
$y_i^k(n)$	Saída de um neurônio i na camada k na iteração n.
$v_i^k(n)$	Saída de um integrador de um neurônio i na camada k na iteração n
$w_{i,j}^k(n)$	Peso sináptico entre o neurônio j e i na camada k na iteração n.
b_i	Valor de *bias* de um nerônio i.
$EQM(n)$	Erro quadrático médio da iteração n.
$\delta_i^k(n)$	Gradiente local do neurônio i para camada k na iteração n.

α	Coeficiente de aprendizagem.
$\mathbf{t}$	Vetor de translação tridimensional.
$\mathbf{R_x}$	Matriz de rotação em torno do eixo x.
$\mathbf{R_y}$	Matriz de rotação em torno do eixo y.
$\mathbf{R_z}$	Matriz de rotação em torno do eixo z.
$^{\mathbf{I}}\mathbf{T_V}$	Matriz de transformação homogênea do VANT em relação ao *frame* Inercial.
$^{\mathbf{V}}\mathbf{T_C}$	Matriz de transformação homogênea da Câmera em relação ao *frame* do VANT.
$^{\mathbf{C}}\mathbf{P_M}$	Posição do marcador (local de pouso) em relação ao *frame* da Câmera.
$^{\mathbf{V}}\mathbf{P_M}$	Posição do marcador (local de pouso) em relação ao *frame* do VANT.
$^{\mathbf{V}}\mathbf{A}$	Atuação no *frame* do VANT.
$^{\mathbf{I}}\mathbf{A}$	Atuação no *frame* Inercial.
$^{\mathbf{M}}\mathbf{T_{id}}$	Matriz de transformação homogênea do marcador de ID genérico em relação ao marcador central de pouso.
$^{\mathbf{C}}\mathbf{P_{id}}$	Posição do marcador de ID genérico em relação ao *frame* da Câmera.
ΔX	Distância relativa do VANT para o local de pouso no eixo X.
ΔY	Distância relativa do VANT para o local de pouso no eixo Y.
V_X	Velocidade do VANT no eixo X.
V_Y	Velocidade do VANT no eixo Y.
V_Z	Velocidade do VANT no eixo Z.

SUMÁRIO

1 INTRODUÇÃO

Os Veículos Aéreos Não Tripulados (VANTs) se destacam cada vez mais na sociedade moderna, principalmente, devido a alta aplicabilidade e preço reduzido desse tipo de equipamento em comparação a outros veículos aéreos. Dado seu futuro promissor, os VANTs são alvos de estudos de engenheiros e pesquisadores que visam, além de aplicá-los, melhorar seu desempenho, segurança e torná-los autônomos e de fácil interação.

Por ser uma tecnologia recente muito se confunde sobre a definição destas aeronaves. No meio popular são denominados de *Drones*, Zangão em inglês, devido ao seu som característico. Contudo, quando esse tipo de veículo é utilizado para fins além do de lazer e recreação, ou seja, carregam uma carga útil, são caracterizados como VANTs. Os VANTs podem ser controlados/pilotados remotamente, sendo denominados de *Remotely-Piloted Aircraft* (RPA), ou voarem de maneira autônoma. Outra consequência dessa recente tecnologia é que muitos países ainda possuem empecilhos ou sequer estão preparados legalmente para o seu uso. Por exemplo, no Brasil, a Agência Nacional de Aviação Civil (ANAC), o órgão regulamentador e fiscalizador que monitora e define as regras para esse tipo de aeronave, estabeleceu a legislação do tema apenas no ano de 2015 (1). As regras definem e distinguem os *Drones* dos VANTs/RPAs e regulamentam seu uso para serviço, pesquisa e recreação (2). O departamento de controle do espaço aéreo, sob o comando da aeronáutica brasileira, emite certificados de voo e operação a interessados em utilização de RPAs, além de disponibilizar instruções e fornecer guias de utilização (3).

Estes veículos aéreos podem possuir diversas formas e são divididos em 3 classes principais: asas fixas, asas giratórias e os *Vertical Takeoff and Landing* (VTOL). Os VANTs de asas fixas são similares a aviões e utilizados em missões de longo alcance dado sua capacidade de percorrer grandes distâncias com consumo reduzido de energia. A velocidade também é seu ponto forte. Contudo, esse tipo de forma necessita de áreas grandes para decolagem e pouso, e não é capaz de realizar manobras mais delicadas. Os VANTs de asas giratórias são veículos mais lentos, porém mais sensíveis e precisos, úteis para aplicações mais sutis. Esses modelos possuem a capacidade de pouso e decolagem em áreas reduzidas, todavia não atingem velocidades elevadas nem possuem grande autonomia de voo. Esse tipo pode ser baseado em helicópteros ou n-rotores (como quadrotores, hexatores e afins). Já o VTOL é o modelo proposto capaz de integrar as funcionalidades de ambas formas anteriormente citadas. Como próprio nome diz, o VTOL é capaz de realizar decolagem e pouso verticais e voar de maneira rápida e econômica, como um avião. A Figura 1 elucida as classes de VANTs discutidas.

Devido a alta aplicabilidade perante custos reduzidos, os VANTs tornaram-se objetos de estudos de pesquisadores e engenheiros ao redor do globo. Além de melhoria do desempenho, segurança e confiabilidade, o foco das pesquisas é a utilização destes

Figura 1 – Classes de VANTs

Fonte: Elaborada pelo autor

tipos de veículos em aplicações no meio militar, privado e acadêmico. No âmbito militar, estes tipos de aeronaves são extensamente utilizados para o monitoramento de fronteiras e regiões de conflitos. Como exemplo, tem-se o modelo *Predator RQ-1* da Força Aérea dos Estados Unidos, cujo primeiro voo foi realizado em 1994. A proposta era a aplicação de reconhecimento e vigilância em uma aeronave robusta o suficiente para longos voos e grandes distâncias, excluindo a necessidade de humanos a bordo (4). As outras forças armadas dos Estados Unidos, como os *Navys*, *U.S Army* (5) e os *U.S Marine* desenvolvem programas independentes desse recurso desde o ano de 2001, pós atentado terrorista de 11 de setembro (4). Os RPAs propostos nesses projetos são utilizados largamente em operações de suporte às forças terrestres e monitoramento nas ações no Oriente Médio, tornando-se um tema polêmico perante a comunidade mundial. Um exemplo de pesquisa realizada pelas forças armadas estado-unidenses é o *LOw-Cost Unmanned Aerial Vehicle Swarming Technology* (LOCUST), que visa a múltipla operação e iteração de VANTs (6). Outra atuação é o estudo massivo destes veículos com os *Unmanned Systems*, sendo chamados de *Unmanned Aircraft System* (UAS). Os UAS são sistemas inteligentes, autônomos e embarcados com sensoriamento suficiente para execução de missão sem a intervenção humana (7).

No Brasil, o esquadrão Hórus, unidade da Força Aérea Brasileira (FAB), é o responsável pela operação de VANTs que iniciou no ano de 2011 (8). Os modelos Hermes

450 e Hermes 900, presentes no esquadrão, foram utilizados para vigilâncias aéreas de grandes eventos, como: copa das confederações de 2013 (9), copa do mundo de 2014 (10) e os jogos olímpicos de 2016 (11). As aeronaves também são empregadas para monitoramento de fronteiras, como abordado em (12). Em associação com a polícia federal, a FAB empregou seus VANTs na fronteira com o Paraguai. O foco é a apreensão de drogas e contrabando de produtos.

As forças armadas investem constantemente neste tipo de aeronave, contudo grupos e empresas privadas também vislumbram a utilização dos VANTs. Em 2013, a empresa norte-americana *Amazon* anunciou o início de seu projeto denominado *Amazon Prime Air*, Figura 2. Esse serviço propõe a entrega segura de produtos utilizando VANTs autônomos. Com a capacidade de carga de até $2,27$ kg e raio de atuação de 16 km, a empresa investe em suas pesquisas de entregas por *drones* para atender esses requisitos, já que 86% de suas vendas e entregas atendem esses critérios. As sedes de pesquisas da *Amazon* estão localizados nos Estados Unidos, Reino Unido, Áustria e Israel e indicam a aposta realizada pela empresa. A companhia afirma em (13) o potencial da realização de entregas de forma segura e eficiente para milhões de clientes utilizando veículos aéreos não tripulados. Segundo (13), estudos envolvem a operação, métodos de sensoriamento para segurança, eficiência e desenvolvimento de modelos de aeronaves para a realização das entregas. Experimentos do serviço proposto foram realizados em dezembro de 2016 no Reino Unido, cuja aplicação experimental foi realizada por um *drone*. O produto foi entregue utilizando visão computacional que identificava o local de entrega através de um marcador artificial inserido pelo cliente em sua residência. Esse experimento se assemelha à proposta apresentada nesta dissertação para detecção do local de pouso da aeronave, melhor abordado na Seção 1.2.

Figura 2 – VANTs do projeto *Amazon Prime Air*

Fonte: amazon.com/primeair

A *Amazon* em conjunto com outras empresas, ressaltando a falta de legislação sobre o assunto, propõem em dois documentos a reestruturação do espaço aéreo de baixa altitude (14) e o emprego de UAS em áreas urbanas (15), para que, em um futuro próximo, os *drones* possam ser utilizados normalmente na sociedade.

O *Facebook* também está desenvolvendo o seu projeto de VANTs, denominado Aquila. A proposta é a utilização destes veículos para permitir o acesso à internet de qualidade para regiões remotas. Segundo (16), o Aquila deverá manter um tempo de voo de 90 dias, a uma altura de aproximadamente 184 km para emitir sinais de internet a uma área de 96km de diâmetro abaixo do veículo. Vários VANTs estariam interligados para abranger uma grande área de cobertura. Testes com desempenho de voo de 96 minutos foram realizados no Arizona (EUA) em 28 de Junho de 2016 e os desafios encontrados pela equipe são grandes, principalmente no consumo de energia. A proposta é a utilização de painéis solares que acabam por prejudicar no peso da estrutura, que exigirá maior quantidade de baterias. Em (17) ressalta-se que o uso das aeronaves movidas por energia solar para o fornecimento de internet seriam mais baratas em comparação a infraestrutura necessária para internet por *links* de fibra ou micro-ondas.

Várias empresas surgem visando atingir o recém mercado de VANTs, como a DJI (18) e a 3DR (19). Essas companhias focam na produção e desenvolvimento de aeronaves e propõem modelos capazes de realizar missões autônomas e assistidas para diferentes propósitos. Como exemplo tem-se o *Drone Phantom 4 Pro* da DJI (Figura 3) que, embarcado com diferentes redundâncias sensoriais, consegue realizar voos assistidos que impedem o usuário de se chocar contra obstáculos, dificulta quedas, além de reconhecer um usuário alvo para realização de missões de perseguição. Isso mostra o investimento realizado por esses grupos para tornar os *Drones* mais acessíveis a usuários comuns, reduzindo as dificuldades inerentes às aeronaves na pilotagem e controle.

Figura 3 – Phantom 4 Pro

Fonte: dji.com

No mesmo contexto, objetivando melhorias de desempenho e aplicações, e percebendo o potencial e a vasta área de pesquisa que os veículos aéreos não tripulados proporcionam, a comunidade acadêmica se engaja na pesquisa e desenvolvimento de técnicas para melhoria dos VANTs. Modelagem e controladores são propostos para diferentes estruturas e formas de aeronaves, com o objetivo de viabilizar o seu uso. Para o caso de VANTs com asas giratórias, essas aeronaves atuam de acordo com uma referência de

atitude (ângulos de *roll*, *pitch* e *yaw*) e de posição, com resposta baseada em acelerações em seus motores. Como exemplo dessas pesquisas tem-se (20), cuja proposta é a modelagem de um quadrotor em *hover condition* onde considerações inicias são estabelecidas para simplificação do modelo. Contudo, o modelo não é suficientemente confiável para aplicações extremas de voo, e a continuação do trabalho é apresentado em (21). Os autores de (21) incluem técnicas de otimização para determinação de trajetórias suaves e, assim, gerar as referências de posicionamento para os controladores. Outros trabalhos como (22–25) também apresentam técnicas de modelagem, projetos de controladores e ajustes de ganhos para o controle em baixo nível da aeronave.

Muita das técnicas existentes, incluindo as anteriormente citadas, podem ser consideradas trabalhosas de serem implementadas rotineiramente, ainda mais para aplicações de camadas mais altas que não visam, em sua grande maioria, lidar com problemas de controle ou funcionamento da própria aeronave. Desta forma, grupos de pequisas propõem novos *firmwares*, *frameworks* e *hardwares* que atendam requisitos necessários para uma plataforma de voo e, assim, outras pesquisas possam dar continuidade a um trabalho mais complexo. Como exemplo, tem-se o *firmware* de controle aéreo proposto em(26). Denominado *PX4*, esse *firmware* é capaz de realizar o controle de diversos tipos de *Drones*. Ele possui embarcado técnicas de controle em diferentes camadas focadas para a utilização em veículos com modelagens distintas. A ideia do trabalho é disponibilizar uma base de desenvolvimento aberto para que possíveis aplicações e melhorias sejam continuadas. Devido ao *PX4* ser um sistema embarcado em um *Real-time Operating System* (RTOS), os autores defendem a sua utilização perante os outros sistemas de *firmware* concorrentes, como *APM* e *OpenPilot*. O motivo é a capacidade de multiprocessamento do *PX4*, sendo capaz de lidar, eficientemente, com diferentes dados sensoriais e de atuações. Outra característica importante do *PX4* é a facilidade de interação com *frameworks* de robótica, como o *Robot Operation System* (ROS). Para utilização do *PX4*, os autores em (27) propõem a placa controladora de voo, ou FCU, *Pixhawk*. Sendo um *hardware open-source*, a *Pixhawk* é capaz de ser embarcada com um sistema RTOS além de permitir a utilização de diferentes módulos sensoriais. No trabalho (27), além da apresentação da FCU, é proposto a sua utilização para aplicações geodéticas assistidas. Nessas operações, câmeras *stereos* são empregadas para auxiliar o operador a se movimentar com um RPA próximo a locais com grande número de obstáculos e, assim, reconstruir digitalmente faixadas e telhados de prédios, por exemplo. Segundo os autores, o processamento é realizado embarcado a aeronave por um computador companheiro (do inglês (*companion computers*)) utilizando o sistema operacional Linux.

Outro exemplo de operação utilizando o recurso de evasão de obstáculos é usado em (28) para execução de missões autônomas por VANTs. Empregando a *Pixhawk* e câmeras *stereos*, são propostos algoritmos de exploração e navegação para execução de missões de deslocamento *indoor*. Câmeras acopladas a aeronave e apontadas para baixo

estimam sua posição enquanto outras, direcionadas para frente, extraem características do local. Missões de exploração e mapeamento por VANTs utilizando câmeras também são apresentadas em (29). Contudo, a aplicação possui o foco para ambientes externos. São embarcados algoritmos de *Simultaneous Localization and Mapping* (SLAM), odometria visual, *frontier-based exploration* e *Wall-Following Exploration*.

Todas as pesquisas, aqui citadas, demonstram a complexidade para transformar veículos aéreos em verdadeiros robôs. Muitas operações simples podem ser custosas computacionalmente, já que a confiabilidade da operação é um fator importante, evitando custos, perdas e acidentes. Esse fato desafia pesquisadores e engenheiros, que muita das vezes necessitam embarcar o processamento em dispositivos mais baratos e com baixo consumo de energia.

1.1 TRABALHOS CORRELACIONADOS

Como abordado na seção anterior, a utilização de VANTs para aplicações que envolvem missões autônomas e assistidas são bastantes exploradas e complexas. Além da complexidade de algoritmos estimadores de rotas, localizadores, planejadores, sensoriamento e eletrônica, uma missão autônoma ou assistida de um VANT também apresenta dificuldades em etapas de voo. Como relatado em (30), essas etapas são: decolagem, subida, cruzeiro, descida e aterrissagem. Sendo a última citada a mais desafiadora (30). O pouso é uma etapa delicada para o veículo, cuja operação inadequada pode resultar em acidentes e perdas. Além do mais, um controlador humano, ou não, durante um procedimento de pouso pode ser afetado por uma condição extrema sobre a aeronave, como uma rajada de vento, e não ser capaz em tempo reduzido de reagir a tal pertubação. Outro fator crítico é a definição do local de pouso. Muita das vezes a região de pouso possui área reduzida, como edifícios e clareiras em matas, ou se localiza sobre objetos móveis, como carros e barcos. Qualquer erro nessas situações poderiam causar danos e até mesmo a perda do equipamento. Logo, torna-se importante alternativas para tal procedimento que, em associação com outras técnicas elevam o grau de robustez da aterrissagem.

A aterrissagem de aeronaves não tripuladas é o foco de diversos estudos que aspiram a confiabilidade e segurança da operação. Veículos comerciais que possuem o recurso de pouso autônomo, muito das vezes utilizam da fusão sensorial de *Inertial Measurement Unit* (IMU) e *Global Positioning System* (GPS). Contudo, a magnitude de erro dessas leituras pode acarretar em pousos pouco precisos. Na literatura é possível verificar propostas de novos métodos de sensoriamento para melhoria de precisão do posicionamento e, consequentemente, melhorias no desempenho do pouso das aeronaves. Como exemplo tem-se a utilização de sonares (31), *Optical Flow* (32) e sensores de infravermelho (33).

Uma abordagem para realização de aterrissagem de VANTs é a utilização de

algoritmos de visão computacional. Esses algoritmos podem se distinguir em duas classes quanto ao reconhecimento de local de pouso: algoritmos que utilizam marcadores artificias (*markers* ou *tags*) e os que usam marcadores, ou características, naturais (*features*).

Algoritmos de *features* buscam extrair características de um determinado conjunto de imagens e, assim, detectar possíveis regiões para a execução segura do procedimento de pouso da aeronave. Os autores de (34) propõem um algoritmo de detecção e rastreamento de bordas de pistas de pouso. Baseado em imagens de vídeos captadas pelo VANT, o algoritmo tem em vista gerar informações complementares de erro longitudinal e de posicionamento lateral da aeronave em relação à pista. O algoritmo proposto visa a utilização em sistemas embarcados com computadores companheiros, e os testes são executados em *Hardware in the Loop* em associação a dados de GPS e IMU. Já em (32), é proposto um algoritmo para detecção de locais seguros de pouso através de melhorias do algoritmo de *Scale Invariant Feature Transform*, cujo usuário informa pontos de navegação e aterrissagem através de imagens dos locais pré carregadas. No momento da execução do pouso, os autores de (32) também propõem a utilização do *Optical Flow* para análise da superfície abaixo da aeronave. Assim verifica-se o quão uniforme é o solo e se ele é seguro para a aterrissagem. Uma proposta semelhante para análise da área de pouso é abordada em (35). Nela, um mapa de elevação do terreno é estimado através da detecção de formas de objetos encontrados no local. Os dados são extraídos por imagens provindas de câmeras *stereos*, gerando referências para a navegação do VANT. Outro trabalho com foco em extração de *features* para detecção do local de pouso é proposto em (36).

No contexto da detecção de marcadores artificiais, muitos trabalhos propõem o desenvolvimento de novos algoritmos para reconhecimento de padrões e execução de aterrissagem de veículos aéreos autônomos. Contudo, como o que se pode verificar em (37,38), os algoritmos propostos não são aplicados à aterrissagem do veículo, e modelos de controle baseado na visão não são executados. Trabalhos que propõem a aterrissagem de VANTs por detecção de marcadores artificias e realizam tal procedimento são listados na Tabela 1. Nesta tabela estão enumeradas características de cada trabalho, como técnica empregada para execução do pouso, algoritmo de detecção, se estão ou não projetados para sistemas embarcados e para que tipos de locais de pouso os mesmo foram testados. A utilização do *framework* ROS é incluída, já que sua utilização é crescente nas aplicações práticas e de divulgação.

Os autores de (39), baseado no pacote de realidade aumentada do ROS, Aruco Eye, utilizam 4 controladores *fuzzy* para execução do pouso autônomo de VANTs. Um controlador para cada deslocamento linear tridimensional e outro para a orientação em torno do eixo Z, ou *Yaw*. Contudo, essa abordagem de vários controladores simultâneos não é uma alternativa prática e simplificada. Já em (40), o pouso é executado com 2 controladores *fuzzy*, compensações verticais, horizontais e de orientação respectivamente.

Tabela 1 – Trabalhos relacionados

Artigo	Técnica Empregada	Algoritmo de Visão	ROS	Embarcado	Tipos de Alvos
Vision Based Fuzzy Control Autonomous Landing with UAVs: From V-REP to Real Experiments (39)	Fuzzy	Aruco Eye	Sim	Não	Estático
Landing of an Ardrone 2.0 Quadcopter on a Mobile Baseusing Fuzzy Logic (40)	Fuzzy	Ar Track Alvar	Sim	Não	Estático
Vision-based UAV Landing on the Moving Vehicle (41)	Matrix modeling guidance algorithm	Reconhecimento de Cor do Open-CV	Não	Sim	Estático e Dinâmico
Unmanned Ground and Aerial Vehicles in Extended Range Indoor and Outdoor Missions (42)	Finite State Automa	Baseado em Open-CV	Sim	Não	Estático e Dinâmico
Precise quadrotor autonomous landing with SRUKF vision perception (43)	PID	Aruco Eye	Sim	Sim	Estático
Landing Control on a Mobile Platform for Multi-copters using an Omnidirectional Image Sensor (44)	PD	Reconhecimento de Cor	Não	Sim	Dinâmico
Autonomous Landing of a VTOL UAV on a Moving Platform Using Image-based Visual Servoing (45)	Adaptive Sliding Mode Controller	Image-based Visual Servoing	Não	Não	Dinâmico

Fonte: Elaborada pelo autor

O pacote utilizado para o reconhecimento e estimativa do local de pouso é o Ar Track Alvar. Porém dados de aterrissagem e os experimentos não são apresentados. Ambos os trabalhos (39, 40), utilizam a composição dos marcadores para o local de pouso tornando-se uma alternativa interessante, já que tamanhos diferentes são propostos para detecção em níveis distintos de altura do veículo. Vale ressaltar também que os dois trabalhos não utilizam de estação embarcada de processamento para execução de aterrissagem.

O artigo (42) apresenta um sistema de interação entre VANT e robô terrestre. Todo o sistema é processado em uma unidade terrestre. Para identificação do local de pouso é utilizado a biblioteca Open-CV e modificações de *hardware* são propostas, elevando a complexidade do sistema. Já em (45), são propostos um modelo de controle e de visão para a aterrissagem. No entanto, na malha de controle é utilizado o sistema Vicon, tornando inviável a aplicação de aterrissagem em sistemas reais. O trabalho apresentado em (43) propõe o uso de um controle PID em associação ao algoritmo de visão Aruco Eye. Resultados são apresentados em *indoor* e *outdoor* para alvos estáticos em alturas reduzidas. Como trabalho futuro, os autores de (43) visam a aplicação com local de pouso mais robusto.

A proposta de (41) é embarcar um sistema de orientação baseado em reconhecimento de cores do local de pouso. É utilizada câmera com lentes olhos de peixe, ou *fish-eye lens*, que aumenta o campo de visão da detecção. Os resultados apresentados são expostos em condições de alvos dinâmicos e estáticos. Essa mesma técnica também é abordada em (44). O reconhecimento por cores pode ser perigoso para a atuação da aeronave, pois a probabilidade de assimilação com outros objetos inesperados pode ser alta. O processamento é embarcado em ambos trabalhos e não utilizam do *framework* ROS.

1.2 CONTRIBUIÇÕES E OBJETIVO

Em linhas gerais, torna-se necessário distinguir os três objetivos apresentados nesta dissertação. Eles são classificados em objetivos geral e específico, e estão listados abaixo:

- **Objetivo Geral:** Este trabalho propõe uma técnica para o pouso autônomo/assistido de VANTs embarcado ao veículo. A proposta é permitir um método alternativo aos pousos já existentes, tornando o procedimento de aterrissagem mais confiável. Como sensoriamento é utilizado um algoritmo de visão computacional para identificação de marcadores artificiais, utilizados como local de pouso. A configuração do local de pouso visa a aplicação da aterrissagem em alturas mais elevadas, pois são utilizados diferentes marcadores artificiais para a sua composição.

- **Primeiro Objetivo Específico:** O algoritmo de pouso também é uma contribuição do trabalho, no qual a execução é realizada por uma RNA, do tipo *Multilayer Perceptron*, cujo treinamento é supervisionado por uma lógica *fuzzy* que utiliza a inferência Mamdani. O algoritmo *fuzzy*, devido suas características não determinísticas (46), torna-se uma opção válida para o procedimento de pouso pois é menos susceptível a erros cuja aeronave está submetida: constante pertubação (ventos e aerodinâmica) e ruídos (leituras errôneas de sensores). Além disso, a lógica *fuzzy* não possui ajustes de ganhos quando comparados aos controladores clássicos e não necessita de um modelo acurado da planta (47–49). Dessa maneira, a calibração para cada tipo de aeronave utilizada, após o projeto do controle *fuzzy* baseado no conhecimento do especilista, é menos custoso ou nulo. Contudo, a mesma característica que torna o *fuzzy* uma boa opção, o torna computacionalmente custoso. O custo computacional inerente da lógica *fuzzy* corresponde a todo o seu processo de tomada de decisão: fuzzificação, inferência, defuzzificação, variáveis com diferentes funções membros e resposta baseada em graus de pertinências. Assim, para embarcar em uma aeronave uma aplicação de pouso por visão, é de interesse a utilização de algoritmos mais simplificados para que o reconhecimento por visão seja menos prejudicado em seu processamento e, assim, detecte o local de pouso de maneira mais rápida e confiável. Como uma RNA treinada é um conjunto de operações matriciais, é proposto o seu

treinamento supervisionado pelo algoritmo *fuzzy* já funcional. As operações de soma e multiplicação da RNA treinada reduz o tempo e a complexidade computacional de tomada de decisão, e devido as características de aprendizagem da RNA, ela será capaz de absorver o comportamento e inteligência inerentes da lógica *fuzzy*. A modelagem do problema por lógica *fuzzy* é mais simplificada em comparação à RNA, portanto o processo de treinamento proposto abstrai o projeto realizado pelo desenvolvedor e evita possíveis análises errôneas.

- **Segundo Objetivo Específico:** O procedimento de aterrissagem proposto neste trabalho é codificado e estruturado em sistemas embarcados. Em adição, é proposta uma estrutura susceptível à replicação. Essa última característica leva em conta estruturas de *framework*, *firmware* e *hardware* de desenvolvimento aberto utilizado pela comunidade desenvolvedora. Tem-se como exemplo a utilização em conjunto do ROS, do *PX4* e da FCU Pixhawk. São abordados também conceitos inerentes à técnica de *Software in the Loop* em associação ao simulador Gazebo. Esse método de simulação permite verificar todo o sistema proposto emulando-o em um computador convencional.

Quanto aos resultados alcançados por esta dissertação, esses são listados e apresentados abaixo:

- Execução, com sucesso, do procedimento de aterrissagem autônoma utilizando visão computacional em um sistema totalmente embarcado. Assim, verifica-se a não necessidade de estações bases de processamento que delimitaria o alcance de atuação da aeronave, além de aumentar o custo do sistema;
- Resultados de aterrissagem validados em SITL e em ambiente externo para alturas factíveis. Desempenho analisado para alvos estático e em movimento, atingindo os limites delimitados pelo local de pouso;
- Validação da proposta de treinamento da RNA supervisionada pela lógica *fuzzy*. Experimentos e dados quantitativos demonstram que a RNA foi capaz de emular o comportamento de aterrissagem da lógica *fuzzy*, além de reduzir a complexidade computacional embarcada.

1.3 ESTRUTURA E ORGANIZAÇÃO

Este trabalho é dividido em 5 capítulos. O Capítulo 1 apresenta a contextualização do tema, os trabalhos correlacionados, a motivação e o objetivo da dissertação. Como visa a criação de um sistema funcional e operacional pronto para embarcar em sistemas reais, o Capítulo 2 aborda todas as ferramentas utilizadas para a execução deste trabalho.

São abordados o *framework* ROS, o protocolo de comunicação aérea *Micro Air Vehicle Communication Protocol* (MAVLINK), os pacotes MAVROS e Ar Track Alvar e o *firmware* de controle *PX4*. São abordados também conceitos inerentes à Lógica *Fuzzy* e a Redes Neurais Artificiais, utilizados para a aplicação de aterrissagem autônoma. No Capítulo 3 a metodologia proposta para realização do pouso autônomo é discutida e apresentada. Nele são abordados o projeto da lógica *fuzzy*, o procedimento de treinamento da RNA, a estrutura de codificação e integração e a estratégia abordada para o local de pouso. Os resultados são apresentados no Capítulo 4 para alvos estáticos e dinâmicos sobre o SITL e em ambiente externos. Nesse capítulo é apresentado também a perfomance computacional. Finalizando o trabalho, a conclusão é exposta no Capítulo 5.

1.4 PUBLICAÇÕES DECORRENTES

A partir dos desenvolvimentos em laboratório relacionados à esta dissertação, artigos em conferências, revistas e capítulos em livros foram aceitos tanto no âmbito nacional quanto internacional. Esses trabalhos são listados a seguir:

- João Pedro Carvalho, Marco Jucá, Alexandre Menezes, Leonardo Olivi, Alexandre Bessa, and André Marcato. *Lecture Notes in Electrical Engineering. 402ed*, chapter Autonomous UAV Outdoor Flight Controlled by an Embedded System Using Odroid and ROS, pages 423–437. Springer International Publishing, 2017
- Lucas Moraes, Luiz Carmo, Rafael Falci, João Pedro Carvalho, Marco Jucá, Alexandre Menezes, Daniel Discini, Thiago Coelho, Alexandre Bessa, and André Marcato. Autonomous quadrotor for accurate positioning. *IEEE Aerospace and Electronic Systems Magazine*, 2017
- João Pedro Carvalho, Marco Jucá, Alexandre Menezes, Leonardo Olivi, Alexandre Bessa, and André Marcato. Landing an UAV in a dynamical target using fuzzy control and computer vision. In *Congresso Brasileiro de Automática (CBA), 2016*, 2016

Os conhecimentos adquiridos propiciaram demais publicações com o foco em desenvolvimento de técnicas para melhoria de desempenho e aplicações de VANTs. Essas publicações são listadas a baixo:

- Filipe Delgado, João Pedro Carvalho, A. Bessa, and Thiago Coelho. An optical fiber sensor and its application in UAVs for current measurements. *Sensors (Basel)*, 2016
- Moraes L., Campos. R., Carmo L., Moreira L., Carvalho J.P, Teixeira A., Discini. D., Coelho T., Marcato A., and Santos A. Desenvolvimento de um VANT híbrido autô-

nomo para inspeção multiespectral em áreas no entorno de reservatórios hidrelétricos. In *IX Congresso de Inovação Tecnológica em Energia Elétrica - IX CITENEL*, 2016

- Ítalo Alvarenga, Filipe Delgado, João Pedro Carvalho, A. Bessa, and Thiago Coelho. Development of fiber-optic current sensor based on long-period fiber grating for uav applications. In *MOMAG 2016*, 2016

2 FUNDAMENTAÇÃO TÉCNICA

Este capítulo apresenta conceitos relativos às fundamentações técnicas aplicadas na presente dissertação. São abordadas as ferramentas utilizadas para realização do pouso autônomo funcional e replicável em dispositivos reais.

O *framework* ROS é o conjunto de ferramentas e gerenciador de pacotes e rotinas no qual o *firmware* de aterrisagem proposto é fundamentado, por esse motivo a Seção 2.1 é apresentada. O protocolo de comunicação MAVLINK é discutido na Seção 2.2, pois o seu uso é de fundamental importância para troca de informações entre a aeronave e o computador companheiro embarcado.

Toda aeronave possui *firmwares* de controle de baixo nível para que funções básicas do veículo sejam executadas. Assim, é apresentado na Seção 2.3 o *firmware* de controle aéreo *PX4*, utilizado nesta dissertação. Em seguida, são abordados os pacotes ROS: *MAVROS* (Seção 2.4) e *Ar Track Alvar* (Seção 2.5). O primeiro citado corresponde à interface entre o *PX4* e o ROS, ou seja, é a implementação do protocolo MAVLINK à estrutura de codificação da presente dissertação. O *Ar Track Alvar* é o pacote responsável pela detecção do local de pouso.

São apresentados também conceitos inerentes às técnicas de inteligência artificiais discutidas e utilizadas nesta dissertação: a lógica *fuzzy*, do tipo Mamdani, e a Rede Neural Artificial.

2.1 ROBOT OPERATION SYSTEM - ROS

Esta seção discute o *framework* ROS, com breve resumo de sua história, apresentação e funcionamento, para que ao longo da dissertação os conceitos e pacotes utilizados possam ser abordados. A versão do ROS utilizada neste trabalho é a ROS *Indigo Igloo*.

2.1.1 História

A robótica surgiu devido o contínuo avanço das tecnologias de processamentos de sinais, processamento de dados, *hardwares* e programação, assim como a capacidade de desenvolver novas teorias e por em prática as anteriormente formuladas: inteligência computacional, métodos de otimização, controle e afins. Dessa forma, várias linhas de estudo nesta área surgiram ao redor do globo, tornando a pesquisa muito ampla e satisfatória. Todavia, essa expansão não proporcionava facilidade nas integrações (56).

Com intuito de padronizar as pesquisas, disponibilizar ferramentas e criar um sistema robusto, eficaz, confiável e simples, a universidade de *Stanford* começou a desenvolver protótipos de *frameworks* a partir do ano de 2007. Alguns meses depois, a *Willow Garage*,

uma incubadora de projetos na área de robótica, consolidou o projeto em um *framework* denominado *Robot Operation System* (ROS) (57).

2.1.2 ROS

O ROS é uma plataforma de desenvolvimento livre e apresenta um crescimento considerável graças à comunidade de colaboradores e pesquisadores ativos, dentre eles empresas e universidades que aderiram à sua abordagem (57).

Esse *framework* oferece ao desenvolvedor abstração de *hardware* sem impedir a programação em baixo nível, gerenciamento de pacotes e rotinas além de métodos de comunicação entre eles. Apresenta suporte a diversas bibliotecas em conjunto com ferramentas que permitem a codificação de rotinas, estas em C/C++ e/ou Python, sendo possível a concomitância entre ambas durante um processo, graças à abordagem "dividir para conquistar" que o ROS adota.

Com esta abordagem, processos ou *softwares* complexos são divididos em tarefas menores que comunicam-se entre si por meio de uma estrutura e protocolos impostos pelo *framework*. Torna-se possível, então, criar uma única rede envolvendo várias unidades processadoras, denominadas de nós, com aplicações específicas e eficazes. Esta rede pode estar presente em apenas uma única unidade de processamento ou em um conjunto de *hardwares* como computadores, microprocessadores ou robôs, desde que conectados entre si.

Também é possível a reutilização de códigos e nós que abrangem desde sensoriamentos, como algoritmos de visão e localização espacial, a atuadores empregados na robótica.

Consolidado em Linux e em desenvolvimento para *Windows* e outros Sistemas Operacionais (SOs), o ROS proporciona uma maior flexibilidade quanto ao desenvolvimento. Todavia, isto demanda ao desenvolvedor um conhecimento prévio do Sistema Operacional que torna o processo de estudo do ROS um pouco mais lento.

Este capítulo elucida, além da estrutura, os três níveis do ROS: o processo, as ferramentas e a comunidade. Todos os conceitos abordados aqui podem ser verificados em (58), (59) e (60) e baseam-se na distribuição *Indigo Igloo*, abordada na Seção 2.1.3.

2.1.3 Distribuições

O ROS possui diferentes distribuições com o objetivo de permitir que desenvolvedores trabalhem sobre um conjunto de códigos e bases estáveis e atualizadas (61). Cada distribuição é um conjunto de pacotes e recursos de uma determinada versão do *framework* que visa facilitar a sua instalação. Além disso, as distribuições acrescentam funcionalidades novas e compatibilidade com novas versões de sistemas operacionais.

As distribuições do ROS acompanham as atualizações do SO Linux Ubuntu e são lançadas, comumente, a cada ano. O funcionamento é garantido e o suporte é dado quando são respeitadas as versões entre o *framework* e o SO em utilização.

Exemplo de versões de distribuições, em ordem cronológica, são: *Box Turtle* (Março de 2010), *C Turtle* (Agosto de 2010), *Diamondback* (Março de 2011), *Electric Emys* (Agosto de 2011), *Fuerte Turtle* (Abril de 2012), *Groovy Galapagos* (Dezembro de 2012), *Hydro Medusa* (Setembro de 2013), *Indigo Igloo* (Julho de 2014), *Jade Turtle* (Maio de 2015), *Kinetic Kame* (Maio de 2016) e *Lunar Loggerhead* (Maio de 2017).

A maior distinção se deu entre as versões *Fuerte Turtle* e *Groovy Galapagos*. Versões anteriores à primeira citada apresentavam a estrutura de seus pacotes e ambiente de trabalhos no modelos *rosbuild*. As versões a partir de *Groovy Galapagos* utilizam do padrão *catkin*. Além da estrutura, cada modelo apresentam diferenças de compilação. O principal motivo por essa mudança drástica era a dificuldade de utilização do *framework* em sistemas que não eram o foco da *Willow Garage*. Outro problema era o processo de compilação cruzada, ou *cross-compilation*, para outras arquiteturas e sistemas (62).

O presente trabalho utiliza a distribuição ROS *Indigo Igloo* formato *catkin*. O motivo da escolha do Indigo se dá por ser a versão mais atual e estável no momento inicial desta dissertação.

2.1.4 Estrutura

O ROS é estruturado em pacotes visando a padronização e organização. Assim é possível a familiarização do usuário/desenvolvedor, permitindo a fácil modificação ou incremento de conteúdos. Mais especificamente, cada pacote possui uma atribuição, como por exemplo os pacotes de visão computacional *Ar Track Alvar* (63) e *Stereo Image Proc* (64). Pacotes como o *MAVLink to ROS gateway with UDP proxy for Ground Control Station (MAVROS)* (65) e o *ROSARIA* (66) são responsáveis por estabelecer a comunicação com bases aéreas e terrestres, respectivamente.

Diversos pacotes podem ser encontrados na comunidade ROS, podendo ser consolidados ou em desenvolvimento. Eles podem ser instalados no sistema raiz do SO ou inserido em um *workspace* de desenvolvimento. Quando instalados, os pacotes são atualizados com o sistema e não são passíveis de modificações pelo usuário.

O *workspace* permite a criação e modificações de pacotes já existentes no computador. A compilação do ROS irá atuar sobre os pacotes do *workspace* em questão. O ambiente de trabalho possui os seguintes diretórios em sua estrutura (67):

- **/source:** diretório para localização das fontes de códigos. É neste local que são localizados todos os pacotes em desenvolvimento/utilização;

- **/build:** diretório de compilação. Nele são salvos todos arquivos temporários utilizados pelo *CMake* na compilação dos códigos dos pacotes do diretório *source*;
- **/devel:** diretório de localização dos arquivos executáveis após a compilação. Nós e bibliotecas dinâmicas de cada pacotes localizam-se nesta pasta.

A organização de cada pacote localizado no *source* do *workspace* se dá pelos seguintes arquivos e diretórios:

- **/src:** diretório utilizado para armazenar os códigos fontes em C/C++ ou Python;
- **/srv:** diretório de definições de serviços;
- **/msg:** diretório onde se localiza as definições de mensagens a ser utilizada pelo pacote;
- **/launch:** diretório onde arquivos inicializadores são armazenados;
- **/include:** diretório onde estão armazenados as bibliotecas do pacote, podem também se encontrar no *root* do sistema;
- **CMakeList.txt:** conjunto de instruções de compilações dadas ao compilador associado ao ROS;
- **package.xml:** listagem de pacotes, bibliotecas e ferramentas utilizadas e vinculadas pelo pacote em questão.

2.1.5 Processos

Conforme explicado em 2.1.2, o ROS permite criar um processo maior por meio de outros menores e mais dedicados, ou seja, existem vários blocos de programação focados a uma determinada tarefa. Estes blocos de processamento são denominados "nós". O ROS se assemelha a um *framework* de *Object Request Broker* (ORB). O ORB é utilizado na computação distribuída para execução de tarefas e processos dentro de uma rede. Mais especificamente, ele proporciona e gerencia a comunicação e troca de mensagens (objetos) entre unidades processadoras (68).

No ROS, a comunicação entre nós pode ser estabelecida de duas formas: publicador-subscritor e cliente-servidor. No modelo publicador-subscritor a comunicação se dá por "mensagens", que são estabelecidas via "tópicos". Por exemplo, um nó habilita uma câmera e começa a captura de imagem, esse envia os dados em forma de mensagens a um segundo nó capaz de filtrar a imagem. A comunicação é contínua e sua taxa é definida pelo desenvolvedor. Na Figura 4(58), através da ferramenta de visualização *rxgraph* do ROS,

Figura 4 – Processo envolvendo dois nós e um tópico.

Fonte: RosWiki.org

Seção 2.1.6.10, é possível ver dois nós, um de teleoperação e outro de simulação. Além disso o tópico de comando de velocidade é estabelecido entre os nós.

No modelo cliente-servidor dois nós se comunicam através de requisição de serviços. Um nó cliente requisita um serviço a um nó servidor. Este tipo de comunicação não é contínua, contudo é fortemente acoplada se implementada corretamente, ou seja, o nó cliente é capaz de distinguir se o serviço foi recebido ou não, e o nó servidor capaz de detectar uma requisição.

Esses são o elementos básicos pelo qual aplicações são desenvolvidas no *framework* ROS, e serão melhores detalhadas nas subseções a seguir.

2.1.5.1 Nós

Os nós são blocos de processamento e instruções desenvolvidos nas linguagens *C/C++*, *Python* e *Matlab*, muitas vezes dedicados a uma tarefa e, quando associados, formam um processo maior. O sistema será mais eficiente, confiável e otimizado, pois muito desses blocos podem ser melhor programados e verificados de formas independentes, facilitando o processo de criação.

Outro fator importante é que os nós podem ser reutilizados quantas vezes for necessário em um processo, desde que devidamente adaptados. Muitos nós são desenvolvidos, otimizados e bem consolidados na comunidade ROS permitindo seu uso em diferentes tarefas. Este fato facilita muito ao pesquisador, que muitas das vezes não necessita implementar algoritmos desenvolvidos pela comunidade, como por exemplo habilitar uma *webcam*, e assim poderá recorrer a um processo mais confiável, sendo esse testado em diversas outras aplicações por outros usuários.

O ROS proporciona um grande número de funções relacionadas aos nós que objetivam organizar processos muito amplos, facilitando o trabalho do desenvolvedor. Estas funções são apresentadas a seguir:

- **rosrun:** executa um nó. É necessário informar como parâmetro o pacote e o seu executável respectivamente;
- **rosnode ping:** testa a conexão entre os nós;
- **rosnode list:** lista os nós que estão sendo executados;
- **rosnode info:** demonstra informações sobre os nós;
- **rosnode machine:** lista os nós de uma máquina específica;
- **rosnode kill:** termina um nó em execução.

No modelo publicador-subescritor, Seção 2.1.5, os nós que publicam mensagens e inicializam tópicos são denominados de publicadores, aqueles que leem são chamados de subscritores. Um nó pode tanto ler quanto publicar ao mesmo tempo. No modelo cliente-servidor, Seção 2.1.5, nós que requisitam serviços são denominados nós clientes. Já os que fornecem serviços são denominados nós servidores.

2.1.5.2 Tópicos

Os tópicos são os canais de comunicação entre os nós de um processo. Cada tópico é criado para transmitir um tipo de mensagem, permitindo uma melhor organização das comunicações. Os nós publicadores são os responsáveis pela inicialização dos tópicos para que suas mensagens sejam enviadas, outros parâmetros são fundamentais, como nome, taxa de publicação e nome dos tópicos. Um nó pode inicializar e executar vários tópicos assim como pode subscrever de vários outros tópicos. Dessa forma, os nós não sabem da existência de outros nós, dissociando-os.

Os tópicos são caracterizados como método de comunicação unidirecional, já os serviços são os indicados caso seja necessário a comunicação em ambas direções, bidirecional. O ROS suporta o transporte via *Transmission Control Protocol* (TCP) e *User Datagram Protocol* (UDP), conhecidos como *ROS Transmission Control Protocol* (TCPROS) e *ROS User Datagram Protocol* (UDPROS) respectivamente, sendo o primeiro o padrão. Vale ressaltar que o ROS gerencia os meios de comunicação, ordem e tipos em tempo de processamento.

As funções relacionadas aos tópicos são apresentadas a seguir:

- **rostopic bw:** informa largura de banda do tópico especificado;
- **rostopic echo:** imprime as mensagens que passam pelo tópico especificado;
- **rostopic hz:** informa a taxa de publicação do tópico;
- **rostopic list:** lista os tópicos ativos;

- **rostopic pub:** publica dados para o tópico;
- **rostopic type:** informa o tipo do tópico;
- **rostopic find:** encontra os tópicos baseados em seus referentes tipos.

2.1.5.3 Mensagens

As mensagens são os dados, ou seja, a informação transmitida de um nó ao outro por um ou mais tópicos. Como explanado na Seção 2.1.5.2, os tipos de mensagens definem os tipos dos tópicos. Estas mensagens podem ser estruturadas em dados convencionais, como *int*, *float*, *bool*, *string*, *char* e suas derivações, e também em dados mais complexos estruturados pelo próprio ROS, como *streaming* de imagens, vídeos e *frames*. Ademais, novas mensagens podem ser definidas como uma combinação de mensagens já existentes.

Arquivos do tipo mensagens são definidos e alocados no diretório */msg* do pacote, vide Seção 2.1.4, e construídos quando a linha *genmsg()* é adicionada em seu *CMakeList.txt*. Portanto, ferramentas do ROS conseguem gerar códigos-fonte a partir das definições de mensagens para que essas sejam reconhecidas em vários tipos de linguagens. Assim, o ROS torna-se uma plataforma expansível nos aspectos de comunicação e integração de sistemas.

A Figura 5 (58) demonstra como são declaradas as mensagens em um bloco.

Figura 5 – Bloco de mensagem.

```
string first_name
string last_name
uint8 age
uint32 score
```

Fonte: RosWiki.org

Comandos relacionados:

- **rosmsg show:** informa o tipo de um campo de mensagem;
- **rosmsg list:** lista todos os tipos de mensagens;
- **rosmsg package:** lista as mensagens de um pacote;
- **rosmsg packages:** lista os pacotes que utilizam da mensagem;
- **rosmsg md5:** imprime a soma de verificação da mensagem, o *checksum*.

2.1.5.4 Serviços

O tipo de comunicação publicador e subescritor é um método muito útil para uma gama de aplicações, mas caso o desenvolvedor necessite de comunicações fortemente acopladas, o ROS disponibiliza um método bidirecional de comunicação, os serviços.

Definido por um par de mensagens, devem ser alocados no diretório */srv* do pacote, vide Seção 2.1.4, e construídos quando inserido a linha *gensrv()* no CMakelist.txt do pacote. Cada mensagem que constitui o serviço é do tipo pedido e resposta e são declaradas em um único bloco como na Figura 6 (58). Os delimitadores (- - -) são os responsáveis por separar os blocos de pedido e resposta. Assim, o ROS consegue gerar um código fonte compatível a diversas linguagens de comunicação, do mesmo modo referido na Seção 2.1.5.3.

Figura 6 – Bloco de serviço.

```
int64 A
int64 B
---
int64 Sum
```

Fonte: RosWiki.org

As funções associadas aos serviços são semelhantes às das mensagens, demonstradas a seguir:

- **rossrv show:** informa o tipo de um campo de serviço;
- **rossrv list:** lista todos os tipos de serviços;
- **rossrv package:** lista os serviços de um pacote;
- **rossrv packages:** lista os pacotes que utilizam do serviço;
- **rossrv md5:** imprime a soma do serviço.

2.1.5.5 Nó Mestre

Para que todo este processo envolvendo nós, mensagens, serviços e tópicos, seja bem estruturado e executado, o nó mestre deve ser executado. É ele o responsável pelos serviços de nomenclatura para toda execução do processo. O nó mestre é uma ferramenta fundamental, pois rastreia os nós e permite que eles se comuniquem. É ele que se encarrega das prioridades e sequenciamento das tarefas.

Quando um nó subscritor requisita um acesso a um tópico específico, o mestre será o encarregado de estabelecer esta conexão. Portanto, é extremamente importante os nomes dados aos elementos do processo, como nós e tópicos, já que o mestre os identifica

assim. Consequentemente, pode-se ter vários nós de mesma função desde que identificados de formas diferentes.

Mudanças em parâmetros relacionados aos elementos do ROS, e inserção de novos podem ser feitos durante o tempo de execução. O mestre é capaz de lidar com estes tipos de mudanças, apesar de não ser o mais indicado.

Portanto, antes de qualquer início de processo se torna necessário a ativação do mestre, também chamado de *core* ou */cout*, sendo esta feita pelo comando *roscore*.

2.1.5.6 *Launch*

Launchs são os arquivos inicializadores, não são cruciais para o funcionamento do sistema, mas torna o processo de execução ágil e prático. Alocados no diretório */launch*, vide Seção 2.1.4, são executados pelo comando *roslaunch*. Escrito em *eXtensible Markup Language* (XML), permite executar automaticamente um conjunto de comandos de forma sequencial, ou seja, é capaz de executar os nós remotamente, além de passar parâmetros necessários para cada nó ou tópico, incluindo renomeamentos. Algumas *tags* foram adicionadas, com o intuito de incrementar o XML para funcionalidades ROS. Essas *tags* são expostas a seguir:

- **<launch>:** inicializador, elemento raiz que permite que outras *tags* atuem;
- **<node>:** especifica o nó a ser executado, mesmo papel do comando *rosrun*;
- **<machine>:** especifica em qual máquina será executado determinado nó;
- **<include>:** permite a recursividade a outros arquivos launch;
- **<remap>:** reconfigura nomes;
- **<env>:** define variáveis de ambiente;
- **<param>:** define parâmetros;
- **<rosparam>:** configura parâmetros no servidor de parâmetros, mesmo papel do comando *rosparam*, Seção 2.1.6.7;
- **<arg>:** entrada de argumentos;
- **<test>:** similar à *tag node* mas executa o nó após um teste estabelecido;
- **<group>:** *tag* de organização, permitindo mudanças a um grupo de nós.

2.1.6 Ferramentas

Uma vez apresentado o funcionamento do *framework* assim como seus conceitos básicos, serão demonstradas a seguir algumas ferramentas do ROS, sendo ilustrado seus comandos de terminal. O desenvolvimento constante da comunidade proporciona o surgimento de novas ferramentas, portanto, serão ilustrados aqui as fundamentais. Muitos destes comandos também potencializam funções existentes do próprio Linux.

2.1.6.1 *roscd*

O *roscd* é o comando que permite o acesso a diretórios, equivalente ao *cd* do Linux, mas sem a necessidade de informar o caminho da pasta, apenas seu nome.

$ roscd [pacote]

2.1.6.2 *rosls*

O *rosls* é um comando de listagem de conteúdo do pacote, equivalente ao *ls* do Linux mas sem necessidade de informar o caminho da pasta, apenas o seu nome.

$ rosls [pacote]

2.1.6.3 *rospack*

O *rospack* localiza um pacote ou informa as suas dependências.

$ rospack find [pacote]

$ rospack depends [pacote]

2.1.6.4 *catkin_create_pkg*

O comando *catkin_create_pkg* cria um pacote nos padrões ROS. Criando seu *CMakeList.txt* e *package.xml* por exemplo.

$ catkin_create_pkg [pacote][dependências]

2.1.6.5 *catkin_make*

Normalmente o desenvolvedor codificará arquivos-fonte em *C/C++* ou *Python*, utilizando ou não uma *Integrated Development Environment* (IDE). Mais especificamente, estes códigos-fonte gerarão os nós, e devem ser alocados no diretório */src* do pacote do projeto. Uma vez o código-fonte desenvolvido, este deverá ser referenciado no *CMakelist.txt* e todos pacotes utilizados no *package.xml*.

Após estas etapas, o compilador deverá ser chamado para gerar os executáveis *bin*, e o comando *catkin_make* deve ser utilizado dentro do pacote ou fora, sendo o último

necessário referenciar o pacote. Após a compilação, o arquivo *bin* gerado se alocará na pasta *devel* do *workspace* utilizado, Seção 2.1.4.

$ catkin_make [pacote]

2.1.6.6 *roswtf*

O *roswtf* imprime erros referentes a um processo ROS em execução.

$ roswtf

2.1.6.7 *rosparam*

O *rosparam* é utilizado para definir parâmetros do servidor de parâmetros do ROS.

- **rosparam set:** configura um parâmetro;
- **rosparam get:** retorna um parâmetro;
- **rosparam load:** carrega um parâmetro de um arquivo determinado;
- **rosparam delete:** deleta o parâmetro especificado;
- **rosparam dump:** deposita um parâmetro para um arquivo determinado;
- **rosparam list:** lista os parâmetros.

2.1.6.8 *rosservice*

Rosservice é uma ferramentas de Serviço do ROS.

- **rosservice list:** lista informações referentes a serviços ativos;
- **rosservice node:** informa o nome do nó que está prestando o serviço;
- **rosservice call:** chama o serviço dado o argumento;
- **rosservice args:** lista os argumentos de um serviço especificado;
- **rosservice type:** informa o tipo do serviço;
- **rosservice uri:** informa o serviço ROSRPC uri;
- **rosservice find:** procura por serviços.

2.1.6.9 *image_view*

O *image_view* é uma ferramenta para visualizar uma imagem publicada por um nó, sendo este o responsável pelo acesso a dispositivos de captação de imagens.

Esta ferramenta é capaz de ler dados em *streaming* contínuo, de tópicos especiais para o transporte de imagem, sendo essas em RGB, escala de cinza, preto e branco, níveis de profundidade ou outros formatos. É importante, também, para interpretações de resultados e *debugs* de sistema.

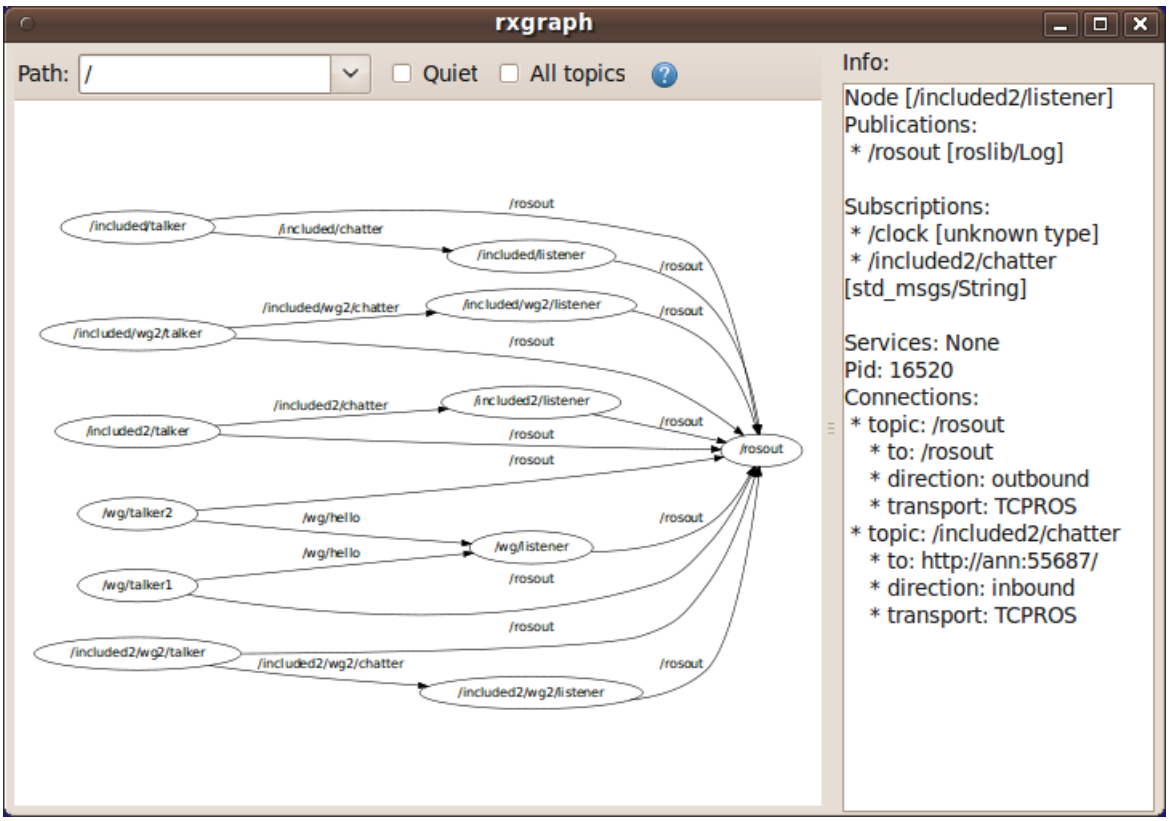
$ rosrun image_view image_view _image:=[caminho do tópico a ser lido]

2.1.6.10 *rxgraph*

A ferramenta *rxgraph* gera um grafo do processo em execução, ou seja, demonstra os nós ativos e como se comunicam pelos tópicos. Um exemplo é apresentado na Figura 7 (58).

$ rxgraph

Figura 7 – Interface rxgraph.



Fonte: RosWiki.org

2.1.6.11 *tf_echo*

A ferramenta *tf_echo* é um método de impressão de tópicos especiais de *streaming* de dados relacionados à *frames*. As mensagens que fluem por estes tópicos podem ser encaradas como blocos de dados que se referem a uma matriz e/ou vetores tridimensionais relacionados às transformações homogêneas (69).

$ rosrun tf tf_echo [frame1] [frame2]

2.1.6.12 *view_frames*

O *view_frames* informa todo o conjunto de transformações de coordenadas.

$ rosrun tf view_frames

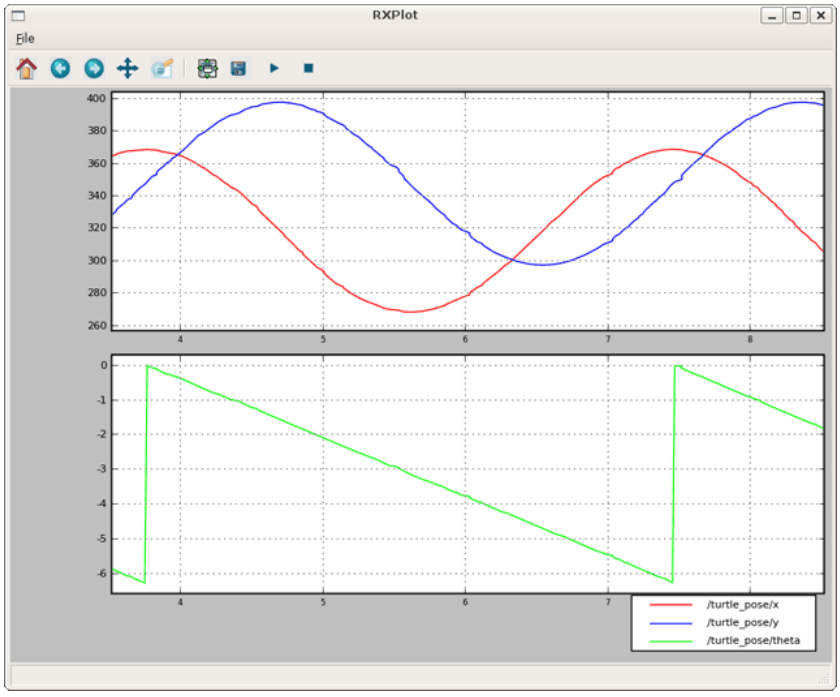
2.1.6.13 *rxplot*

O *rxplot* é outra ferramenta gráfica, sendo a sua função plotar a informação e dados que fluem por um determinado tópico. A Figura 8 (58) demonstra este recurso gráfico.

$ rxplot [/topico1/campo1] [/topico2/campo2], dois gráficos independentes;

$ rxplot [/topico1/campo1], [/topico2/campo2], dois gráficos sobrepostos.

Figura 8 – Interface *rxplot*.



Fonte: RosWiki.org

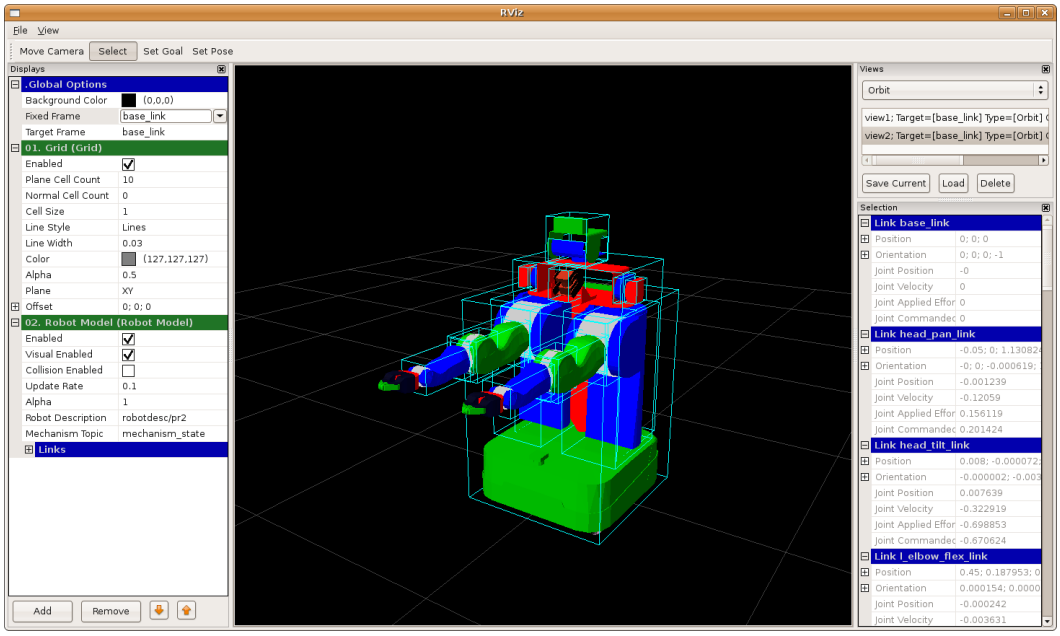
2.1.6.14 *rviz*

O *rviz* é um ambiente de visualização tridimensional para simulação ou testes de resultados. Além da possibilidade de ser simulados situações e robôs, é também muito importante para dimensionamento e compreensão de sistemas, principalmente aqueles envolvendo frames e informações em 3 dimensões. Vale ressaltar que simulações 2D também são possíveis.

Parâmetros devem ser modificados no próprio ambiente de acordo com a necessidade, arquivos do tipo *.rviz* podem ser salvos tornando esta configuração pré-selecionada. Na Figura 9 (58) é possível ver os parâmetros configurados à esquerda da janela, exemplo de modelagem de uma base robótica.

$ rosrun rviz rviz

Figura 9 – Interface rviz.



Fonte: RosWiki.org

2.1.7 Comunidade

O conceito relacionado nesta seção explana a comunidade ROS, como locais de documentação e repositórios, mantendo o *framework* em seu contínuo crescimento.

2.1.7.1 Repositórios

Vários grupos podem disponibilizar seus pacotes, aplicações e desenvolvimentos pelos diversos repositórios da rede do ROS, semelhante aos existentes para o desenvolvimento Linux.

2.1.7.2 ROS Wiki

Principal página de documentação sobre o ROS, nele informações diversas sobre o *framework* são encontradas. O acesso à edição é livre, qualquer usuário cadastrado pode gerar informações complementares, discutir sobre *bugs* ou criar novos temas. Nesta seção são encontrados também tutoriais e auxílio aos iniciantes. Endereço abaixo:

<http://wiki.ros.org/>

2.1.7.3 *ROS Answers*

Ros Answers é um fórum de questionamento para interação e auxílio entre os usuários. Endereço abaixo:

<https://answers.ros.org/questions/>

2.2 MAVLINK

Esta seção apresenta o protocolo de comunicação MAVLINK, abordando sua estrutura e funcionamento.

2.2.1 Apresentação

O MAVLINK é uma biblioteca e protocolo de comunicação para veículos aéreos, permitindo a troca de informação e dados entre o controlador de voo da aeronave (o FCU) e um computador de controle externo, com desempenho focado em velocidade e segurança.

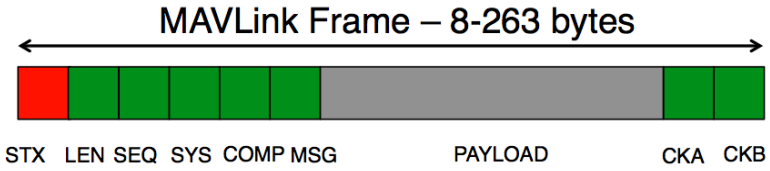
É um *software* aberto desenvolvido por Lorenz Meier (70) em 2009. Extensamente utilizado em placas como *Pixhawk*, *Ar.Drone* e *APM* e *softwares* como *APM planner*, *Copter GCS* e *QgroundControl*, sendo esta a sua mantenedora com documentação e exemplos.

2.2.2 Pacote de mensagens

O MAVLINK cria uma estrutura de mensagem leve, codificando e empacotando-a para que o fluxo de dados possa ser estabelecido entre os processadores ou microcontroladores. Estas mensagens são baseadas no protocolo *Controller Area Network* (CAN) muito utilizado na indústria automobilística e naval, apresentado em 1986 por Bosch R. com intuito de permitir que vários microcontroladores e *chips* se interajam por mensagens seriais curtas e síncronas sem um servidor central (71).

As mensagens MAVLINK são constituídas de uma sequência de *bytes* (pacote) que variam, podendo ser no mínimo 8 e máximo 263 *bytes* de acordo com seu *Payload*, Figura 10 (70).

Figura 10 – Estrutura de um pacote de mensagem MAVLINK.



Fonte: QgroundControl.org

Mais especificamente, cada bloco do pacote de mensagem (Figura 10) tem uma determinada função exposta a seguir:

- **STX:** Padrão de um *byte* responsável por indicar o início de um novo pacote de mensagem;

- **LEN:** Indica o comprimento do *payload* do bloco, também é de um *byte* podendo variar de 0 a 255;
- **SEQ:** Uma mensagem com *payload* maior que 255 *bytes* é dividida entre diferentes pacotes que são ordenados por este bloco. Permite também a detecção de perdas de pacotes. Varia de 0 a 255;
- **SYS:** Identificação de vários IDs de diversas aeronaves conectadas a um único controlador central. São permitidos um total de 255 aeronaves, sendo 0 um parâmetro inválido;
- **COMP:** Sobre uma FCU, diversos componentes como sensores de GPS, barômetro e acelerômetro são correspondidos com um ID deste bloco;
- **MSG:** Este bloco informa que tipo de mensagem o pacote transporta, para que seja decodificada de forma correta;
- **PAYLOAD:** O conteúdo da mensagem, como por exemplo a resposta de dados de um sensor ou determinação do modo de voo da aeronave;
- **CKA e CKB:** Responsável pelo *checksum*. Parâmetros de controle de erro, que permitem verificar se a mensagem esta corrompida e desta forma ser descartada.

Os tipos de variáveis suportadas são os convencionais de programação, como: inteiros sinalizados ou não de 8, 16, 32, 64 bits, flutuantes do tipo *float* e *double*, e vetores. Além destas, uma variável especial foi definida, a *uint8_mavlink_version*, que carrega a informação de versão do protocolo e pode apenas ser lida.

2.2.3 Mensagens

O conteúdo das mensagens MAVLINK possuem diversas funcionalidades de atuar e ler informações da aeronave, sendo este processo realizado durante o voo ou não.

A mensagem mais importante é a *Heartbeat*, esta é enviada ao computador de controle assim que a conexão com a aeronave é estabelecida, e posteriormente é mantido o seu envio durante todo o processo informando a existência da comunicação. Caso esta mensagem seja perdida, a aeronave entrará em um estado de segurança, podendo voltar ao local de decolagem, por exemplo.

Algumas mensagens de pré voo podem ser estabelecidas informando parâmetros de *Home Location*, local onde o veículo irá retornar em caso de falha, velocidades horizontal e vertical máximas, tipo de aeronave utilizada, entre outras.

Definições de missão podem ser estabelecidas por estas mensagens, sendo estabelecidas antes do voo, informando atitudes que o *hardware* deve tomar quando determinada

coordenada é atingida. Estas missões podem incluir voo autônomo a determinada latitude e longitude, *Loitter* (manter a aeronave fixa em determinada altura), pouso ou decolagem.

Durante o voo, mensagens contendo informações de bateria, acelerômetro, barômetro, GPS entre outros sensores podem ser verificadas. As mensagens possíveis são inúmeras uma vez que o protocolo é de desenvolvimento aberto e contínuas modificações são aplicadas pela comunidade, as principais podem ser verificas em (70).

Para que o desenvolvedor estruture sua própria mensagem, um arquivo XML deve ser montado e posteriormente gerado e compilado em *C/C++* ou *Python* de acordo com o código de aplicação. Este arquivo deve possuir as seguintes informações e *tags*:

- **ID:** identificador da mensagem para o sistema;
- **Name:** nome da mensagem usado para referências em nível de codificação;
- **Description:** não obrigatório para o funcionamento, contudo descreve a mensagem para que outros usuários possam utilizá-la;
- **Field:** codifica uma determinada variável ou mensagem do sistema nos tipos descritos na Seção 2.2.2. Desta forma a nova mensagem contém as informações de interesse.

Um exemplo de XML de mensagem é exposto na Figura 11, ela mostra a mensagem *Heartbeat*. É possível inferir que as *tags Field* remetem a um *name* de funções pré-definidas do protocolo associado ao *hardware* utilizado.

Figura 11 – XML da Mensagem *Heartbeat*

```
1 <message id="0" name="HEARTBEAT">
2   <description>The heartbeat message shows that a system is present and
      responding. The type of the MAV and Autopilot hardware allow the
      receiving system to treat further messages from this system
      appropriate (e.g. by laying out the user interface based on the
      autopilot).</description>
3   <field type="uint8_t" name="type">Type of the MAV (quadrotor,
      helicopter, etc., up to 15 types, defined in MAV_TYPE ENUM)</field>
4   <field type="uint8_t" name="autopilot">Autopilot type / class. defined
      in MAV_CLASS ENUM</field>
5   <field type="uint8_t" name="base_mode">System mode bitfield, see
      MAV_MODE_FLAGS ENUM in mavlink/include/mavlink_types.h</field>
6   <field type="uint32_t" name="custom_mode">Navigation mode bitfield, see
      MAV_AUTOPILOT_CUSTOM_MODE ENUM for some examples. This field is
      autopilot-specific.</field>
7   <field type="uint8_t" name="system_status">System status flag, see
      MAV_STATUS ENUM</field>
8   <field type="uint8_t_mavlink_version" name="mavlink_version">MAVLink
      version</field>
9 </message>
```

Fonte: Elaborada pelo autor

2.3 *PX4 FIRMWARE*

Nesta seção é descrito o *firmware* de controle aéreo utilizado na presente dissertação: o *PX4 Autopilot*. São descritos o *PX4 Middleware* e o *PX4 Flight Stacks* que fazem parte do *PX4*. As informações apresentadas nesta seção são baseadas nas referências (20, 21, 26, 72).

2.3.1 Apresentação

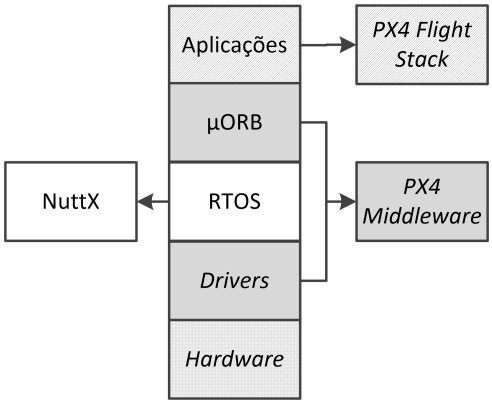
Visando o desenvolvimento de um sistema robusto, genérico, expansível e aplicável em sistemas embarcados de voo aéreo, o laboratório de Visão Computacional e Geometria do Instituto Federal da Suíça (ETH Zurich) iniciou o projeto do *PX4 Autopilot firmware*. A ideia principal é o desenvolvimento de um *firmware Open Source* de controle aéreo com suporte à preempção, ou seja, capacidade de execução de multiprocessamento de forma eficaz com prioridades e bloqueios de tarefas. Dessa forma, o sistema é capaz de lidar com múltiplos processamentos inerentes a um veículo aéreo: sensoriamento e atuação. Outro foco, como relatado em (26), é a simplicidade em interações com sistemas multinodais, como é o caso do *framework* ROS. Essas abordagens descritas tornaram o *PX4* uma opção válida, e em crescimento, em comparação a outros *firmwares* amplamente difundidos, como: *OpenPilot* e o *APM*. Esses *firmwares* não realizam o multiprocessamento ou não visam interações mais eficientes com outros sistemas multinodais.

A arquitetura do *PX4* é constituída de três componentes principais: o *Real-time Operating System* (RTOS) *Nuttx*, o *PX4 Middleware* e o *PX4 Flight Stacks* (Figura 12). O *Nuttx* é um sistema operacional Linux que permite o gerenciamento de memória, o multiprocessamento e interrupções do sistema em tempo real de processamento. Além disso, o *Nuttx* proporciona o sistema base para os módulos do *PX4*. O *PX4 Middleware*, Seção 2.3.2, é o responsável pelos *drivers* e o sistema μORB, enquanto o *PX4 Flight Stacks*, Seção 2.3.2, se encarrega das aplicações embarcadas, como controladores de voos e estimadores.

2.3.2 *PX4 Middleware*

O *PX4 Middleware* é dividido em duas camadas: a camada de *drivers* e o μORB. A camada de *drivers* se diz respeito ao conjunto de códigos e *drivers* de sensores e atuadores. A proposta dos *drivers* (26) é interfacear o *hardware* com o *software*, permitindo a leitura de dados de sensores e atuadores para a aeronave. Além do mais, os *drivers* suportam a execução em *callbacks* de sistema e permitem interrupções, sendo compatíveis com o processamento *multithreading* do *PX4*. O foco também é lidar com diferentes sensores com distintas taxas de atualizações proporcionados por *hardwares*, como a *Pixhawk*, e consequentemente disponibilizar dados em barramentos separados.

Figura 12 – Arquitetura PX4



Fonte: Elaborada pelo autor

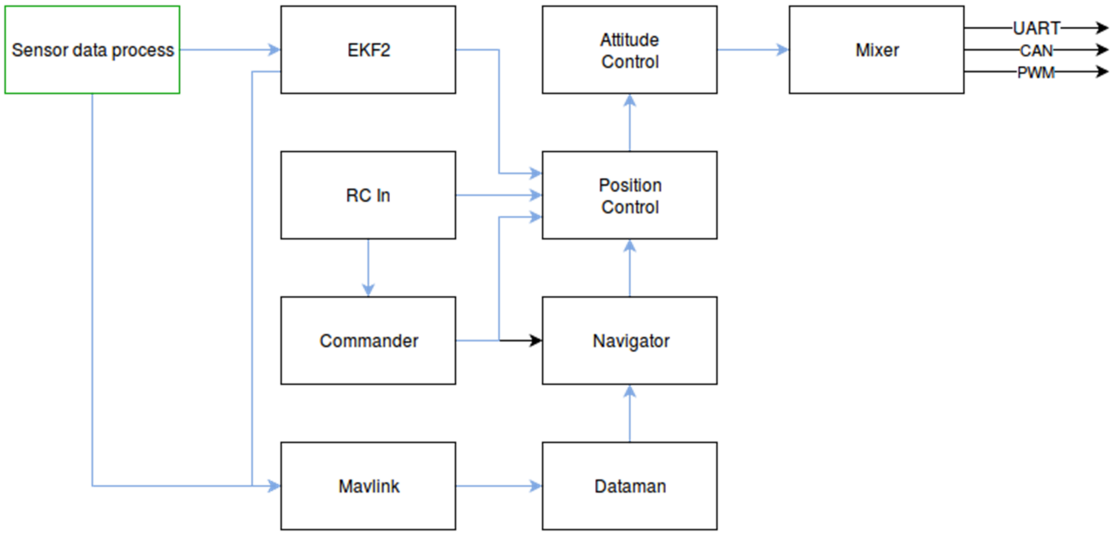
A segunda camada do *middleware* implementa o sistema denominado μORB, ou seja, permite através de um conjunto de bibliotecas que aplicações sejam realizadas sobre o *firmware* em uma estrutura ORB, como descrito na Seção 2.1. Portanto as aplicações desenvolvidas serão multinodais: com nós publicadores e subscritores. Os nós utilizam da requisição e fornecimento de dados através de mensagens pré-definidas transportadas em tópicos. O padrão publicador-subscritor que o sistema μORB proporciona permite o gerenciamento e execuções de processos de forma mais eficiente em um sistema *multithreading*. Esses nós, assim como os *drivers*, estão submetidos a *callbacks* e interrupções do sistema (26). Adicionalmente, essa estrutura assemelha o *PX4* ao ROS, tornando-o compatível com esse *framework* (26).

2.3.3 *PX4 Flight Stack*

O *PX4 Flight Stack* é o conjunto de aplicações embarcadas no *firmware PX4*. Cada aplicação é desenvolvida e executada em nós independentes e utilizam do sistema μORB. As aplicações envolvem algoritmos de controle, navegação, orientação e estimadores de posição para veículos aéreos. Baseando-se na Figura 13, as aplicações são divididas como seguem:

- **EKF2**: O EKF2 é o algoritmo estimador espacial da aeronave baseado em filtro de Kalman estendido. Este estimador realiza a fusão sensorial de dados provindos de sensores da aeronave, e assim, é capaz de estimar o posicionamento, a velocidade e a orientação. Os sensores compatíveis são: IMU, GPS, Magnetômetro, *Ranger Finder*, *Airspeed Sensor*, *Optical Flow* e algoritmos de visão;
- **Commander**: O *Commander* é aplicação de mais alto nível embarcada na aeronave. Realiza comandos como: armar motores, *return to home*, decolagem e pouso;

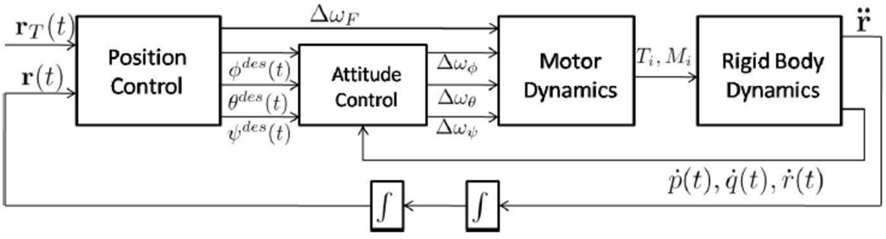
Figura 13 – Estrutura do *PX4 Flight Stack*.



Fonte: dev.px4.io

- **Navigator**: O *Navigator* é aplicação responsável por traduzir rotas de missões preestabelecidas em comandos para os controladores de baixo nível da aeronave;
- **Position Control**: *Position Control* é o controle PID de posição de baixo nível da aeronave, Figura 14 (20). Baseado nas rotas estabelecidas por aplicações superiores, o *Position Control* determina os ângulos de *roll*, *pitch* e *yaw* desejados para o cumprimento da trajetória. A variação de velocidades angulares dos motores que resultem em uma força vertical para o cumprimento da trajetória também é determinado por este controlador (20);
- **Attitude Control**: *Attitude Control* é controle de atitude PD de baixo nível da aeronave, Figura 14 (20). Dado os ângulos de *roll*, *pitch* e *yaw* estabelecidos pelo *Position Control*, o *Attitude Control* traduz esse espaço de estados em variações de velocidades angulares dos motores que resultem nos ângulos de *roll*, *pitch* e *yaw* (20).

Figura 14 – Malha de controle embarcada no *PX4*



Fonte: The grasp multiple micro-UAV testbed.
N. Michael, D. Mellinger and Q.Lindsey e V. Kumar

- **Mixer**: As aeronaves suportadas e incluídas no *stack* são: asas fixas, n-rotores e VTOLs. A estrutura específica da aeronave é informada através de arquivos de *Mixers* que indicam seus *airframes*. A aplicação mixer é responsável também por traduzir sinais de controle (força vertical, taxa de variação de *roll*, *pitch* e *yaw*) em sinais de *Pulse-width Modulation* (PWM) que são atuados diretamente nos motores.
- **RC In**: O RC In é o módulo que realiza o reconhecimento de sinais vindo de canais do rádio controle. Posteriormente os envia para aplicações de controla da aeronave;
- **Dataman**: O *Dataman* é responsável pelo gerenciamento de memória de dados da aeronave e missão.
- **Mavlink**: A aplicação Mavlink implementa o protocolo MAVLINK, Seção 2.2, na controladora de voo. Logo envia e recebe mensagens desse protocolo;

2.3.4 Modos de Voo

O *PX4 autopilot* estabelece o uso da aeronave baseando-se em diferentes modos de voos. Cada um deles proporciona métodos de controle da aeronave, como manual, assistido ou autônomo. Desta forma, o *firmware* gerencia a atuação de suas aplicações na camada de *stacks*. Os tipos de modos de voo de n-rotores são apresentados abaixo:

- **MANUAL:**
  - **Manual:** Controle do piloto é enviado diretamente ao *Mixer*, ou seja, direto aos motores atuando nas taxas de *roll*, *pitch*, *yaw* e de subida.
  - **Stabilized:** Controle do piloto é enviado como ângulos de *roll* e *pitch* e como taxa de *Yaw* (velocidade angular em *Yaw*). O comando translacional no eixo Z é realizado diretamente no *Mixer*.
- **ASSISTIDO:**
  - **AltCtl:** Comandos de orientação são livres como o modo manual, contudo a aceleração é limitada em uma faixa predeterminada para subida e descida. Logo, mantém-se a aeronave em uma região de altura desejada.
  - **PosCtl:** modo manual, porém controla-se a velocidade da esquerda para direita e de frente para trás, ao invés da taxa de *roll* ou *pitch*.
  - **Loiter:** Mantém a aeronave parada sobre um posição.
  - **RTL:** Retorna a aeronave para o local de *Home*. O retorno é executado em uma altura de segurança preestabelecida em linha reta. Quando a aeronave se encontra sobre o *Home*, o procedimento de descida é executado.

  - **Mission:** Executa a missão preestabelecida em *softwares* de estação base. Comutará para *loiter* caso a missão seja finalizada ou não encontrada.

- **OFFBOARD:** Modo de voo que indica que comandos de posicionamento e velocidade são realizados por um processador ou computador externo, também denominado computador companheiro. A comunicação entre os processadores da FCU e do computador companheiro devem ser realizados sobre o protocolo MAVLINK através de uma comunicação serial. Por critério de segurança, essa comunicação deve ser estabelecida com uma taxa mínima de 2Hz. Caso contrário o *firmware* irá comutar para o modo *PosCtl*.

2.4 MAVROS

Nesta seção é abordado e discutido o pacote do ROS *Indigo* denominado *MAVROS*. O estudo aqui apresentado leva em consideração o pacote já instalado e funcional no *framework*, assim como a biblioteca do protocolo MAVLINK, apresentado no Capítulo 2.2. Versão Linux Ubuntu 14.04 utilizada.

2.4.1 Apresentação

O *MAVROS* é um pacote do *framework* ROS, abordado no Capítulo 2.1, que se encontra em construção e é mantido por Vladimir Ermakov (65). Devido ao seu atual desenvolvimento até a execução do presente projeto, a sua documentação é escassa, sendo boa parte do conteúdo aqui abordado construída no âmbito deste trabalho.

Este pacote é um *proxy*, integrando funcionalidades MAVLINK, Capítulo 2.2, com o ROS (73). Inclusive, permite maior abstração quanto às regras e técnicas do protocolo MAVLINK desenvolvidas no Capítulo 2.2, ou seja, dados de leituras de sensores e atuadores do *hardware* são disponíveis para o desenvolvedor de forma mais simplificada.

O *MAVROS* possui compatibilidade com placas de controle *ArdupilotMega* e *Pixhawk*, e com *softwares* e *firmwares* de controle *APM* e *PX4*. Porém é indicado o seu uso com o *firmware PX4* embarcado à *Pixhawk*. A justificativa se dá pelo foco dado pela comunidade desenvolvedora no acesso aos dados e no desempenho dos algoritmos embarcados. Outra vantagem em se utilizar o *PX4* em associação ao *MAVROS* é a capacidade de conexão via protocolo UDP com o *software Qgroundcontrol*, permitindo análises de voos mais sofisticadas, que envolvem representações gráficas de trajetórias em mapas, calibrações de sensores, além de dados de *log* de missões. Este pacote não é compatível com as versões *rosbuild* do ROS, como *Electric* e *Fuerte* (59).

2.4.2 Funcionamento

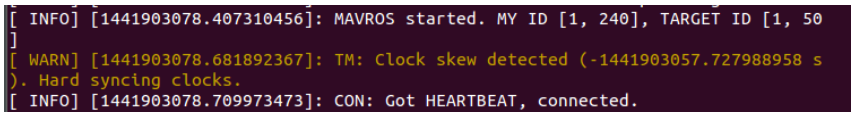
O nó principal que estabelece o fluxo de troca de mensagens entre a FCU e o computador companheiro é executado por *launchs*, Seção 2.1.5.6. Cada *launch* carrega argumentos como *baud rate* e dispositivo de comunicação. Esses dispositivos podem ser do tipo UDP, cabo USB/*Serial* ou *Wireless*/*Serial*. A FCU deve ser identificada neste arquivo, além de qual componente. Por padrão todos componentes são selecionados. Esta identificação é feita de acordo com o protocolo MAVLINK, elucidado no Capítulo 2.2, pelos códigos *SYS* e *COMP*.

Outro argumento importante que o *launch* contém é o arquivo de configuração com extensão *YAML Ain't Markup Language* (YAML), sendo este o responsável por definir

qual *firmware* é utilizado, como *PX4* ou *APM*. Esse arquivo configura, define e habilita mensagens compatíveis com o *firmware* utilizado em projeto.

Inicializado o *launch*, uma mensagem de *heartbeat*, Capítulo 2.2, é exposta informando o estabelecimento da comunicação entre a FCU e o computador companheiro (Figura 15). Portanto, o desenvolvedor estará apto a inicializar o uso das funcionalidades ROS e MAVLINK.

Figura 15 – Conexão estabelecida entre o ROS e a FCU indicada pela mensagem *HEARTBEAT*



Fonte: Elaborada pelo autor

2.4.3 Tópicos

Com o *MAVROS* em execução, tópicos são estabelecidos com fluxo de mensagens permitindo a leitura de dados e atuação sobre a FCU. Estes tópicos serão listados e explanados a seguir, ressaltando que os dados podem ser providos diretamente pelos dispositivos ou sensores, ou após serem processados pelos algoritmos da própria FCU:

- **.../from** : lê parâmetros da FCU. Estrutura de mensagem MAVLINK;
- **.../to** : envia parâmetros para FCU. Estrutura de mensagem MAVLINK;
- **.../actuator_control**: controle de atuadores da aeronave nos canais de saída de PWM da FCU. Permite a mixagem de sinais e atuação independente e individual de cada atuador da aeronave;
- **.../battery**: informações de estado da bateria;
- **.../global_position/compass_hdg**: informação do ângulo da bússola estimada pela FCU através de dados de GPS;
- **.../global_position/global**: dados de coordenadas de GPS, via latitude e longitude, processados pela FCU;
- **.../global_position/gp_vel**: dados de coordenadas *Universal Transverse Mercator* (UTM), método de representação horizontal da posição;
- **.../global_position/local**: velocidade estimada pela FCU através dos dados de GPS;
- **.../global_position/rel_alt**: altura relativa;

- **.../gps/fix**: posição por coordenada GPS vindas diretamente do módulo de GPS, ou seja, não há nenhum processamento pela FCU;
- **.../gps/gps_vel**: velocidade estimada pelo módulo de GPS, ou seja, não há nenhum processamento pela FCU;
- **.../imu/atm_pressure**: pressão atmosférica;
- **.../imu/data**: dados de IMU com orientação processada pela FCU;
- **.../imu/data_raw**: dados diretamente do dispositivo IMU, logo sem orientação processada;
- **.../imu/mag**: dados do campo magnético processados pela FCU;
- **.../imu/temperature**: dados de temperatura relatados pela FCU;
- **.../local_position/local**: dados de posição relativa ao momento em que a FCU é ligada;
- **.../mission/waypoints**: lista de *waypoints* salvos previamente;
- **.../radio_status**: informações do rádio controle;
- **.../rc/in**: acesso aos dados provindos dos canais do rádio controle;
- **.../rc/out**: publica nas saídas dos servos;
- **.../rc/override**: permite publicar dados nos canais do rádio;
- **.../safety_area/set**: configura a área de segurança;
- **.../setpoint_attitude/att_throttle**: permite enviar dados de aceleração de *roll*, *pitch* e *yaw*;
- **.../setpoint_attitude/attitude**: permite enviar valores de *roll*, *pitch* e *yaw*;
- **.../setpoint_position/local**: permite enviar deslocamentos em posição tridimensional;
- **.../setpoint_velocity/cmd_vel**: envia velocidades lineares;
- **.../state**: análise de estado do VANT, como estado dos motores ou modo de voo;
- **.../time_reference**: referência de tempo computada pela FCU;
- **.../vfr_hud**: dados para *display* de voo ou *Head Up Display* (HUD).

2.4.4 Serviços

O *MAVROS* também estabelece serviços, Seção 2.1.5.4, que são listados a seguir:

- **.../cmd/arming**: arma os motores;
- **.../cmd/command**: envia mensagens no formato comando *long* do MAVLINK;
- **.../cmd/command_int**: envia mensagens no formato comando *int* do MAV LINK;
- **.../cmd/guided_enabled**: habilita o modo de voo *OFFBOARD*;
- **.../cmd/land**: determina o pouso da aeronave em um *waypoint* passado como parâmetro;
- **.../cmd/set_home**: define a localização do *HOME*, local de segurança;
- **.../cmd/takeoff**: determina decolagem da aeronave em um *waypoint* passado como parâmetro;
- **.../mission/clear**: limpa a lista de *waypoint* gravada previamente;
- **.../mission/pull**: requisita a lista de *waypoint*;
- **.../mission/push**: envia novos *waypoints*;
- **.../mission/set**: acrescenta a posição atual na lista de *waypoint*;
- **.../set_mode**: determina o modo de voo da aeronave;
- **.../set_stream_rate** : verifica a taxa de comunicação com a FCU.

O *MAVROS* também proporciona serviços de acesso e manipulação a arquivos do sistema embarcado à FCU, todos pelo protocolo *File Transfer Protocol* (FTP). Estes envolvem desde copia, leitura e modificações de arquivos, criação de diretórios, além de outras funcionalidades.

2.4.5 Nós de comandos

Nós de comandos, ou apenas comandos, são recursos extras disponibilizados pelo pacote *MAVROS* codificados em *Python*. Utilizam serviços de maneira a abstrair a interação com o desenvolvedor, visando maior facilidade e rapidez para testes e *debugs*. Os comandos são executados em linhas no terminal Linux, sendo sua estrutura apresentada a seguir:

rosrun mavros <nó> <comando>

Os nós comandos são listados abaixo. Em adição, os comandos relativos a cada nó também são apresentados.

- **mavcmd:** comandos básicos, sendo eles:

  - *long:* envia qualquer comando do tipo MAVLINK *long*;

  - *int:* envia qualquer comando do tipo MAVLINK *int*;

  - *sethome:* define um novo ponto como *home*;

  - *takeoff:* determina a decolagem no *home*;

  - *land:* determina o pouso no *home*;

  - *takeoffcur:* determina a decolagem na coordenada atual;

  - *landcur:* determina o pouso na coordenada atual;

  - *trigger_control:* sinais de controle do gimbal.

- **mavsafety:** comandos relacionados a segurança:

  - *arm:* arma os motores (estado de baixa rotação);

  - *disarm:* desarma os motores (sem rotação);

  - *safetyarea:* define a área de segurança. É um volume que delimita o espaço de atuação da aeronave, definida por dois vértices de um cubo imaginário.

- **mavparam:** manipulação de parâmetros:

  - *load:* carrega os parâmetros do arquivo;

  - *dump:* esvazia os parâmetros do arquivo;

  - *get:* lê o valor de um parâmetro;

  - *set:* configura um parâmetro.

- **mavsetp:** atuador de posição:

  - *local:* envia uma posição espacial.

- **mavsys:** configurações do sistema:

  - *mode:* determina o modo de voo;

  - *rate:* determina a taxa de comunicação.

- **mavwp:** manipulação de missões de *waypoints* por GPS. Só é possível manipular esta missão previamente ao voo.

  - *show:* lista os *waypoins* gravados previamente na placa;

  - *load:* carrega os *waypoints* na placa;

- *pull:* maneja os *waypoints* gravados;

- *clear:* esvazia os *waypoints*;

- *setcur:* grava a posição atual como um *waypoint*.

- **mavftp:** manipulação de arquivos do *firmware PX4* remotamente, semelhante a comandos do sistema Unix:

  - *cd:* mudar de diretório;

  - *list:* listar arquivos e diretórios;

  - *cat:* ler o conteúdo do arquivo;

  - *remove:* remover um arquivo;

  - *mkdir:* criar um diretório;

  - *rmdir:* remover um diretório;

  - *download:* baixar um arquivo embarcado no *firmware*;

  - *upload:* embarcar um arquivo no *firmware*;

  - *verify:* verificar o arquivo;

  - *reset:* resetar o *firmware*.

2.5 AR TRACK ALVAR

O *software* de visão computacional utilizado nessa dissertação é apresentado na presente seção. O intuito é abordar o pacote ROS, *Ar Track Alvar*, elucidando suas funcionalidades e seu funcionamento.

2.5.1 Apresentação

Como relatado no Capítulo 1, a visão computacional mostra-se como fonte alternativa de sensoriamento, além de possuir um crescente potencial de aplicação. Baseado na visão de animais e humanos, vários algoritmos são propostos para realizar tal tarefa. Um dos algoritmos de visão computacional para detecção de marcadores artificiais predispostos no ambiente é o *Alvar*. O *Alvar* foi proposto como biblioteca e conjunto de ferramentas, *Open Source*, para o desenvolvimento de algoritmos de realidade aumentada pela VTT *Technical Research Centre of Finland*. Devido à sua robustez e aplicabilidade, o *Alvar* também se apresenta em pacote ROS. A esse pacote deu-se o nome de *Ar Track Alvar*. O intuito é proporcionar que desenvolvedores o apliquem de forma facilitada baseando-se na estrutura e padronização ROS.

A robustez do *Ar Track Alva*r deve-se ao *threshold* adaptativo à luminosidade e ao método de reconhecimento dos padrões. Esse método, em comparação com outros algoritmos, como o *Ar Tool Kit*, não reduz significativamente o processamento quanto maior for o número de marcadores presente na imagem (63).

2.5.2 Funcionamento

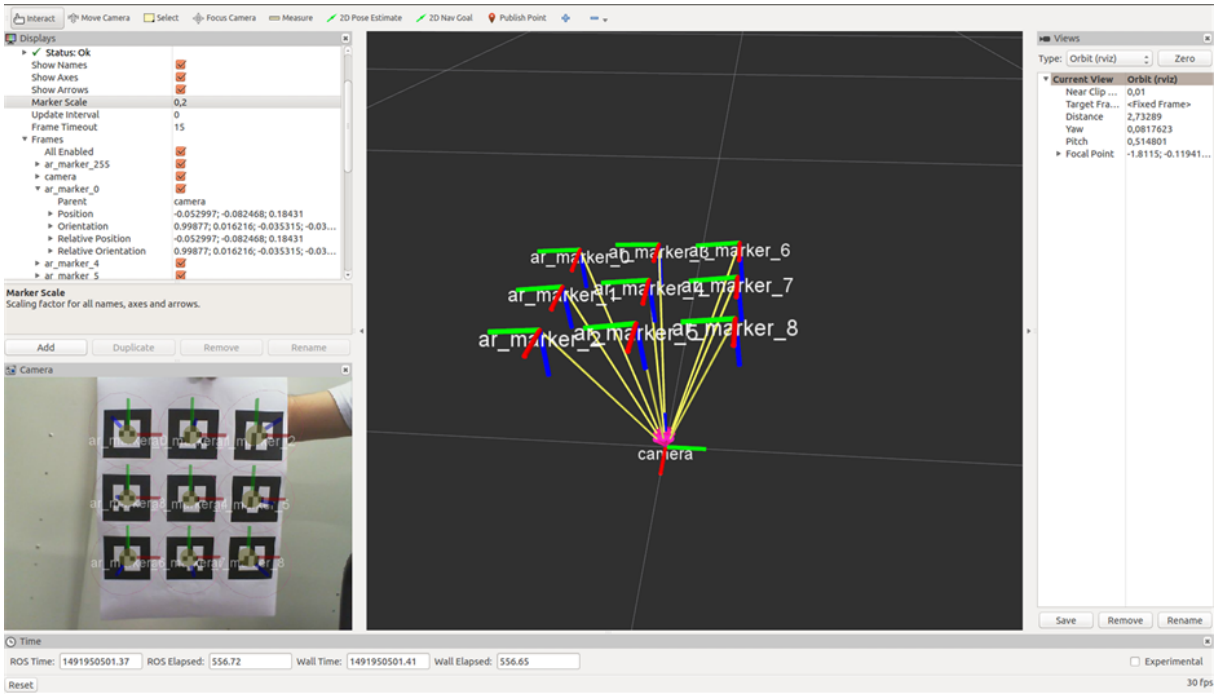
O *Ar Track Alvar* é um pacote que implementa o algoritmo de visão e detecção de *features* artificias, ou seja, ele é capaz de detectar marcadores impressos ou pintados predispostos no ambiente utilizando câmeras convencionais. Além de detectar, o algoritmo é capaz de estimar a posição e orientação relativa do marcador em relação à câmera. Exemplos de marcadores são apresentados na Figura 16.

Figura 16 – Exemplo de marcadores artificiais

Fonte: Elaborada pelo autor

O funcionamento do algoritmo é ilustrado na Figura 17 sobre a ferramenta *RViz* do ROS. Na Figura 17, no canto inferior esquerdo, é possível verificar a imagem adquirida pela *webcam* com a projeção em realidade aumentada dos *frames* de cada marcador artificial identificado. No canto superior esquerdo são mostrados dados de posicionamento dos marcadores, e no centro da imagem são projetadas as estimativas de localização dos marcadores sobre um ambiente virtual.

Figura 17 – Ar Track Alvar sobre o RViz



Fonte: Elaborada pelo autor

Como são figuras binárias, e com tamanhos a critério do desenvolvedor, a resolução dos marcadores podem ser 5x5 ou 7x7 *pixels* (ou blocos), resultando em um total de 256 ou 2418 marcadores diferentes, respectivamente. Essa resolução é determinada pelo número de *pixels* no interior da borda do marcador. Cada marcador distinto possui o seu ID de identificação. Vale ressaltar que a maior resolução proporciona uma maior variedade de IDs diferentes, contudo são mais susceptíveis a ruídos e detectados a distâncias mais curtas. Um nó do pacote denominado *createMarker* é responsável pela geração do marcador e pode ser executado pelo seguinte comando:

$ rosrun ar_track_alvar createMarker -f [ID]

O pacote *Ar Track Alvar* é constituído de dois nós de detecção principais: o *Individual Markers* e o *Individual Markers no Kinect*. O primeiro citado executa o algoritmo de detecção utilizando imagens no padrão RGB e infravermelho, com suporte a profundidade, do sensor *Kinect* da empresa *Microsoft*. Já o *Individual Markers no Kinect*, o utilizado nesta dissertação, executa o algoritmo de detecção utilizando apenas imagens

RGB de câmeras convencionais. Vale ressaltar que pacotes que implementam a aquisição de imagens devem ser utilizados de acordo com a câmera em uso e, assim, fornecer o tópico de imagem para o *Ar Track Alvar*. Um exemplo de pacote de aquisição é o *UVC Camera*.

Ambos nós de detecção necessitam de parâmetros de entrada para serem inicializados. Assim, torna-se necessária a execução de *launchs*, Seção 2.1.5.6. O *launch* do *Ar Track Alvar* deve possuir os seguintes parâmetros:

- **marker_size:** dimensão real do marcador utilizado;
- **max_new_marker_error:** tolerância para considerar um marcador visível;
- **max_track_error:** tolerância para determinar quando um marcador é considerado desaparecido;
- **output_frame**: *frame* de referência em que os dados serão apresentados;
- **camera_image:** tópico por onde as imagens são adquiridas;
- **camera_info:** tópico por onde informação da imagem é adquirida. Como dados de calibração e distorção.

Após a detecção, o *Ar Track Alvar* externa os dados adquiridos através do tópico *ar_pose_marker*. O tipo de mensagem encaminhada é o *ar_track_alvar/AlvarMarkers*. Essa mensagem é um vetor de mensagens que indicam os marcadores vistos em um *frame* de imagem. Cada marcador distinto é representado por uma mensagem *ar_track_alvar/Alvar Marker*. Essa mensagem é constituída de um cabeçalho indicando o ID do marcador e o nível de semelhança dele com o estimado, além de dados de posição e orientação relativas. A orientação é informada em *quaternion* (74).

2.6 LÓGICA FUZZY

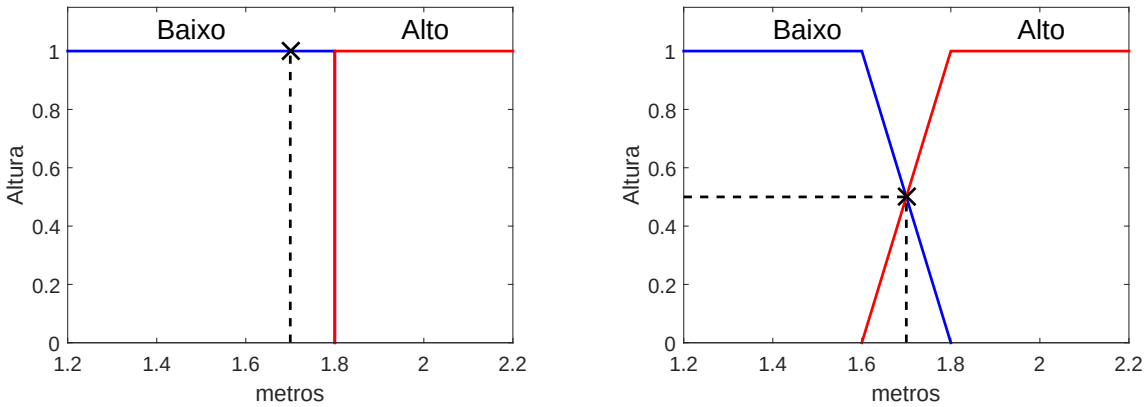
Esta seção explana os principais conceitos sobre a lógica *fuzzy* para a sua aplicação na aterrissagem autônoma de veículos aéreos não tripulados, tema desta dissertação. Os conceitos apresentados são referentes à lógica *fuzzy* do tipo Mamdani. As referências de suporte nesta seção são encontradas em (75) e (76).

2.6.1 Introdução

Teorias de conjuntos clássicas, como a Aristotélica e a Booleana, definem a pertinência de um elemento a um grupo de maneira bem definida, ou seja, o elemento pertence ou não àquele conjunto. Contudo, verifica-se que o raciocínio humano não funciona dessa maneira. Por exemplo, dado um universo de discurso para altura da população em metros, $\mathbf{U} = [1.2, 2.2]$, uma pessoa com estatura de 1,50 é considera baixa, enquanto outra de 1,85 é visto como alta. Porém, uma pessoa com o tamanho de 1,70 pode ser classificada com níveis de pertinência dos grupos alto e baixo, ou seja, não muito alta e nem muito baixa, Figura 18b. Esses níveis de pertinência contrapõem as determinações, bem definidas, dos conjuntos clássicos: verdadeiro/falso, sim/não ou 0/1, Figura 18a.

Essa lógica nebulosa ou incerta, denominada no inglês de *fuzzy*, foi primeiramente modelada em 1965 por Zadeh. Ela permite a modelagem de sistemas incertos e imprecisos e, assim, valores intermediários a 0 e 1, ou níveis de relações, podem ser processados por sistemas computacionais. Consequentemente, problemas podem ser resolvidos o mais próximo da lógica humana, cuja interpretação é realizada através de variáveis linguísticas. Na lógica *fuzzy* variáveis são classificadas de maneira imprecisa, e mais flexível, em subconjuntos como o alto e baixo, ambos ilustrados na Figura 18b.

Figura 18 – Teoria de conjuntos para determinação de altura



(a) Conjuntos Crisp (b) Conjuntos Fuzzy

Fonte: Elaborada pelo autor

2.6.2 Conjuntos *Fuzzy*

Como dito anteriormente, teorias clássicas de conjuntos determinam de maneira bem definida, chamada de *crisp*, a inserção de um elemento a um grupo. Já na teoria de conjuntos *fuzzy*, os elementos são inseridos em diferentes grupos possuindo graus de pertinência equivalentes. Formalizando: dado um universo de discurso **U**, defini-se um grupo A, ou subconjunto A, como o par ordenado do elemento, u, sobre a função pertinência do subconjunto, μ_A (Seção 2.6.3), como descrito abaixo:

$$A = \{(u, \mu_A(u)) | u \in U\} \tag{2.1}$$

A Figura 18, anteriormente apresentada, ilustras dois exemplos de subconjuntos *fuzzy* para o exemplo de altura da população: baixo e alto.

2.6.3 Funções pertinências

As funções pertinências representam o grau que uma variável pertence a um subconjunto *fuzzy*, ou seja, realiza o mapeamento da variável em um valor entre 0 e 1. Uma variável pode possuir graus de pertinência a distintos subconjuntos. A definição de função pertinência do subconjunto A, $\mu_A(u)$, é apresentada abaixo:

$$\mu_A(u) : U \rightarrow [0; 1]; u \in U \tag{2.2}$$

A função pertinência é uma função contínua. Elas podem possuir formas distintas e são utilizadas de acordo com o problema em questão. Exemplos de formas dessas funções são apresentadas na Tabela 2.

2.6.4 Operações de conjuntos *Fuzzy*

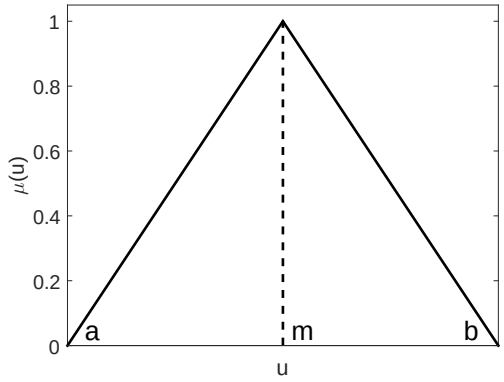
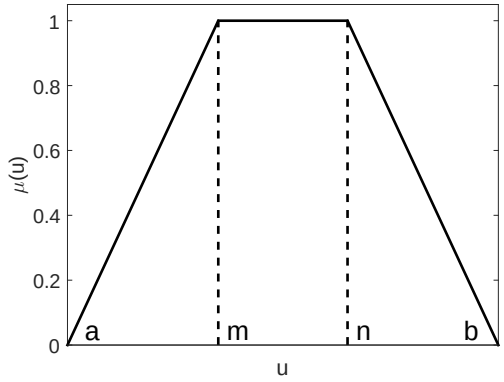
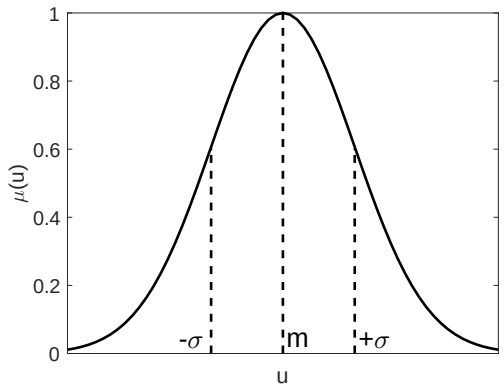
Dois subconjuntos *fuzzy*, A e B, com respectivas funções pertinências $\mu_A(u)$ e $\mu_B(u)$, são representados sobre um universo de discurso **U**. A função pertinência de **U** é definida como $\mu_{\mathbf{U}} = 1$ para qualquer $u \in \mathbf{U}$. Já o conjunto vazio, Θ, complemento de **U**, tem como função pertinência $\mu_\Theta = 0$. Assim, defini-se as três operações típicas dos conjuntos *fuzzy* como segue: União, Interseção e Complemento, Equações 2.3, 2.4 e 2.5 respectivamente:

$$\mu_{(A \cup B)}(u) = \max\{\mu_A(u), \mu_B(u)\}, u \in U \tag{2.3}$$

$$\mu_{(A \cap B)}(u) = \min\{\mu_A(u), \mu_B(u)\}, u \in U \tag{2.4}$$

$$\mu_{A'}(u) = 1 - \mu_A(u), u \in U \tag{2.5}$$

Tabela 2 – Exemplos de funções pertinências

Forma	Descritor
	$\mu(u,a,m,b) = \begin{cases} 0, & se \quad u \leq a \\ \frac{u-a}{m-a}, & se \quad u \in [a,m) \\ \frac{b-u}{b-m}, & se \quad u \in [m,b] \\ 0, & se \quad x \geq b \end{cases}$
	$\mu(u,a,m,n,b) = \begin{cases} 0, & se \quad u < a \\ \frac{u-a}{m-a}, & se \quad u \in [a,m) \\ 1, & se \quad u \in [m,n) \\ \frac{b-u}{b-n} & se \quad u \in [n,b] \\ 0 & e \quad u > b \end{cases}$
	$\mu(u,m,\sigma) = \exp\left(-\frac{(u-m)^2}{\sigma^2}\right)$

Fonte: Elaborada pelo autor

2.6.5 Processo *Fuzzy*

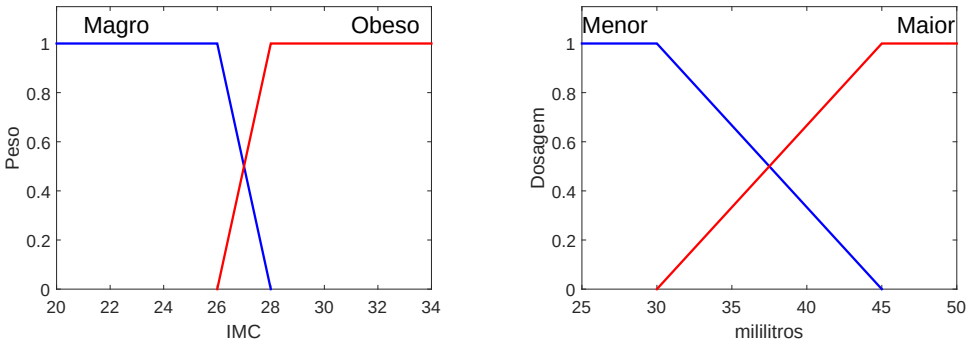
Um Processo *fuzzy* é constituído de etapas. Inicialmente uma variável amostrada, do tipo *crisp*, é traduzida em uma variável *fuzzy* através do processo de fuzzificação. Nesse processo, as variáveis definidas são mapeadas e representadas por variáveis linguísticas dos subconjuntos *fuzzy*. A modelagem realizada pelo especialista para as variáveis de entrada

torna-se importante nessa etapa.

Em seguida, o processo de inferência será o responsável por determinar a ação de saída do *fuzzy*, ou seja, determinar o conjunto de varáveis de saída de ação do sistema. A inferência baseia-se no conjunto de regras projetadas pelo especialista. Contudo, as saídas do sistema são variáveis *fuzzy* que precisam, em muito dos problemas, ser traduzidas para variáveis *crisp*. A esse processo de tradução dá-se o nome de defuzzificação, e é onde o projeto de variáveis de saída torna-se importante.

Nessa seção serão descritos como a fuzzificação é realizada, o procedimento de inferência e a defuzificação. Vale relembrar que o processo *fuzzy* empregado utiliza do sistema de inferência Mamdani. A título de exemplificação, adota-se o problema de determinação de dosagem, em mililitros, de um medicamento para um paciente baseado em seu Índice de Massa Corporal (IMC). As variáveis de entrada e saída, com seu respectivos subconjuntos *fuzzy* são apresentadas na Figura 19.

Figura 19 – Exemplo de caso: determinação de dosagem



(a) Variável Peso (b) Variável Dosagem

Fonte: Elaborada pelo autor

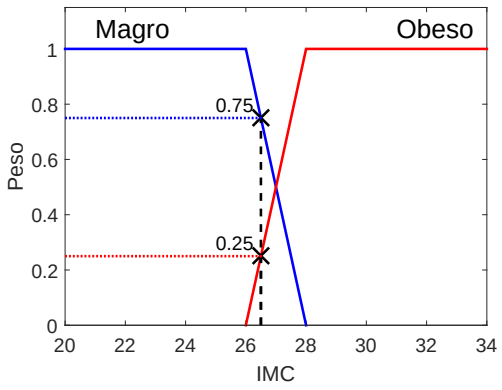
2.6.5.1 Fuzzificação

Dado uma variável *crisp* de entrada, $u \in \mathbf{U}$, essa será descrita de acordo com uma combinação de graus de pertinência e, assim, traduzida em uma variável *fuzzy*. Por exemplo, considerando um paciente do sexo masculino com IMC de 26,5 ($u = 26.5 \in [20, 34]$) o mapeamento desse valor em variável *fuzzy* é descrito por: $\mu_{Magro}(26,5) = 0.75$ e $\mu_{Obeso}(26,5) = 0.25$ (Figura 20).

2.6.5.2 Regras e Inferência

O processo de inferência *fuzzy*, ou seja, a tomada de decisões do sistema *fuzzy*, é baseado em um conjunto de regras descritivas projetadas pelo especialista. As regras são constituídas de variáveis linguísticas antecedentes e consequentes. As antecedentes

Figura 20 – Exemplo de fuzzificação



Fonte: Elaborada pelo autor

são responsáveis pelo disparo de regra, enquanto as consequentes pela determinação da ação a ser executada dado uma regra. A estrutura de regra apresentada abaixo leva em consideração n variáveis *fuzzy* de entrada, x_n, e m variáveis *fuzzy* de saída, y_m:

SE x_1 é <antecedente> **E/OU** x_2 é <antecedente> ...
ENTÃO y_1 é <consequente> **E/OU** y_2 é <consequente> ...

Variáveis de entrada combinadas entre si através das conjunções “E” e “OU”, que usualmente representam as operações descritas pelas Equações 2.3 e 2.4 respectivamente, também são responsáveis por ativações de regras.

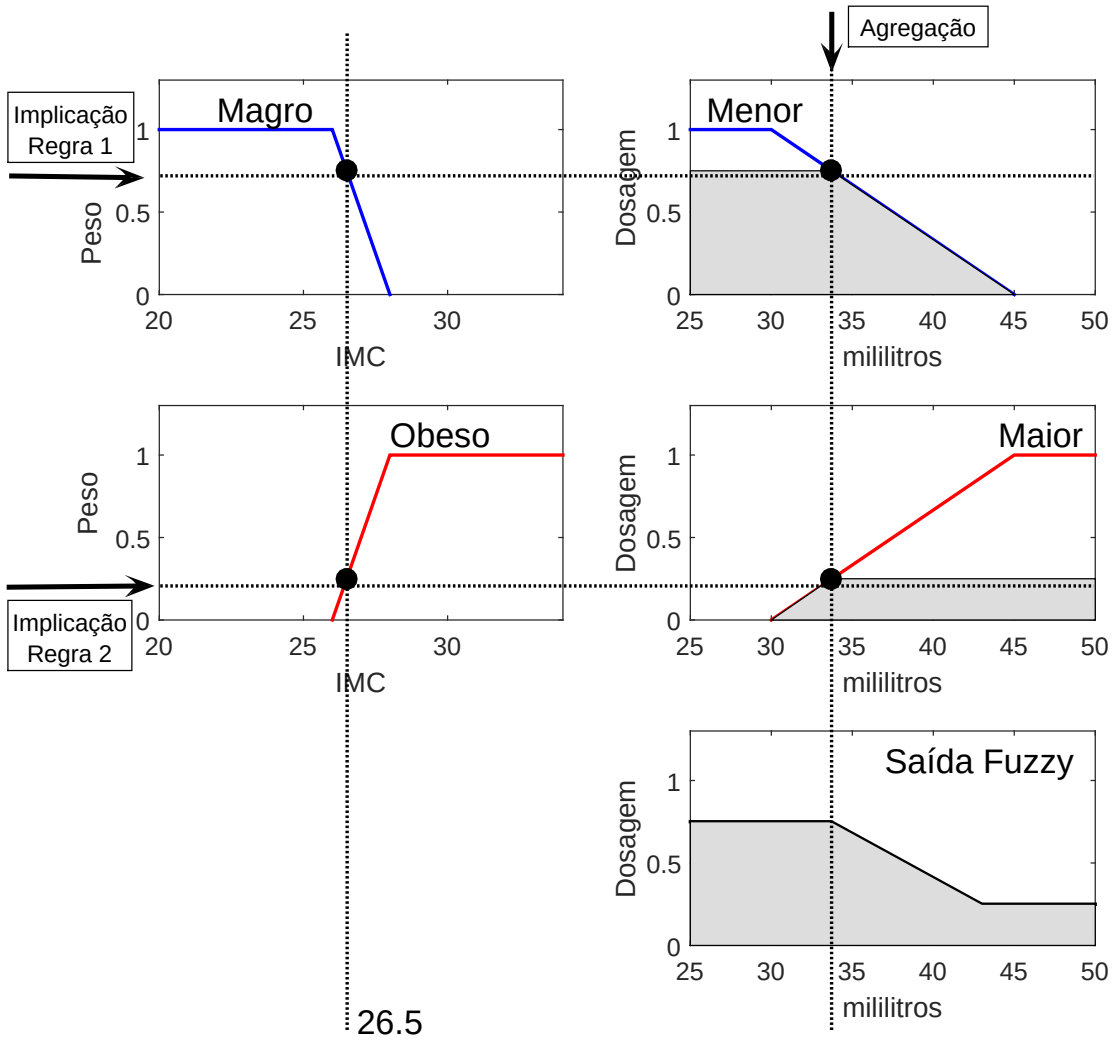
A etapa de associação das variáveis de entrada com as de saída, dado uma regra, é denominada de implicação. Após a implicação, cada regra apresentará uma região resultante. Todas regiões geradas pela lista de regras ativadas são combinadas na fase de agregação, resultando na variável *fuzzy* de saída.

Para o problema de dosagem de remédio a um paciente, o projetista pode determinar o conjunto de regras conforme abaixo:

SE paciente é MAGRO **ENTÃO** dosagem é MENOR
SE paciente é OBESO **ENTÃO** dosagem é MAIOR

Assim o procedimento de inferência por lógica fuzzy Mamdani para o paciente cujo IMC é de 26,5 ($u = 26,5$), é ilustrado na Figura 21. As linhas horizontais da Figura 21 representam a etapa de implicação através do procedimento de mínimo, ou seja, a área resultante será determinada abaixo da linha de corte. Já a agregação, representada na Figura 21, é realizada pela operação de máximo, Equação 2.3.

Figura 21 – Exemplo de inferência



Fonte: Elaborada pelo autor

2.6.5.3 Defuzificação

De posse da variável de saída *fuzzy*, a última etapa do processo é transformá-la em um valor *crisp* para que sistemas de atuações possam executar a ação resultante. Na lógica Mamdani, a variável *fuzzy* é interpretada como uma região resultada do processo de agregação. Logo, para determinar o valor quantitativo dessa área, técnicas como cálculo de centroide, média dos máximos e centro dos máximos podem ser utilizadas .

A técnica centroide, utilizada na presente dissertação para o procedimento de aterrissagem, é descrita pela Equação 2.6:

$$c = \frac{\int_X x \cdot S(x)dx}{\int_X S(x)dx} \tag{2.6}$$

em que $S(x)$ é a função que descreve a área da figura geométrica resultante dado um ponto $x \in X$.

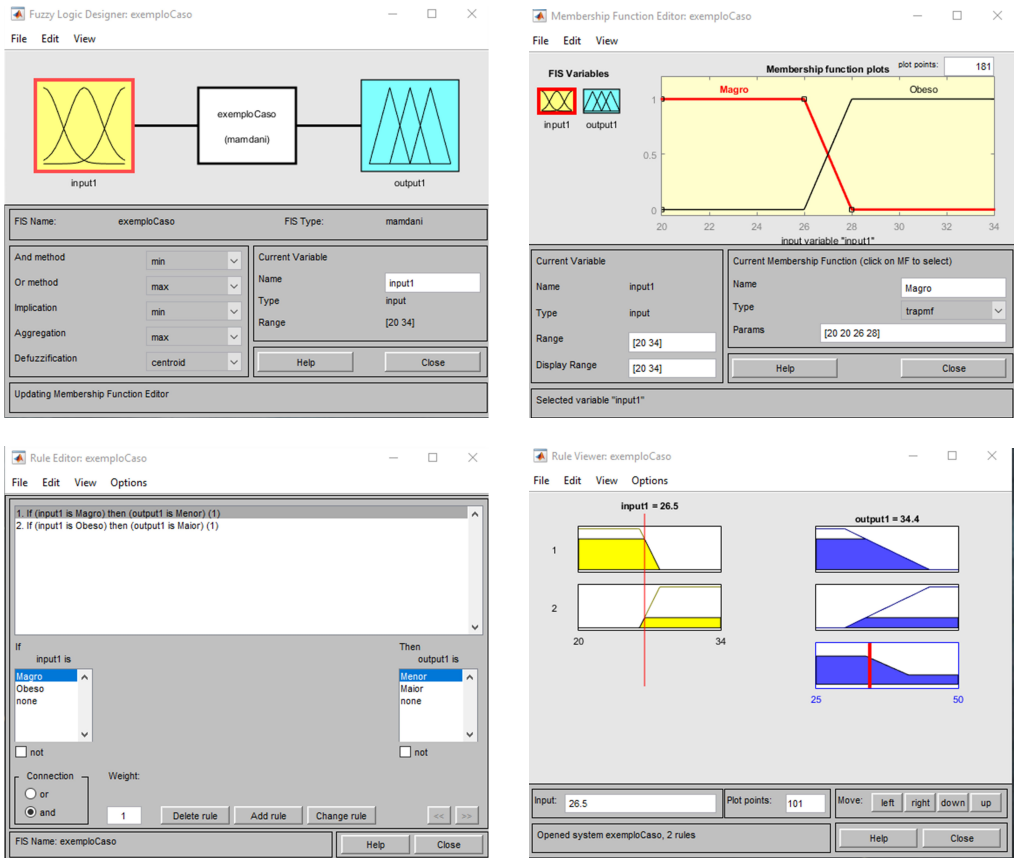
Retornando ao exemplo da dosagem de medicamento, a área resultante que descreve a saída *fuzzy*, apresentada na Figura 21, possui como centroide: $c = 34.44ml$. Portanto, a paciente cujo IMC é de $26,5$ deverá tomar $34,44ml$ do medicamento.

2.6.6 *Matlab*® *Fuzzy Toolbox*

O *software Matlab*® possui uma *toolbox* que facilita a concepção e projeto de processos *fuzzy*, denominada *Fuzzy Toolbox*, Figura 22. Sendo assim, essa ferramenta é utilizada na proposta desta dissertação.

A inserção de variáveis de entrada e de saída é realizada de maneira gráfica e a listas de regras são descritas pelo projetista. Dessa maneira modificações podem ser realizadas de forma mais rápida. Ainda é possível verificar os processos de implicação, agregação e fuzzificação, além de determinar suas operações.

Figura 22 – *Fuzzy Toolbox Matlab*®



Fonte: Elaborada pelo autor

2.6.7 Biblioteca *C++ Fuzzylite*

O *Fuzzylite*(77) é uma biblioteca, *open source*, para implementação de lógica *fuzzy* orientada a objetos. A biblioteca é disponível para diferentes plataformas como *Java* e *C++*. Essa ferramenta é utilizada nesta dissertação para implementação do algoritmo de aterrissagem *fuzzy* embarcado em *C++*.

Incluída a biblioteca *fuzzylite*, o objeto que representa a lógica *fuzzy* é instanciado pelo construtor *Engine*:

```
3 #include<Headers/fl.h>
4 using namespace fl;
5
6 Engine* engine = new Engine;
7 engine->setName("exemploCaso");
```

Em seguida são definidas as variáveis de entrada. Cada objeto do tipo *InputVariables* possui como propriedades seu nome e *range*, além de métodos relativos a inclusão de funções membros, ou subconjuntos *fuzzy*. Ao término da declaração, a variável de entrada é inserida ao *engine* da lógica fuzzy.

```
7  InputVariable* inputVariable = new InputVariable;
8  inputVariable->setEnabled(true);
9  inputVariable->setName("IMC");
10 inputVariable->setRange(20.000, 34.000);
11 inputVariable->addTerm(new Trapezoid("Magro", 20.000, 20.000, 26.000,
       28.000));
12 inputVariable->addTerm(new Trapezoid("Obeso", 26.000, 28.000, 34.000,
       43.000));
13 engine->addInputVariable(inputVariable);
```

O mesmo procedimento de definição das variáveis de entrada é realizada para as variáveis de saída através da classe *OutputVariables*. Contudo, esse tipo de objeto também possui métodos que determinam o tipo de agregação e defuzificação:

```
15 OutputVariable* outputVariable = new OutputVariable;
16 outputVariable->setEnabled(true);
17 outputVariable->setName("ML");
18 outputVariable->setRange(25.000, 50.000);
19 outputVariable->fuzzyOutput()->setAccumulation(new Maximum);
20 outputVariable->setDefuzzifier(new Centroid(200));
21 outputVariable->setDefaultValue(fl::nan);
22 outputVariable->setLockPreviousOutputValue(false);
23 outputVariable->setLockOutputValueInRange(false);
24 outputVariable->addTerm(new Trapezoid("Menor", 25.000, 25.000, 30.000,
       45.000));
25 outputVariable->addTerm(new Trapezoid("Maior", 30.000, 45.000, 50.000,
       50.000));
```

```
26 engine->addOutputVariable(outputVariable);
```

A lista de regras que compõem o processo de inferência é instanciada pela classe *RuleBock*. Cada regra é escrita em forma de *string* de acordo com os nome das variáveis de entrada e saídas projetadas. Cada regra é inserida à lista de regras que, por sua vez, é inserida à *engine fuzzy*. Também são determinados os tipos de operações e a implicação utilizada:

```
14 ruleBlock->setName("");
15 ruleBlock->setConjunction(new Minimum);
16 ruleBlock->setDisjunction(new Maximum);
17 ruleBlock->setActivation(new Minimum);
18 ruleBlock->addRule(fl::Rule::parse("if IMC is Magro then ML is Menor",
       engine));
19 ruleBlock->addRule(fl::Rule::parse("if IMC is Obeso then ML is Maior",
       engine));
```

Já codificada, a lógica *fuzzy* é utilizada com o comando abaixo:

```
21 IMC->setInputValue(26.5);
22 engine->process();
23 cout<< "Mililitros necessarios:" << ML->getOutputValue() << endl;
```

2.7 REDE NEURAL ARTIFICIAL

Nesta seção são abordados conceitos inerentes à técnica de inteligência artificial por RNA. Essa técnica é utilizada na realização da aterrissagem por visão computacional de VANTs, proposta da presente dissertação. É abordado também a RNA perceptron de múltiplas camadas e o método *backpropagation* de treinamento. Essa seção é fundamentada a partir das referências (78–82), onde informações complementares a respeito de RNAs podem ser encontradas.

2.7.1 Introdução

O cérebro humano, assim como os de outros animais, motivaram o estudos de diferentes técnicas e algoritmos para que computadores emulassem seu comportamento. O sistema nervoso é capaz de interpretar e processar informações e executar tarefas complexas, não-lineares e de maneira paralela. Como descrito em (78), um exemplo de tarefa seria a capacidade de reconhecimento perceptivo do ser humano através da visão. O sistema nervoso é capaz de lidar com alto nível de informação e identificar uma face familiar em um ambiente desconhecido. Toda essa tomada de ação é realizada em milésimos de segundo. Outro ponto fundamental é a capacidade de adaptação, aprendizagem e generalização do cérebro, que por meio de uma rede de neurônios realiza tais façanhas. Dado esse potencial, McCulloch, um psiquiatra e neuroanatomista, e Pitts, um matemático, propuseram em 1943 o primeiro modelo matemático de um neurônio artificial, iniciando assim os estudos de RNAs. A ideia é a representação do sistema nervoso através de combinação de neurônios, ou rede de neurônios, visando sua aplicação em sistemas computacionais.

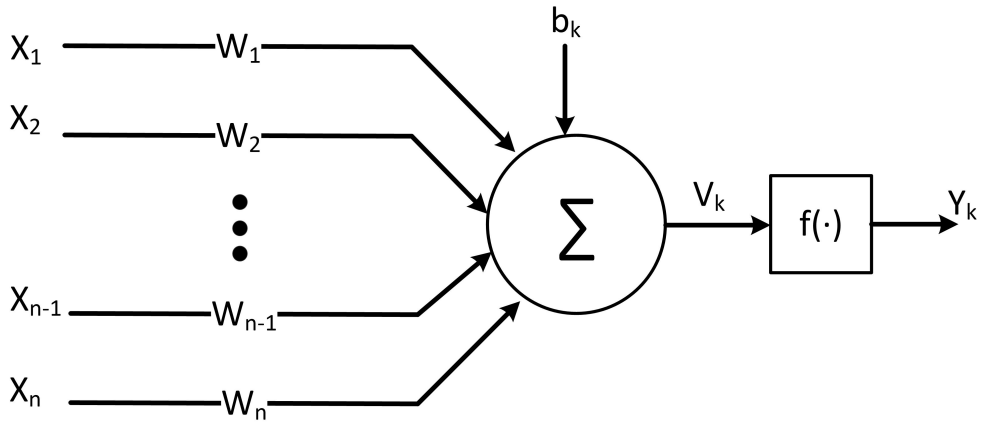
Segundo (78), uma rede neural artificial pode ser definida como um processador paralelamente distribuído composto de unidades processadoras simples, chamados de neurônios. Desta forma, a RNA teria a capacidade de armazenar um conhecimento experimental e torná-lo disponível. Já como explanado em (79), a RNA é capaz de resolver problemas e generalizá-los. Esse conhecimento seria armazenado por meio de pesos sinápticos adquiridos e ajustados por métodos de aprendizagem. Desta maneira, uma RNA se assemelharia ao sistema nervoso real.

2.7.2 O neurônio artificial

Um neurônio, em RNA, é uma unidade processadora que compõe uma rede distribuída. A representação dessa unidade é apresentada na Figura 23. Baseando-se nessa figura, pode-se verificar que o neurônio é constituído de três partes distintas: as sinapses, o somador e a função de ativação, abordados abaixo:

- **Sinapses:** A sinapse é a conexão entre cada neurônio. Essas conexões são representadas por pesos sinápticos, W_n. Os pesos são responsáveis pela inteligência adquirida

Figura 23 – Representação de um neurônio



Fonte: Elaborada pelo autor

durante um processo de treinamento da RNA. Já o *bias*, b_k é considerado um peso que ajusta a entrada V_k na função de ativação, $f(\cdot)$;

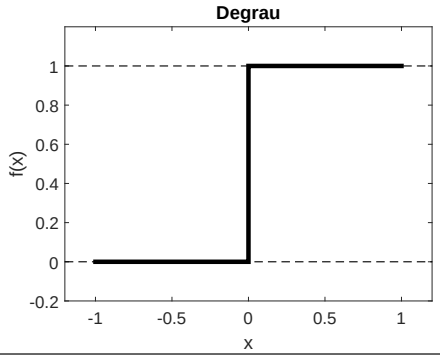
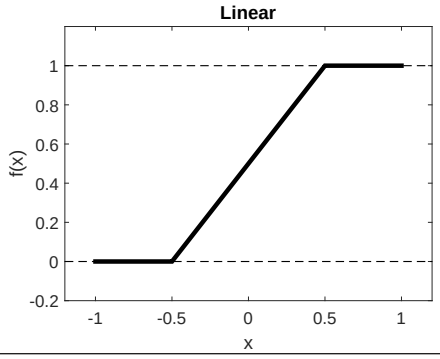
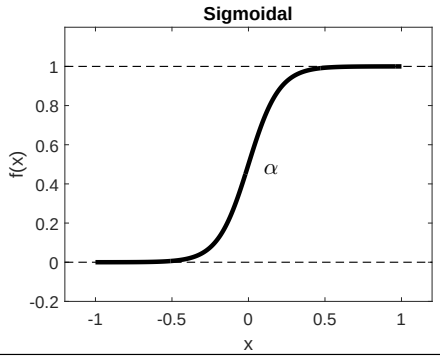
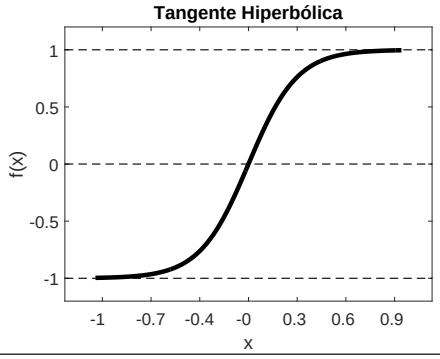
- **Somador:** O somador, $\sum$, é o responsável pela combinação linear dos valores de entradas ponderados pelos pesos sinápticos;
- **Função de ativação:** A função de ativação, $f(\cdot)$, é a encarregada de limitar ou remapear o valor de saída do somador e assim gerar o valor de saída, Y_k. Três tipos de funções de ativação usuais podem ser utilizadas. Esses tipos são apresentados na Tabela 3. Vale ressaltar que funções do tipo degrau e linear podem ser limitadas entre $[0, 1]$ ou $[-1, 1]$.

2.7.3 RNA perceptron de múltiplas camadas

As RNAs podem se caracterizar de acordo com o seu método de treinamento: supervisionado ou não. Assim, para o caso de treinamento supervisionado, utilizado nesta dissertação, são apresentadas à RNA entradas e suas respectivas saídas esperadas, ou seja, um supervisionador, ou professor(78), é inserido no processo de aprendizagem. Desta forma, algoritmos de treinamento visarão reduzir o erro da saída estimada com a desejada e, assim ajustar os pesos sinápticos por iterações, também dominadas de épocas.

Alternativamente, a caracterização de RNAs pode ser realizada quanto a sua estrutura e topologia. Alguns exemplos são a rede perceptron de múltiplas camadas, a Adaline e a Hopfield (82). Verifica-se que a RNA perceptron de múltiplas camadas (MLP) é aplicável ao problema de aterrissagem, sendo a abordada na presente dissertação. Um motivo adicional é simplicidade da MLP em relação a outras topologias.

Tabela 3 – Exemplos de funções de ativação

Forma	Descritor
	$f(x) = \begin{cases} 1, & se \quad x \geq 0 \\ 0, & se \quad x < 0 \end{cases}$
	$f(x) = \begin{cases} 0, & se \quad x \leq -0,5 \\ x, & se \quad -0,5 < x < 0,5 \\ 1, & se \quad x > 0,5 \end{cases}$
	$f(x) = \frac{1}{1+\exp(-\alpha \cdot x)}$
	$f(x) = \tanh(x)$

Fonte: Elaborada pelo autor

2.7.3.1 *Backpropagation*

O *backpropagation* é um método de treinamento de RNAs supervisionado. Por meio de uma etapa direta, ou *forward*, sinais de entrada são propagados ao longo da rede estimando uma saída. Baseado no erro entre a saída estimada e a saída desejada, a etapa *backward* realiza os ajustes dos pesos sinápticos. Esse processo é realizado para todos os dados do conjunto de treinamento finalizando uma época. Épocas são repetidas iterativamente até que um critério de parada seja atingido.

Segundo (78), o algoritmo de *backpropagation* por gradiente descendente pode ser divididos nas seguintes etapas:

- **Inicialização:** Inicializa-se os ganhos da RNA com valores aleatórios de pesos sinápticos;
- **Definição do conjunto de treinamento:** Determina-se o conjunto de treinamento, ou seja, o par ordenado de dados que representam as entradas, $\mathbf{X} = [x_1, x_2, ...]$, com as saídas desejadas $\mathbf{D} = [d_1, d_2, ...]$. Cada par ordenado deve ser apresentado para execução das etapas *Forward* e *Backward*. Ao término do conjunto de treinamento uma nova época é realizada até a convergência do treinamento;
- **Fase *forward*:** A etapa *forward* estima a saída propagando o sinal de entrada ao longo da rede. Determinando-se o número de camadas da RNA como $k = [1, 2, ..., K]$ e o número de neurônios por cada camada de $i = [1, 2, ..., I(k)]$, a saída de cada neurônio na iteração n é descrita na Equação 2.7:

$$y_i^k(n) = f(v_i^k(n)) \tag{2.7}$$

Onde $f(\cdot)$ é a função de ativação aplicada sobre a saída do somador, $v_i^k(n)$, Equação 2.8:

$$v_i^k(n) = \sum_{j=1}^{I(k-1)} w_{i,j}^k(n) y_j^{(k-1)}(n) + b_i(n) \tag{2.8}$$

cujos pesos sinápticos por iteração são representados por $w_{i,j}^k(n)$ e $j = [1, 2, ..., I(k-1)]$. O *bias* de cada neurônio é representado por $b_i(n)$. Vale ressaltar que y_j^1 representa os sinais da camada de entrada, ou x_j. O valor de saída estimada da rede para o neurônio i da última camada é representado por $o_i = y_i^K$. Logo o sinal de erro é calculado como a Equação 2.9:

$$e_i(n) = d_i - o_i(n) \tag{2.9}$$

- **Fase *backward*:** Na fase *backward* o algoritmo de otimização é executado atualizando os ganhos sinápticos. O algoritmo do gradiente descendente, baseado na minimização do erro quadrático médio, Equação 2.10, estima o gradiente local pelas Equações 2.11, para última camada, e 2.12, para as demais camadas.

$$EQM(n) = \frac{1}{2} \sum_{i=1}^{I(K)} e_i^2(n) \tag{2.10}$$

$$\delta_i^K(n) = e_i^K(n) f'(v_i^K(n)) \tag{2.11}$$

$$\delta_j^{k-1}(n) = f'(v_j^{k-1}(n)) \sum_{i=1}^{I(k)} \delta_i^k w_{i,j}^k \tag{2.12}$$

Assim a atualização das variáveis é realizada pela Equação 2.13:

$$w_{i,j}^k(n+1) = w_{i,j}^k(n) - \alpha \delta_i^k(n) y_j^{k-1}(n) \tag{2.13}$$

onde α é o coeficiente de aprendizagem.

O algoritmo de otimização por gradiente descendente, apesar de bastante difundido, possui uma convergência lenta para o problema de minimização de uma função custo. O desempenho desse algoritmo está ligado à determinação da taxa de aprendizagem. Essa taxa deve-se adequar ao gradiente: gradiente maior e menor requisitariam taxas maiores e menores, respectivamente. Logo, devido à dinâmica da busca pela solução de mínimo, essa taxa fixa resultaria em problemas. Além da lentidão do treinamento, a taxa influenciará na convergência em mínimos locais e o efeito *zig-zag* sobre a região de convergência (78). Logo, métodos de otimização baseados em Gauss-Newton, como o de Levenberg–Marquardt, objetivam melhorar o desempenho de otimização de uma função custo em comparação ao gradiente descendente para o treinamento por *backpropagation*. Informações complementares e demonstrações desses métodos podem ser verificados em (80, 81).

3 METODOLOGIA PROPOSTA

Esse capítulo aborda a proposta da presente dissertação, onde todo o procedimento para execução da aterrissagem autônoma por visão computacional é apresentado. Dividido em seções, o capítulo define o problema, sua modelagem e descrição. Além disso, apresenta a estrutura do *firmware* de aterrissagem, o desenvolvimento do local de pouso, a modelagem da lógica *fuzzy* supervisionadora e o processo de elaboração da rede neural artificial. São abordados também o conceito de *Software in the Loop* (SITL) e o protótipo de VANT utilizado.

3.1 DEFINIÇÃO DO PROBLEMA

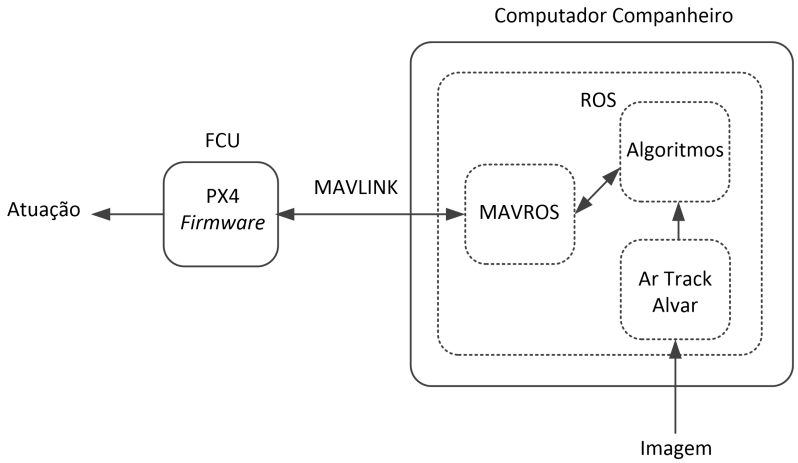
De posse de um VANT com uma câmera convencional de computador, ou *webcam*, acoplada em sua parte inferior, deseja-se criar um *firmware* para o pouso autônomo da aeronave. A câmera, com foco para baixo, indentifica marcadores artificiais predispostos no ambiente. Esses marcadores constituem o local de pouso que pode estar parado ou em movimento. O *firmware* é o conjunto de códigos, algoritmos e técnicas necessárias para realização do procedimento.

3.2 *FIRMWARE* PARA ATERRISSAGEM

O *firmware* de aterrissagem proposto, engloba o conjunto de códigos e estrutura da inteligência embarcada para a realização do pouso autônomo de um VANT por visão computacional. Inclui-se também considerações de projeto e de montagem, pois visa-se a fácil replicação e aplicação em sistemas reais. O *firmware* é embarcado em duas entidades processadoras: o computador companheiro, ou *companion computer*, e a *Flight Control Unit* (FCU). O computador companheiro é embarcado com os códigos relativos aos processamentos de alto nível, como algoritmos para o pouso e visão computacional. Já a FCU é a responsável pelo controle de baixo nível da aeronave. A Figura 24 ilustra as entidades de processamento utilizadas.

Inicialmente defini-se as plataformas e ferramentas que deseja-se utilizar para o desenvolvimento do *firmware*. Como base para gerenciamento de pacotes, processos e fornecimento de ferramentas e bibliotecas, o ROS é o *framework* utilizado. Outro ponto importante do ROS, como descrito na Seção 2.1, é a difusão e o constante uso pela comunidade científica. Torna-se necessário também a determinação de um *firmware* de controle aéreo compatível embarcado na placa controladora de voo. Esse *firmware* é o responsável pelo controle de baixo nível da aeronave, que através de sinais de referências, como ângulos ou posicionamento, realiza a geração de sinais de atuação sobre os motores. Ele também tem a funcionalidade de empacotar dados de sensores e disponibilizá-los. Logo, o *firmware PX4* (Seção 2.3) é o selecionado devido às suas características de

Figura 24 – Entidades processadoras do *firmware* de aterrissagem



Fonte: Elaborada pelo autor

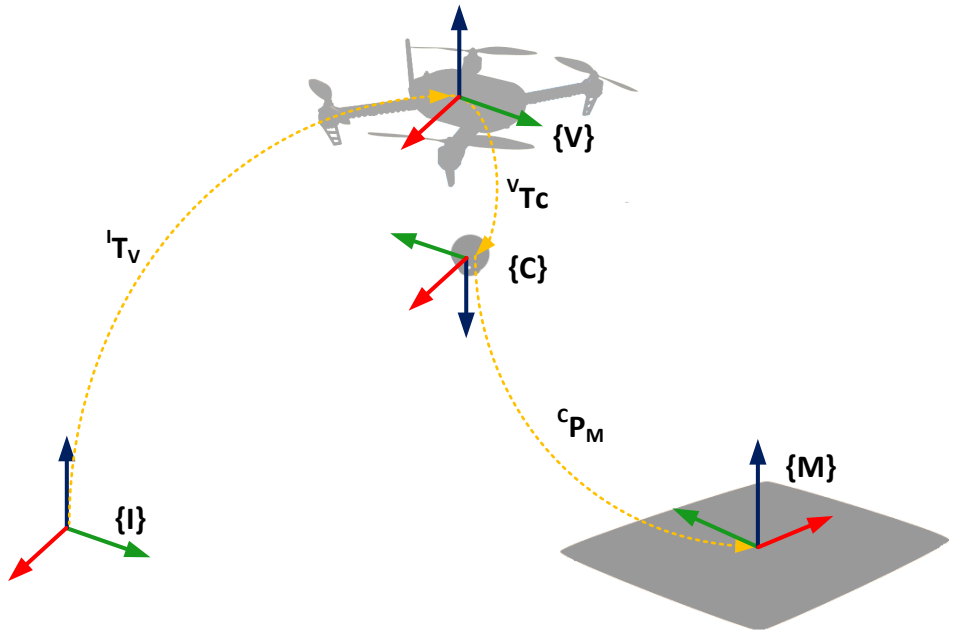
desenvolvimento aberto e multiprocessamento. Em adição, o *PX4* estabelece o protocolo MAVLINK (Seção 2.2), possibilitando a utilização de computadores companheiros e, assim, estabelecer trocas de informações. O computador companheiro é de grande importância, pois nele é embarcado os códigos de detecção de visão computacional, o pacote *Ar Track Alvar*, os algoritmos de pouso propostos, *fuzzy* e RNA, e o pacote MAVROS, Seção 2.4. A Figura 24 também ilustra as ferramentas utilizadas sobre as entidades processadoras. Os equipamentos utilizados e o protótipo proposto são apresentados na Seção 3.9.

3.3 SISTEMA DE COORDENADAS PARA ATERRISSAGEM

A Figura 25 descreve o problema de aterrissagem abordado. Nela, o VANT, a câmera e o marcador de pouso são indicados por {V},{C} e {M}, respectivamente. Nessa figura são indicadas todas referências, ou *frames*, adotadas no problema. Os eixos X,Y e Z são representados pelas cores vermelha, verde e azul, respectivamente (padrão RGB).

A definição dos *frames* vão de acordo com o sistema usado, que os utilizam da maneira indicada. Toda a atuação realizada sobre o VANT, velocidade ou deslocamento, possui como referência o *frame* inercial, indicado por {I}. Esse *frame* é criado no momento que a aeronave é ligada em solo. Isso é um padrão do pacote *MAVROS* utilizado. Já os algoritmos propostos para a execução da aterrissagem possuem como referência o corpo do VANT. Essa abordagem facilita o desenvolvimento dos algoritmos. Em contrapartida, a câmera identifica os marcadores com referência à sua imagem. Logo, percebe-se que cada *frame* possui sua disposição, tornando-se necessário a utilização de transformações homogêneas como modelagem do problema (69). Cada matriz de transformação homogênea é também representada na Figura 25, e são indicadas por ${}^{\mathbf{A}}\mathbf{T}_{\mathbf{B}}$. Essa nomenclatura indica a representação do *frame* B em relação ao *frame* A. A Equação 3.1 demonstra essa transformação (69):

Figura 25 – Sistema de coordenadas para o problema



Fonte: Elaborada pelo autor

$$ {}^{\mathbf{A}}\mathbf{T}_{\mathbf{B}} = \begin{bmatrix} {}^{\mathbf{A}}\mathbf{R}_{\mathbf{B}} & \mathbf{t} \\ \mathbf{0}_{1x3} & 1 \end{bmatrix}_{4x4} \tag{3.1} $$

em que a matriz de rotação, ${}^{\mathbf{A}}\mathbf{R}_{\mathbf{B}}$, é dada pela combinação das matrizes de rotações individuais para cada eixo dado um ângulo de *roll* (θ_x), *pitch* (θ_y) e *yaw* (θ_z) (69):

$$ {}^{\mathbf{A}}\mathbf{R}_{\mathbf{B}} = \mathbf{R}_{\mathbf{x}} \cdot \mathbf{R}_{\mathbf{y}} \cdot \mathbf{R}_{\mathbf{z}} \tag{3.2} $$

dado que:

$$ \begin{aligned} \mathbf{R}_{\mathbf{x}} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} \\ \mathbf{R}_{\mathbf{y}} &= \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \\ \mathbf{R}_{\mathbf{z}} &= \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{3.3} $$

O vetor $\mathbf{t}$ representa a translação tridimensional. Sua representação é indicada abaixo (69):

$$\mathbf{t} = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} \tag{3.4}$$

A matriz ${}^{\mathbf{I}}\mathbf{T}_{\mathbf{V}}$ indica a posição e orientação do VANT em relação ao *frame* inercial. Essa matriz é atualizada constantemente por leituras de IMU e GPS da aeronave. Já ${}^{\mathbf{V}}\mathbf{T}_{\mathbf{C}}$ é uma matriz fixa, que indica a posição relativa da câmera em relação ao corpo da aeronave ou centro de massa. A visão computacional utilizada nesse trabalho utiliza como referência o padrão para eixo de imagens, com eixo X para direita, Y para baixo e Z para "fora" da câmera. Assim, o marcador terá uma posição e orientação relativa à câmera. Considerando o marcador como um ponto de pouso, pode-se indicar a sua posição e orientação pela matriz ${}^{\mathbf{C}}\mathbf{P}_{\mathbf{M}}$ e, como já foi informado anteriormente, esse dado deverá ser transformado para ser processado pelos algoritmos de pouso que possuem sua referência ao corpo do veículo. Portanto, quando o local, ou ponto, de aterrissagem for abordado nesta dissertação considera-se o dado como ${}^{\mathbf{V}}\mathbf{P}_{\mathbf{M}}$, obtido pela seguinte equação:

$${}^{\mathbf{V}}\mathbf{P}_{\mathbf{M}} = {}^{\mathbf{V}}\mathbf{T}_{\mathbf{C}} \cdot {}^{\mathbf{C}}\mathbf{P}_{\mathbf{M}} \tag{3.5}$$

Os valores de atuação estimados pelos algoritmos de pouso, que podem ser deslocamento e velocidade, estão sobre a referência do corpo da aeronave e são indicados por um ponto ${}^{\mathbf{V}}\mathbf{A}$. Como a atuação deve ser referenciada ao *frame* {I}, os valores de atuação estimados nesta dissertação é indicado por ${}^{\mathbf{I}}\mathbf{A}$, obtido por:

$${}^{\mathbf{I}}\mathbf{A} = {}^{\mathbf{I}}\mathbf{T}_{\mathbf{V}} \cdot {}^{\mathbf{V}}\mathbf{A} \tag{3.6}$$

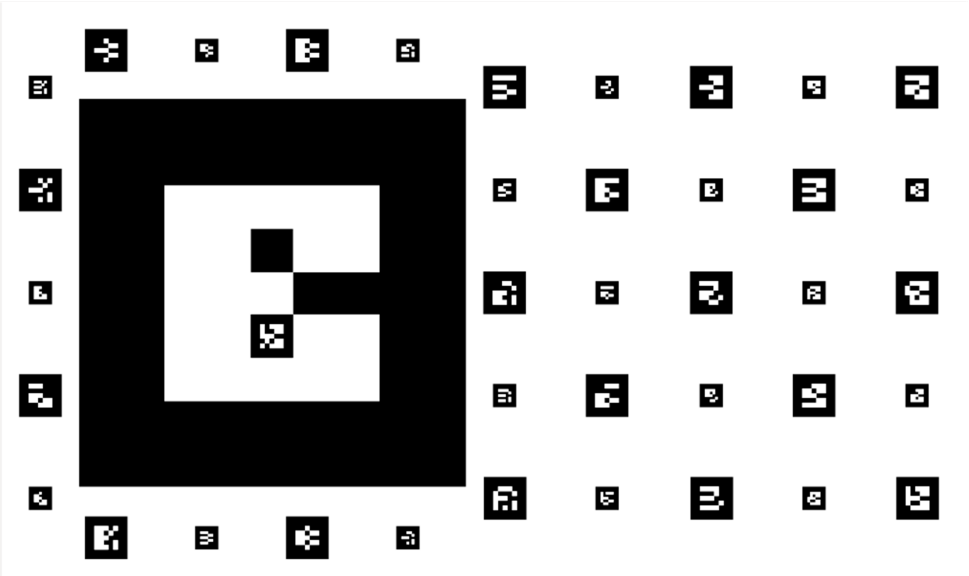
3.4 PONTO DE ATERRISSAGEM

O ponto de aterrissagem, ${}^{\mathbf{V}}\mathbf{P}_{\mathbf{M}}$ (Seção 3.3), é estimado pela detecção do marcador de realidade aumentada. Esse marcador pode ser impresso ou pintado, e pode possuir diferentes tamanhos e IDs, como abordado na Seção 2.5. Devido à testes preliminares, verifica-se que a determinação e confecção do marcador de pouso torna-se importante para um procedimento de aterrissagem satisfatório. Uma abordagem, também verificada em (39, 40), é a combinação de diferentes marcadores para determinação do local de pouso. Essa estratégia visa aumentar a robustez da detecção por visão. Características do projeto de confecção com esse mesmo intuito, e adotadas nesta dissertação, são listadas abaixo:

- **Combinação de diferentes marcadores:** A combinação permite que o algoritmo tenha redundância para determinação do local de pouso. Isso contorna possíveis problemas cuja visão sobre um determinado marcador é comprometida. Exemplos são diferença de luminosidade, saturação da imagem sobre uma região, ou imagem não completamente visível do marcador;
- **Marcadores com tamanhos distintos:** Os tamanhos permitem aumentar o alcance e precisão da visão. Marcadores maiores são vistos a longas distâncias enquanto os menores são vistos a distâncias mais curtas;
- **Predisposição dos marcadores sem interferência mútua:** Os marcadores não devem se influenciar, ou seja, marcadores muito próximos ou sobrepostos podem comprometer a detecção. Para minimizar os efeitos negativos dessa abordagem, ou os marcadores devem se espaçar para gerar o contraste de suas bordas, ou devem possuir tamanhos discrepantes quando sobrepostos;
- **Material com pouca reflexibilidade:** Altas reflexões podem saturar a imagem adquirida pela câmera, atrapalhando a detecção.

Seguindo as características citadas acima, o local de pouso proposto é confeccionado no software *Corel Draw*® e possui o formato retangular com a dimensão de 2,50 x 1,50m. Ele é indicado na Figura 26 e é constituído por 40 marcadores distintos com formato quadrado, de resolução de 5 *pixels*, vide Seção 2.5. Três dimensões são adotadas, 1,0 x 1,0m, 11,0 x 11,0cm e 6,0 x 6,0cm para 1, 20 e 19 marcadores respectivamente.

Figura 26 – Marcador que define o local de pouso



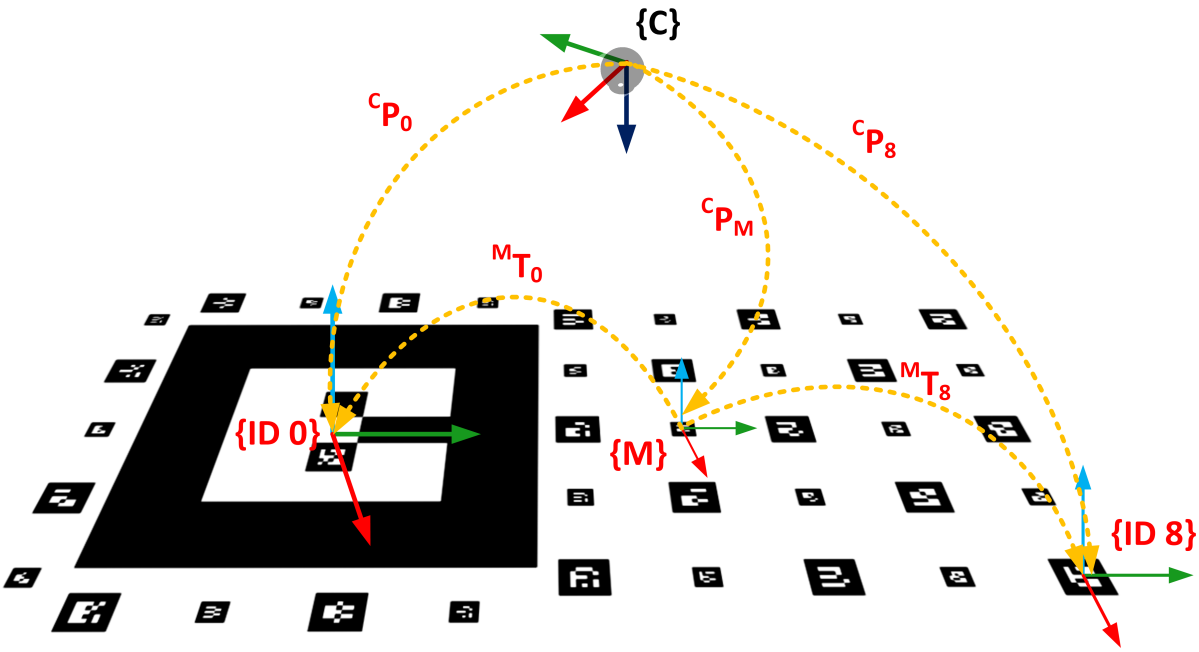
Fonte: Elaborada pelo autor

Todos os marcadores devem determinar o mesmo local de pouso, ${}^{\mathbf{C}}\mathbf{P_M}$, e assim direcionar o VANT para a região central de aterrissagem após o processamento indicado na Equação 3.5. O ID 34 é eleito como marcador central e, portanto, sua identificação determinará ${}^{\mathbf{C}}\mathbf{P_M}$. A detecção dos outros IDs é dada por ${}^{\mathbf{C}}\mathbf{P_{id}}$. A disposição de todos os outros IDs em relação ao ID central é conhecida a priori devido ao projeto do local de pouso. Assim ${}^{\mathbf{M}}\mathbf{T_{id}}$ de cada ID é conhecida. Em posse desses dados é possível estimar ${}^{\mathbf{C}}\mathbf{P_M}$ mesmo quando o ID central não está visível, através do seguinte processamento de transformação homogênea:

$$ {}^{\mathbf{C}}\mathbf{P_M} = ({}^{\mathbf{M}}\mathbf{T_{id}} \cdot {}^{\mathbf{id}}\mathbf{P_C})^{-\mathbf{1}} \tag{3.7} $$

A Figura 27 exemplifica esse conceito. Nela é indicado as matrizes de transformação homogênea dos ID 0 e 8, ${}^{\mathbf{M}}\mathbf{T_0}$ e ${}^{\mathbf{M}}\mathbf{T_8}$, em relação ao ID central, ou {M}. Dessa forma, é possível estimar a posição {M}, ${}^{\mathbf{C}}\mathbf{P_M}$, apenas com a detecção de ${}^{\mathbf{C}}\mathbf{P_0}$ e/ou ${}^{\mathbf{C}}\mathbf{P_8}$.

Figura 27 – Exemplo de transformações dos marcadores que constituem o local de pouso



Fonte: Elaborada pelo autor

Determinada a posição do local de pouso, ${}^{\mathbf{C}}\mathbf{P_M}$, torna-se necessária a sua filtragem para que seja aplicada nos algoritmos de aterrissagem. O filtro é necessário para evitar que valores errôneos afetem o desempenho da aterrissagem. Esses erros são inerentes ao *Ar Track Alvar*, que durante o chaveamento de detecção de diferentes IDs apresenta a probabilidade de distúrbios de transição. Vale ressaltar que esse distúrbios são propagados, e agravados pelas multiplicações, após as transformações homogêneas discutidas na presente seção.

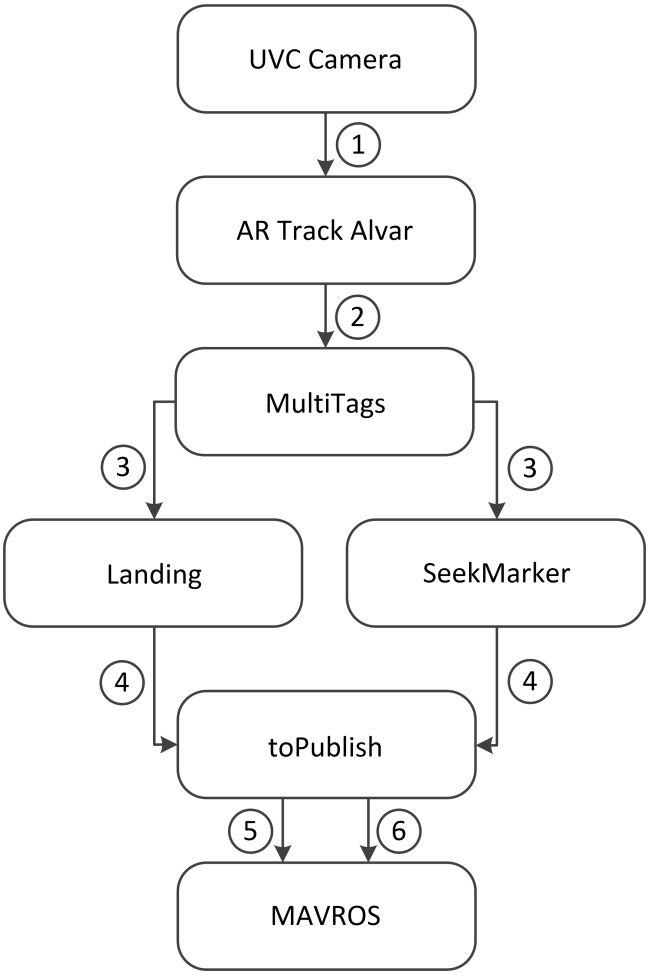
O filtro que se adequou ao problema de detecção do local de pouso é o filtro de medianas. Esse filtro é capaz de suavizar curvas retirando medidas impulsivas, ou *outliers*, através de uma janela de amostragem. Para o problema em questão foi adotada uma janela de 4 amostras, que resultou, experimentalmente, em uma resposta aplicável sem que houvesse atraso significativo que comprometesse o procedimento e aterrissagem.

O nó denominado *MultiTags*, abordado na Seção 3.5, é projetado para realizar as transformações e disponibilizar o ponto de pouso para o algoritmo de aterrissagem. Desta forma, o algoritmo principal não se ocupa com o grande volume de transformações necessárias.

3.5 ESTRUTURA DO *FIRMWARE* DE ATERRISSAGEM

A estrutura do *firmware*, sobre o computador companheiro, é ilustrada no processamento grafo da Figura 28. Nele são abordados os nós de processamento ROS, além dos tópicos de comunicação inerentes ao procedimento de aterrissagem. Conceitos intrínsecos ao ROS, e abordados na presente seção, podem ser verificados na Seção 2.1.

Figura 28 – Processamento grafo do *firmware*



Fonte: Elaborada pelo autor

Como representado na Figura 28, o primeiro nó em execução é o *UVC Camera* e o seu papel é semelhante a de um *driver*. Ele é responsável por capturar as imagens provindas por uma *webcam* e encapsulá-las em mensagens do tipo *sensor_msgs/CameraInfo* e *sensor_msgs/Image*. A primeira mensagem contém informações da imagem, como dimensionamento e parâmetros de calibração, enquanto a segunda transporta os *pixels* da imagem. Essas mensagens fluem pelos tópicos */camera_info* e */image_raw*, representados por 1 e torna as imagens acessíveis a aplicações estruturadas em ROS. Já o nó *Ar Track Alvar* é o encarregado de detectar os marcadores artificiais dispostos no ambiente. Como explicado na Seção 2.5, ele encaminha um vetor de mensagens do tipo *ar_track_alvar/AlvarMarker* que possui a disposição espacial dos

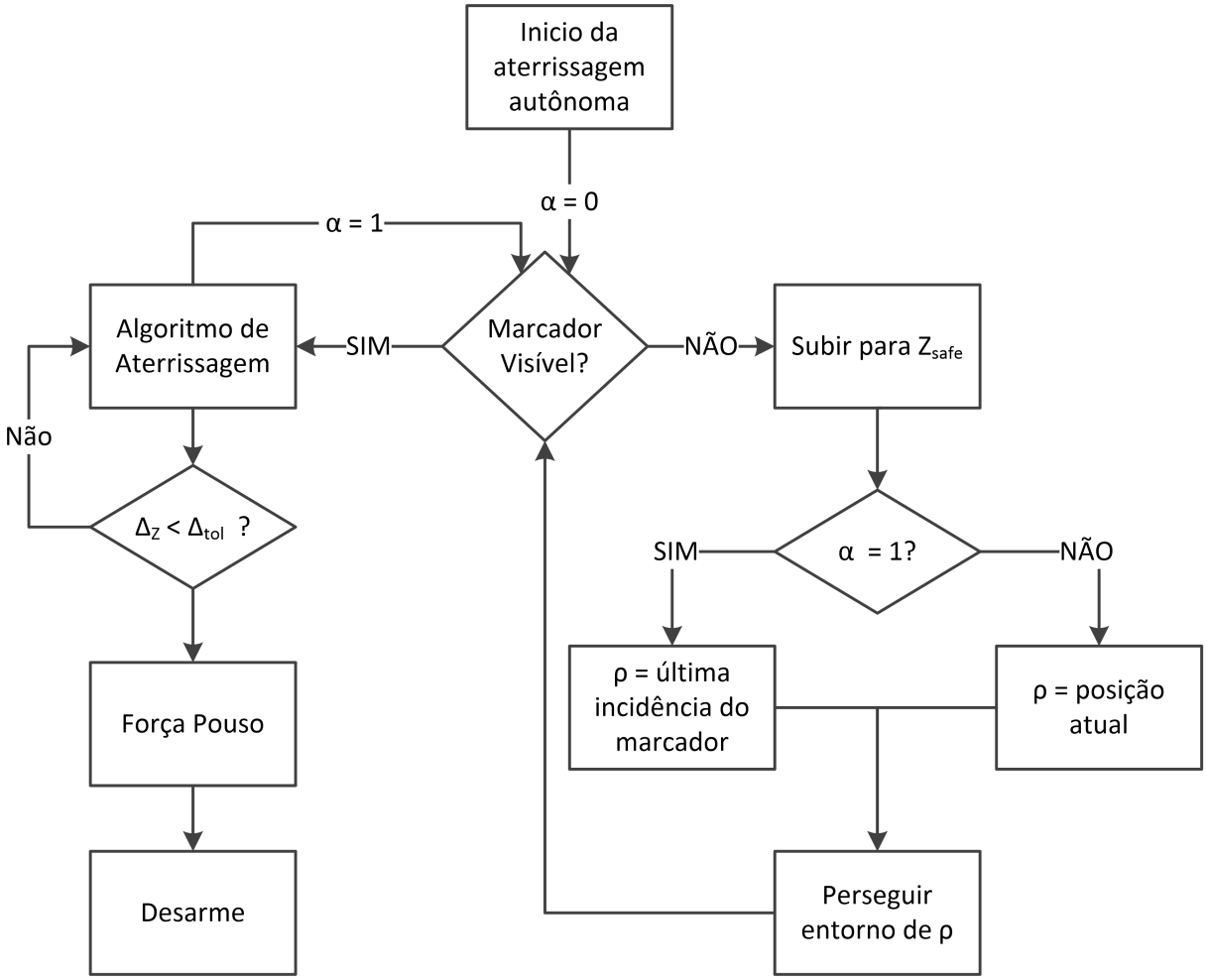
marcadores vistos, posição tridimensional e orientação em *quaternion*, e a identificação do IDs dos marcadores associados. O seu tópico é denominado */ar_pose_marker* indicado por 2.

O *multiTags* é responsável por examinar os vetores que indicam as posições dos marcadores detectados para o local de pouso. Através dos IDs detectados, esse algoritmo executa as transformações homogêneas relatadas nas Seções 3.3 e 3.4, ou seja, gera o ponto ${}^{\mathbf{V}}\mathbf{P}_{\mathbf{M}}$. Esse ponto é encapsulado na mensagem do tipo */pose*, com dados de posição e orientação em formato *quaternion*, e transmitido pelo tópico */vision/main_tag_pose*, representado por 3. A mensagem que flui por 3 também possui um cabeçalho com uma *flag* que indica se existe ou não um marcador visível. Essa *flag* torna-se importante pois é responsável por ativar o nós *Landing* ou *SeekMarker*. O *Landing* é o código responsável pela aterrissagem. Nele são codificados as lógicas *fuzzy* ou RNA, abordados nas Seções 3.6 e 3.7, respectivamente. São gerados por ele comandos de velocidade e encapsulados em mensagens do tipo */toOffboard* sobre o tópico */setpoint_offboard*, representado por 4. Já o nó *SeekMarker* realiza uma rota de procura por marcadores visíveis em situações de ausência do local de pouso. Ele executa atuação de deslocamentos incrementais através da mensagem de posicionamento */toOffboard*. O seu tópico de publicação é o */setpoint_offboard*, número 4. A mensagem */toOffboard* é criada para o presente sistema, ela encapsula cabeçalhos com *flags* e *timestamps*, além de dados de posicionamento e/ou velocidade. As *flags* do cabeçalho indicam qual tipo de dado deve-se considerar para os cálculos. Isso torna-se necessário, já que o nó *toPublish* identifica o tipo de mensagem e a reproduz corretamente para que o nó MAVROS a encapsule e a envie ao PX4. O *toPublish* envia mensagens de posicionamento através do tópico */mavros/setpoint_local/position*, ou de velocidade através do tópico */mavros/setpoint_position/cmd_vel*, ambos indicados por 5 e 6, respectivamente. O nó *toPublish* é um processo de segurança de operação e a sua principal função é estabelecer uma taxa de comunicação constante de 10Hz e, assim atender o requisito mínimo de segurança do modo de voo *OFFBOARD*, como descrito na Seção 2.3.4. O *toPublish* desacopla a exigência de requisito de taxa mínima que sobrecarregaria os nós *Landing* ou *SeekMarker*, permitindo que os últimos citados foquem apenas nas tomadas de decisões.

O fluxograma da Figura 29 elucida de maneira mais detalhada o procedimento e tomada de decisão realizado pelos algoritmos *Landing* e *SeekMarker*, que são executados paralelamente, para a execução da aterrissagem.

Assim que o procedimento de pouso é acionado, a variável de decisão α é configurada como zero e se mantém nesse valor até que o local de pouso esteja visível. No momento em que o local de pouso torna visível, α é comutada para 1. Se o marcador estiver visível, o algoritmo de aterrissagem atua sempre verificando duas situações: se o marcador de pouso continua à vista e se a altura, Δ_Z, da aeronave já se encontra dentro de uma

Figura 29 – Fluxograma para o procedimento de aterrissagem autônoma



Fonte: Elaborada pelo autor

região de tolerância, Δ_{tol}. A variável de tolerância de altura é importante, pois ela indica quando a missão está próxima de seu fim. Isso é necessário posto que em alturas muito baixas, ou seja, quando a câmera está muito próxima do marcador de pouso, a visão pode ser comprometida. Como a altura é pequena uma rotina de "força pouso" é executada, induzindo uma velocidade para baixo na aeronave para que ela encoste no local de pouso e, consequente, desarme os motores. Essa estratégia de "força pouso" também é verificada em (39, 43, 45).

Outra situação possível de ocorrer é quando nenhum local de pouso é detectado. Assim, o *SeekMarker* começa a atuar, subindo até uma altura de segurança, Z_{safe}. Essa altura permite abrir o campo de visão do algoritmo de detecção e expandir a área de movimentação da aeronave de forma segura. Atingido a altura Z_{safe}, a variável α é analisada. Caso α seja 1, o algoritmo executa uma rota quadrada espiral entorno da última incidência de marcador visto. Caso α seja 0, a rota quadrada espiral é executada na posição da aeronave. Essa rota de busca é interrompida assim que um marcador for

visto, reativando o algoritmo de aterrissagem *Landing*.

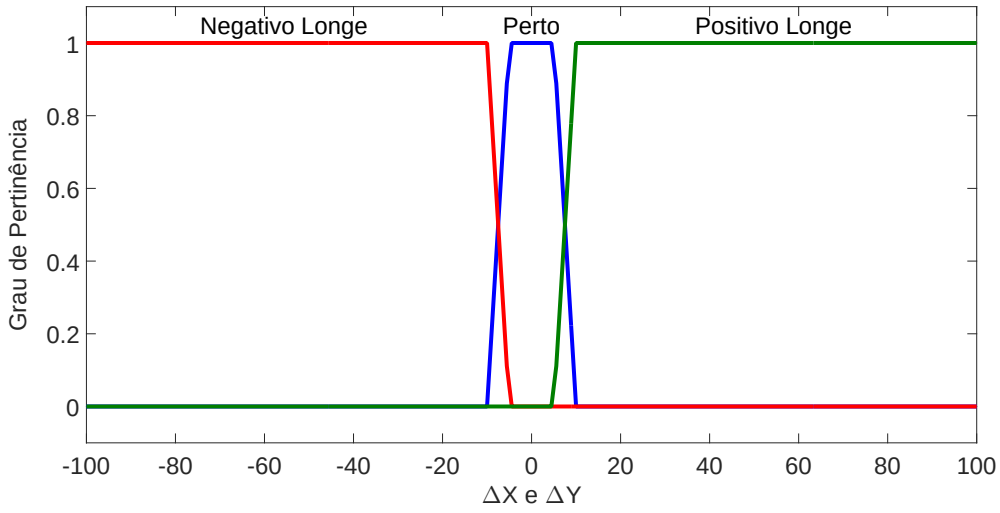
3.6 LÓGICA *FUZZY* PARA ATERRISSAGEM

Até o momento o algoritmo de aterrissagem foi abordado de forma genérica. O intuito era explicar o funcionamento do *firmware* como um todo e, assim, explanar os processos, os nós, as mensagens e as comunicações. Agora a presente seção, além da Seção 3.7, explorará o desenvolvimento do algoritmo. A priori torna-se necessário o projeto da lógica *fuzzy* para que, em seguida, seja utilizada no treinamento da RNA como entidade supervisionadora. Vale ressaltar que o projeto visa a redução da complexidade computacional e a lógica *fuzzy* deve também atender esse requisito. Logo, a modelagem do problema deve ser mais simplificada possível, sem que haja a perda de generalidade.

A modelagem é divida em duas etapas: compensação horizontal e vertical dos erros relativos à posição do local de pouso. Ou seja, inicialmente o VANT compensa os erros ΔX e ΔY provindos do algoritmo de visão computacional. Esses erros são eleitos como variáveis de entrada *fuzzy*, Figura 30. Logo, a aeronave executa um movimento de arraste bidimensional até se encontrar em uma região aceitável sobre o marcador de pouso, mantendo a sua altura de deslocamento. Quando essa região aceitável é atingida, o VANT executa a sua descida. Caso haja a perda dessa região durante o procedimento de descida, o movimento de arraste se repete mantendo a altura momentânea. Devido à testes prévios, verifica-se a não necessidade da modelagem do erro de altura, ou ΔZ, sobre o algoritmo *fuzzy*. O motivo dessa abordagem é justificada pela combinação das variáveis ΔX e ΔY que já inclui a informação do momento de descida. Logicamente, a variável ΔZ é monitorada por outro *looping* de processamento que informa quando o pouso será finalizado. Dessa maneira, a redução das variáveis no processamento *fuzzy* acarreta em um ganho do tempo de tomada de decisão do algoritmo. O algoritmo não precisa fuzzificar mais uma variável, estimar áreas de funções membros e não gera regras extras para o processo de inferência. Cada um dos erros ΔX, ΔY e ΔZ são extraídos do ponto ${}^{\mathbf{V}}\mathbf{P}_{\mathbf{M}}$.

As variáveis de entrada possuem três regiões ou grupos *fuzzy* trapezoidais: Negativo Longe (NL), Perto (P) e Positivo Longe (PL). Verifica-se também, a partir de testes anteriores, que esses grupos são suficientes para o problema, além de reduzir a complexidade computacional. A essa fato deve-se o ângulo de abertura da câmera utilizada. Esse ângulo gera uma projeção pouco efetiva para ativação de grupos mais distantes ou elaborados, como por exemplo grupos Médio Longe ou Muito Longe. A função membro Perto (P) indica que o VANT atingiu a região sobre o local de pouso. Como o *fuzzy* não é determinístico, essa região, que varia de -10% a 10%, é menos susceptível a erros inerentes ao procedimento, como leituras de sensores erradas ou oscilatórias. A tolerância também se dá pela sobreposição do grupo Perto com os outros grupos. A referência desses grupos é relativa ao *frame* da aeronave, indicados pela Figura 31. Vale lembrar que a abscissa da

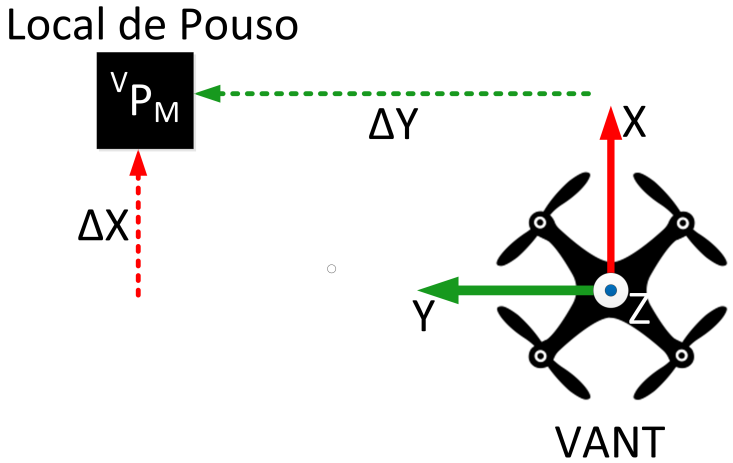
Figura 30 – Configuração das variáveis *fuzzy* de entrada: ΔX e ΔY



Fonte: Elaborada pelo autor

representação das variáveis *fuzzy* indica a porcentagem de -100% a 100% da imagem total vista, dado uma altura de atuação máxima. Essa altura é a maior distância que o local de pouso é visível pelo algoritmo de visão. Já a ordenada indica o grau de pertinência de 0 a 1.

Figura 31 – Sistema de coordenadas das variáveis *fuzzy*



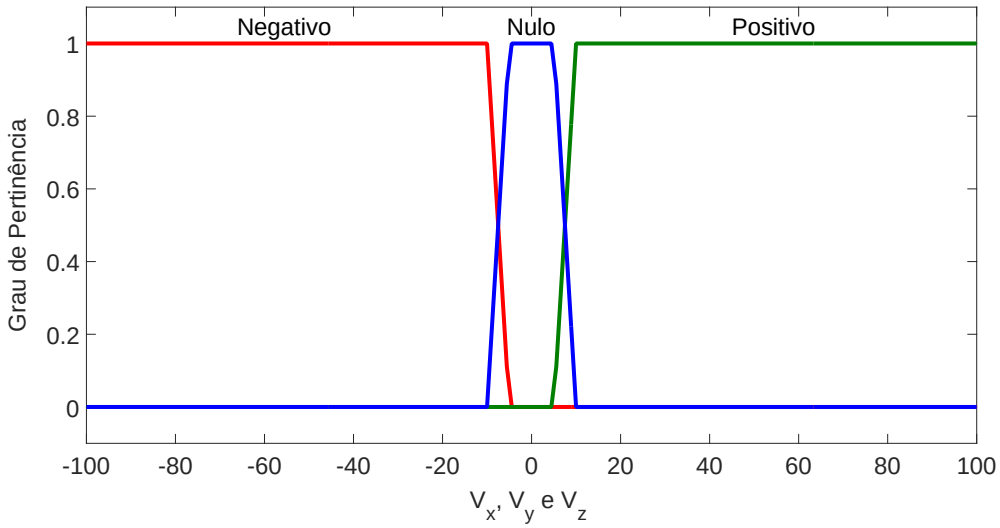
Fonte: Elaborada pelo autor

As atuações do *fuzzy* podem ser em velocidade ou em deslocamento incremental. Contudo, o *MAVROS* aplica suas atuações de deslocamento baseado no *frame* inercial. Dessa maneira, os deslocamentos seriam influenciados pelas medidas de IMU e GPS, que apresentam erros em torno de 1 a 2 metros quando preprocessados. Logo, a magnitude do erro de estimativa de posição do VANT pode influenciar em sua atuação de pouso.

Consequentemente, adota-se a atuação de velocidade linear pelo algoritmo. A influência das medidas do *frame* inercial sobre as atuações de velocidade são menos críticas, já que são enviados de forma indireta dados de acelerações ao motores. Mesmo que a velocidade indicada não seja atendida, o algoritmo de visão realimenta o sistema constantemente, reajustando possíveis erros de direcionamento.

As variáveis de saída do *fuzzy*, V_X, V_Y e V_Z, são indicadas na Figura 32. Elas possuem os mesmos conjuntos *fuzzy*: Negativo (N), Nulo (Z) e Positivo (P). Verifica-se a não necessidade de outros níveis de velocidade, como já dito, a abertura da câmera não proporciona a possibilidade de detecção extremamente longe, exigindo uma resposta mais rápida do algoritmo. O eixo das abscissas da representação das variáveis de saída *fuzzy* indicam porcentagem de -100% a 100% da velocidade máxima estabelecida pela aeronave. Já a ordenada indica o grau de pertinência de 0 a 1. Como constantemente a aeronave trabalha nas regiões de interseção dos grupos N-Z-P, as sobreposições estimam atuações de velocidade tridimensional mais sutis e suavizadas, ou seja, espera-se que o veículo não execute manobras bruscas em função do algoritmo *fuzzy*. A referência dos grupos de velocidade são relativa ao *frame* da aeronave, indicadas pela Figura 31.

Figura 32 – Configuração das variáveis *fuzzy* de saída: V_X, V_Y e V_Z



Fonte: Elaborada pelo autor

Evitando o acréscimo de processamento, nenhuma angulação relativa do VANT em relação ao local de pouso foi modelada e considerada. Essa estratégia, além de economizar processamento e tempo de reação do algoritmo, evita a inserção de erros inerentes a leituras errôneas de orientação da IMU da aeronave. Outro motivo é o fato do veículo ser onidirecional. Logo a aeronave pode atuar para qualquer direção que for necessária sem precisar se alinhar ao local de pouso.

As regras *fuzzy*, importantes para o processo de inferência do algoritmo, foram

projetadas visando atender o critério de simplicidade computacional. A redução das variáveis de entrada, excluindo o erro ΔZ, e o número de grupos *fuzzy* por variável permitem o quantidade reduzida de regras. Um total de 9 regras, Tabela 4. Essa abordagem é uma evolução do artigo precursor (52) a presente dissertação. Em (52) são verificadas um total de 73 regras, que exigiam computacionalmente e executavam o procedimento de maneira análoga ao apresentado neste trabalho.

Tabela 4 – Regras fuzzy para aterrissagem

	Variáveis de Entrada		**Variáveis de Saída**		
	ΔX	ΔY	V_X	V_Y	V_Z
Regras	Positivo Longe	Negativo Longe	Positivo	Negativo	Nulo
	Positivo Longe	Perto	Positivo	Nulo	Nulo
	Positivo Longe	Positivo Longe	Positivo	Positivo	Nulo
	Negativo Longe	Negativo Longe	Negativo	Negativo	Nulo
	Negativo Longe	Perto	Negativo	Nulo	Nulo
	Negativo Longe	Positivo Longe	Negativo	Positivo	Nulo
	Perto	Negativo Longe	Nulo	Negativo	Nulo
	Perto	Perto	Nulo	Nulo	Negativo
	Perto	Positivo Longe	Nulo	Positivo	Nulo

Fonte: Elaborada pelo autor

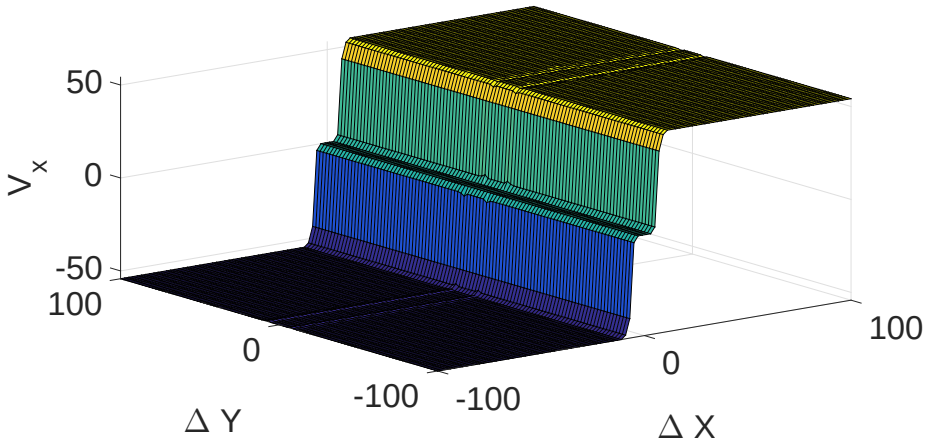
As regras são apresentadas na Tabela 4. De forma genérica, elas descrevem o movimento de compensação horizontal, ou arraste, da aeronave até ela se encontrar sobre o local de pouso e começar a sua descida. Caso a região sobre o marcador seja extrapolada, o movimento de arraste irá reatar mantendo a altura momentânea. Quando o pouso finalizar, o *looping* externo que monitora ΔZ finalizará a atuação do controle, dando início ao "força pouso" descrito na Seção 3.5.

Vale ressaltar que são utilizados os procedimentos de mínimo e máximo para os processos de implicação e agregação, respectivamente. A regras foram compostas apenas pela conjunção "E" através da operação de mínimo. O método de defuzzificação por centroide é o escolhido. As superfícies de resposta do processo de inferência estabelecido são apresentadas na Figura 33 para as variáveis V_X, V_Y e V_Z dado as entradas ΔX e ΔY.

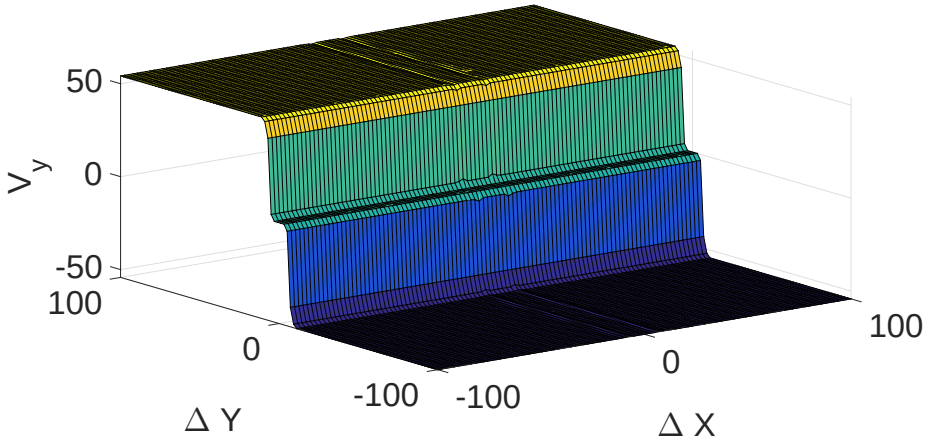
Para o projeto do sistema *fuzzy* proposto, foi utilizado a *toolbox* denominada *fuzzy* do software *Matlab*®. Essa *toolbox* permite facilidades gráficas de implementação, como relatado na Seção 2.6.6.

Uma vez concebido, o algoritmo precisa ser implementado para validação e execução. Os códigos que geram o nó ROS de execução são implementados utilizando a biblioteca *FuzzyLite* descrita na Seção 2.6.7 em linguagem *C++*. A lógica *fuzzy* é testada e validada no Capítulo 4.

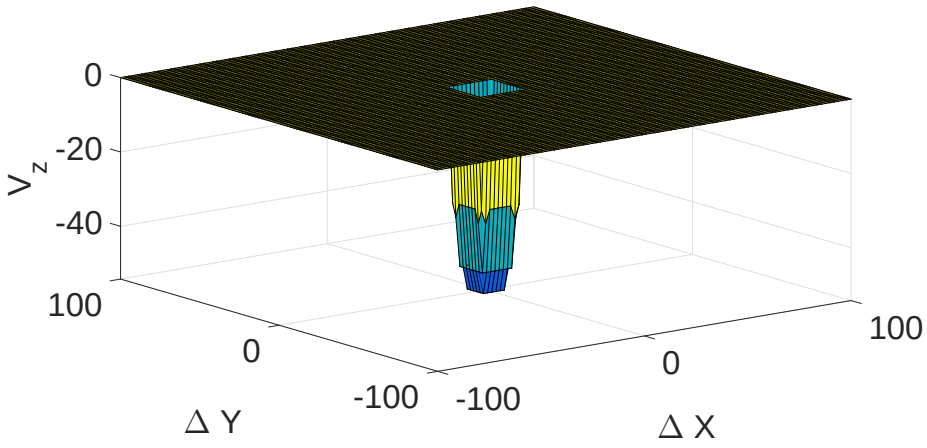
Figura 33 – Superfícies de decisão das variáveis de saída *fuzzy*



(a) Superfície de resposta V_x



(b) Superfície de resposta V_y



(c) Superfície de resposta V_z

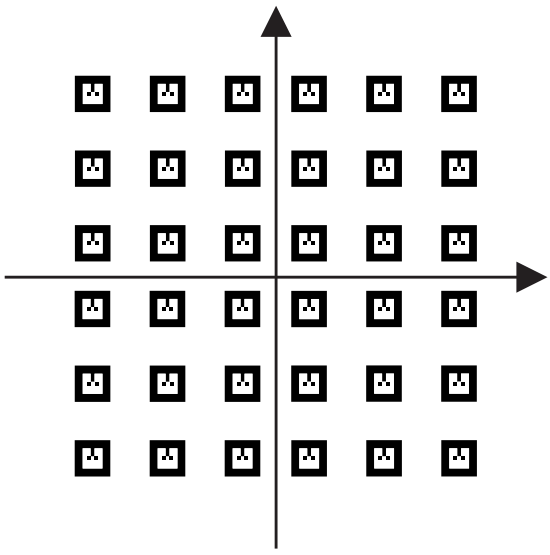
Fonte: Elaborada pelo autor

3.7 REDE NEURAL ARTIFICIAL PARA ATERRISSAGEM

Visto que o algoritmo de aterrissagem por lógica *fuzzy*, Seção 3.6, encontra-se realizado e funcional, visa-se a sua utilização como entidade supervisionadora no processo de treinamento da rede neural artificial proposta nesta dissertação. Como já dito anteriormente, a rede neural é capaz de reduzir a complexidade computacional, desde que devidamente projetada, pois não possui processos de fuzzificação, inferência, defuzzificação, estimativa de áreas e determinação de graus de pertinência que a lógica *fuzzy* Mamdani realiza para tomar suas decisões. Por outro lado, como entidade supervisionadora no processo de treinamento da RNA, o *fuzzy* é capaz de instruir e transmitir a inteligência inerente ao seu processamento à rede neural. Além disso, o processo de treinamento é facilitado, sem a necessidade de inserção de um especialista humano, que poderia incluir ações errôneas ou mal interpretadas na modelagem.

Para iniciar o procedimento de treinamento supervisionado da RNA, primeiramente é determinado o banco de dados de treinamento. O banco de dados de treinamento é constituído de possíveis posições do local de pouso sobre uma imagem adquirida pela *webcam*. Logo, cria-se e estima-se diferentes posições de marcadores como ilustrado na Figura 34. Vale ressaltar que cada dado do conjunto é constituído de posições bidimensionais X e Y, equivalentes aos erros ΔX e ΔY da lógica *fuzzy* (limitados de -100% a 100% da imagem total vista). Posteriormente, esse banco de dados é dividido em: 70%, 15% e 15% para conjuntos de treinamento, validação e teste, respectivamente.

Figura 34 – Exemplo de um conjunto de treinamento



Fonte: Elaborada pelo autor

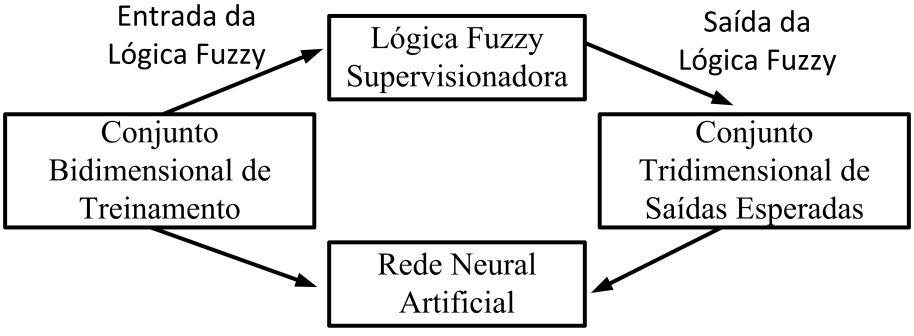
O processo de treinamento consiste da forma indicada na Figura 35. O conjunto de treinamento criado é apresentado para a rede neural e para a lógica *fuzzy* simultaneamente. Torna-se necessário um mapeamento dos valores desse conjunto, de -100% a 100% para -1 a 1, já que a RNA trabalha com essa excursão de sinal. As respostas geradas pela tomada de decisão da lógica *fuzzy* são inseridas como respostas desejadas, dado a entrada equivalente, à rede neural. Dado que a atuação do algoritmo *fuzzy* é baseado entre -100% a 100% da velocidade máxima estabelecida, é necessário também a normalização das saídas esperadas, pois a excursão do sinal de saída da RNA é de -1 a 1. Essa normalização, igual para os três eixos de atuação de velocidade, é realizada pela Equação 3.8:

$$v_{RNA} = \frac{-2 \cdot \left(V_{fuzzy}^{\max} - v_{fuzzy}\right) - \left(V_{fuzzy}^{\max} - V_{fuzzy}^{\min}\right)}{\left(V_{fuzzy}^{\max} - V_{fuzzy}^{\min}\right)} \tag{3.8}$$

em que v_{RNA} e v_{fuzzy} representam as saídas de uma amostra de treino qualquer para a RNA e a lógica *fuzzy*, e $V_{fuzzy}^{\max}$ e $V_{fuzzy}^{\min}$ são a máximas e mínimas velocidades determinadas pelo algoritmo *fuzzy* supervisionador.

Assim um processo de treinamento supervisionado por um especialista "artificial" é proposto. O método de treinamento utilizado é o Levenberg-Marquardt com $\mu = 0{,}001$. Esse método de otimização apresenta um melhor desempenho e maior velocidade de convergência nos ajustes de ganhos da RNA. Esse desempenho melhorado é a consequência da utilização de derivadas de segunda ordem do erro. Além disso, o algoritmo de Levenberg-Marquardt estima, de maneira simplificada, a matriz Hessiana inversa para casos em que métodos baseados em algoritmos de Newton não possuem convergência $(80, 81)$. A restrição do método é quanto ao tamanho da RNA que se deseja treinar, o que tornaria o processo custoso computacionalmente e impraticável. Contudo, para o problema de aterrissagem esse efeito não foi perceptível, resultando em melhores respostas e tempo de convergência.

Figura 35 – Processo de treinamento da RNA



Fonte: Elaborada pelo autor

Determinado o processo de treinamento, é necessário projetar a topologia da rede neural. Mais especificamente determinar o número de neurônios utilizados, o número de camadas ocultas e os tipos de funções de ativação. Essas escolhas devem atender o critério de simplicidade computacional sem perda de generalidade do problema. Com dados bidimensionais de treinamento, sem a inserção da variável ΔZ, a rede projetada possui dois neurônios em sua camada de entrada, reduzindo assim sua complexidade. Já a camada de saída irá de acordo com as atuações tridimensionais de velocidade, assim, são utilizados três neurônios de saída.

A camada oculta proporciona a criação do conjunto de hiperplanos separadores mais elaborados, permitindo que a rede neural artificial extraia características intrínsecas, complexas e não lineares de um determinado problema (78). Como visa-se absorver a tomada de ações não lineares do algoritmo *fuzzy*, verifica-se a necessidade de utilização de uma camada oculta na rede. Como explanado em (78), a inserção de uma única camada oculta na rede neural é capaz de resolver a grande maioria dos problemas existentes. Logo, adota-se apenas uma camada visando a redução da complexidade computacional. As funções de ativação para cada neurônio tornam-se importantes para a convergência do problema. Como a variável de entrada é limitada pela ângulo de abertura da câmera, ou campo de visão do algoritmo, e a variável de saída é limitada pela velocidade máxima da aeronave, a função de ativação deve saturar as respostas do neurônio. Desta forma usa-se a função de ativação tangente hiperbólica, vide Tabela 3. Essa função limita valores entre -1 e 1, representando a saturação das variáveis de saída e entrada. Adicionalmente, ela apresenta um comportamento suave que acarretará em uma atuação sutil da aeronave. A normalização dos dados de entrada durante o treinamento foi escolhida de maneira a ajustar os sinais de treinamento dentro da região de variação da função de ativação e, assim, valores não são constantemente saturados permitindo que o treinamento da rede seja mais eficiente.

A análise necessita determinar, ainda, quantos neurônios ocultos são necessários para a resolução do problema. Dessa forma são projetadas e testadas diferentes redes com números de neurônios na camada oculta. A execução desse procedimento é realizada no *software Matlab*®. Consequentemente, a média de erro no fim do processo de aprendizagem é verificado para quatro repetições. Mais especificamente, é avaliado a média do erro quadrático médio acumulado atingido pela rede proposta em comparação aos valores desejados, Equação 2.10. A Tabela 5 elucida o desempenho das diferentes redes testadas para 10^3 épocas. Verifica-se que o erro satura a partir de 10 neurônios ocultos, em torno de 0,005, e o aumento da quantidade de neurônios apenas resultaria no acréscimo da complexidade computacional. O número de neurônios é relacionado à complexidade da operação matricial da rede treinada. Dessa maneira, são escolhidos dez neurônios para a camada oculta da RNA proposta.

Tabela 5 – Desempenho da RNA dado diferentes números de neurônios na camada oculta

Neurônios	**Erro**
2	0,1190
3	0,0709
5	0,0353
10	0,0050
12	0,0053
15	0,0047

Fonte: Elaborada pelo autor

Com a rede neural artificial projetada e treinada, Figura 36, é possível verificar os ganhos sinápticos e de *bias* ajustados. Esses ganhos são apresentados pelas matrizes 3.11, 3.12, 3.13 e 3.14. Em posse desses valores, extraídos do software *Matlab*®, a etapa consequente é a implementação da rede neural em um nó de processamento ROS em *C++*. Todo ponto de aterrissagem ${}^{\mathbf{V}}\mathbf{P}_{\mathbf{M}}$ entregue pelo tópico */vision/main_tag_pose* é normalizado e aplicado sobre a rede neural. Em seguida, através de operação matricial dos ganhos sinápticos com os valores de entrada, a rede responderá através de valores tridimensionais de velocidade normalizados entre -1 a 1.

Como dito anteriormente, as saídas da RNA estão normalizadas entre -1 e 1 para as atuações de V_x, V_y e V_z. As velocidades nas coordenadas X e Y são desnormalizadas de acordo com a velocidade máxima positiva e negativa estabelecidas pelo algoritmo *fuzzy* supervisionador, Equação 3.9:

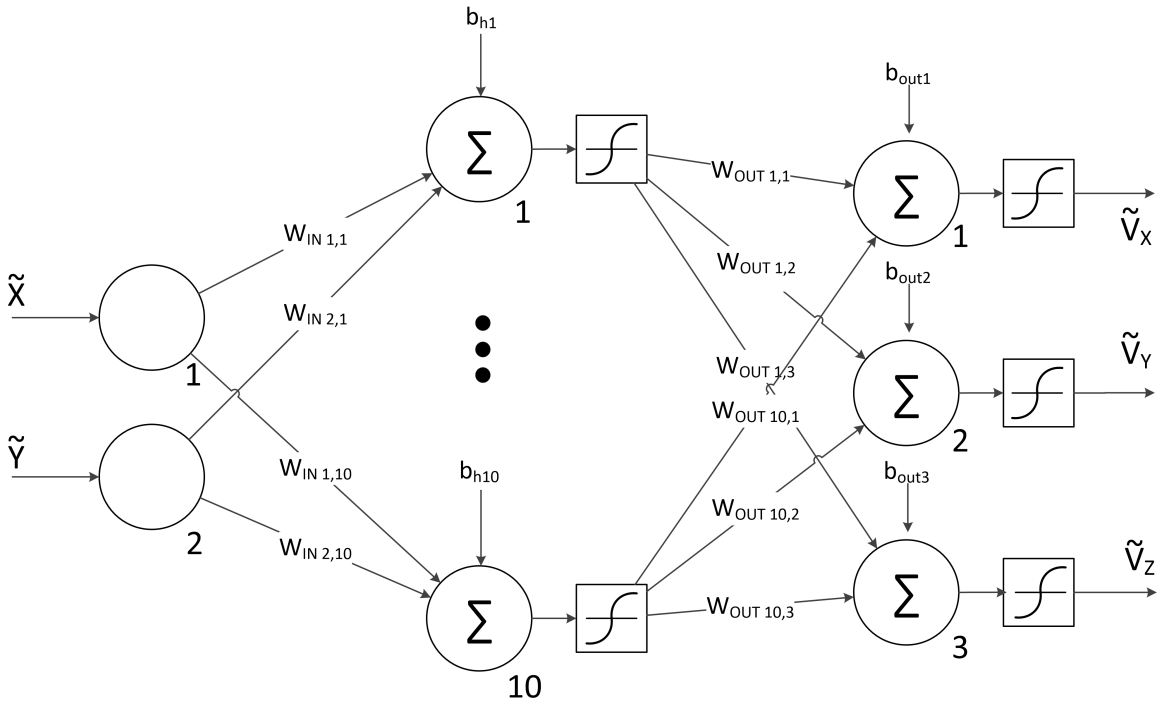
$$V_{x,y} = -\left(\frac{\left(1-\tilde{V}_{x,y}\right)\cdot\left(V_{x,y}^{\max}-V_{x,y}^{\min}\right)}{2} - V_{x,y}^{\max}\right) \tag{3.9}$$

Porém a velocidade no eixo Z é desnormalizada entre zero e a máxima velocidade de descida, ou velocidade mínima Equação 3.10:

$$V_z = -\left(\frac{\left(1-\tilde{V}_z\right)\cdot\left(-V_z^{\min}\right)}{2}\right) \tag{3.10}$$

em que $V^{\max}$ e $V^{\min}$ representam a máxima e mínima velocidades estabelecidas pelo algoritmo *fuzzy* supervisionador, e V e $\tilde{V}$ indicam as velocidades desnormalizada e normalizada estimada pela RNA, respectivamente.

Figura 36 – RNA projetada



Fonte: Elaborada pelo autor

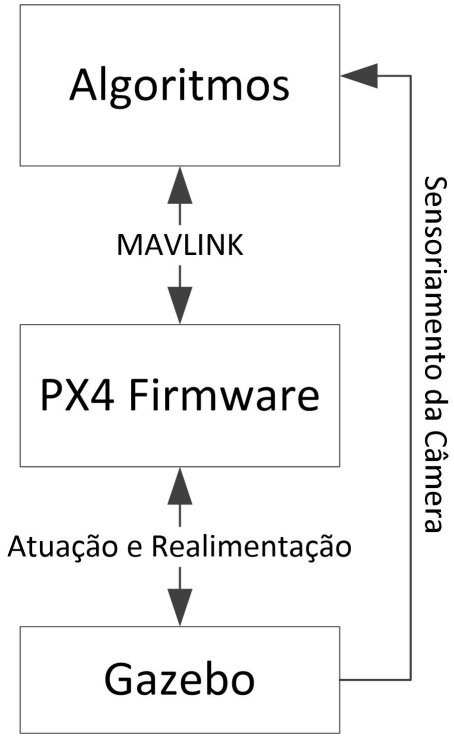
$$W_{IN} = \begin{bmatrix} 1,6630 & -0,9051 \\ 4,5297 & -1,6467 \\ 6,3364 & -2,4894 \\ 0,0549 & -45,9219 \\ 34,8281 & -0,1677 \\ -0,0581 & -43,5006 \\ -34,5214 & -0,1280 \\ 6,3091 & 2,4739 \\ 0,0133 & 0,8535 \\ -4,5106 & -1,6365 \end{bmatrix} \quad (3.11) \qquad b_h = \begin{bmatrix} -7,5854 \\ -0,3350 \\ -0,4300 \\ 6,4777 \\ -5,7899 \\ -6,2890 \\ -5,7581 \\ 0,4310 \\ 0,1768 \\ -0,3366 \end{bmatrix} \quad (3.12)$$

$$W_{OUT} = \begin{bmatrix} -8,3247 & -21,1811 & -198,7123 \\ -0,6537 & 0,1140 & -427,0764 \\ 0,1295 & -0,0677 & 203,5506 \\ 0,0017 & -592,3620 & -29,4103 \\ 662,7402 & -0,0145 & 176,5266 \\ -0,0053 & -551,1137 & 39,8660 \\ -647,4154 & -0,0012 & 177,2376 \\ 0,2068 & -0,0449 & -206,6239 \\ 0,1460 & -2,6300 & -501,6509 \\ 0,8484 & -0,0378 & -433,6014 \end{bmatrix} \quad (3.13) \qquad b_{out} = \begin{bmatrix} 7,0013 \\ 20,5298 \\ 197,2203 \end{bmatrix} \quad (3.14)$$

3.8 *SOFTWARE IN THE LOOP*

O *Software in the Loop* (SITL) é um modelo de simulação não convencional. Nesse procedimento a dinâmica do dispositivo a ser controlado é simulada, assim como sua interação com o ambiente, porém a planta e o *firmware* de controle reais são emulados em um computador. Dessa forma, o *firmware* embarcado proposto, Seção 3.5, desconhece a diferença entre um SITL e um teste real. Isto é, todos os protocolos e estruturas do *firmware* se mantêm iguais, sendo exatamente o mesmo utilizado em um sistema embarcado real. Para o caso de aterrissagem proposto nesta dissertação, o *firmware PX4* é emulado sobre um computador convencional com 4GB de memória RAM, processador Intel® I5 3,10GHz, placa de vídeo com memória dedicada de 1GB e sistema operacional Linux Ubuntu 14.04 de 64bits. O ambiente de simulação é o *software* Gazebo. O esquema da Figura 37 ilustra esse processo.

Figura 37 – *Software in the loop*



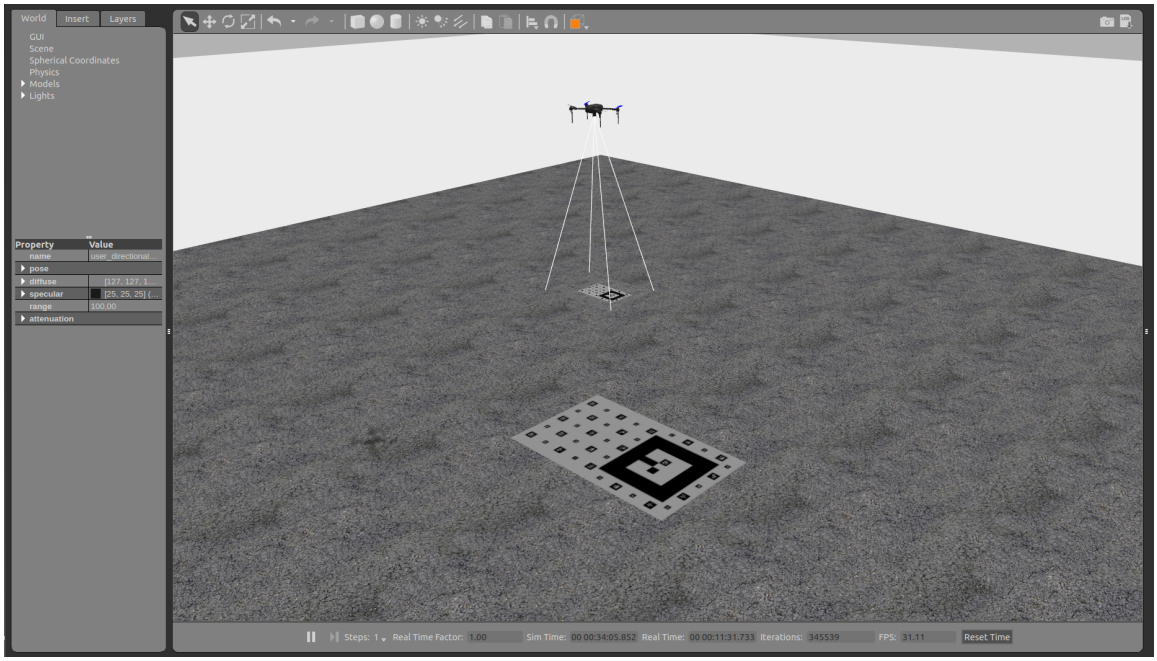
Fonte: Elaborada pelo autor

O ambiente de simulação adotado, Gazebo (Figura 38), simula a física e dinâmica de corpos de maneira próxima a realidade. Permite também que modificações sejam realizadas pelo usuário, tanto em nível de modelos quanto em nível de física e ambientações. Desenvolvido para Linux, o Gazebo está em constante desenvolvimento devido à comunidade, pois é um *software Open Source*. O uso do simulador é consolidado nas comunidades do ROS e da *PX4*. Por esses motivos, ele é o simulador adotado no SITL nesta dissertação.

Para que testes sejam realizados em SITL, torna-se necessário a modelagem gráfica de objetos para o ambiente de simulação, como é o caso da câmera e do marcador que representa o local de pouso. O modelo do VANT utilizado é disponibilizado pelo *firmware PX4*, referente ao modelo IRIS da empresa 3DR®. Esse modelo de aeronave interage através do protocolo MAVLINK não se diferindo na comunicação realizada pela aeronave real. O marcador de pouso foi confeccionado sobre o *software Corel Draw*®, com as medidas especificadas na Seção 3.4. Em seguida, por meio do *software* de modelagem *Blender*®, para Linux, a forma do local de pouso é modelada e a textura gerada pelo *Corel Draw*® é anexada ao material. Desta maneira, o modelo gráfico do marcador de pouso em formato de arquivo *collada* é exportado, Figura 39.

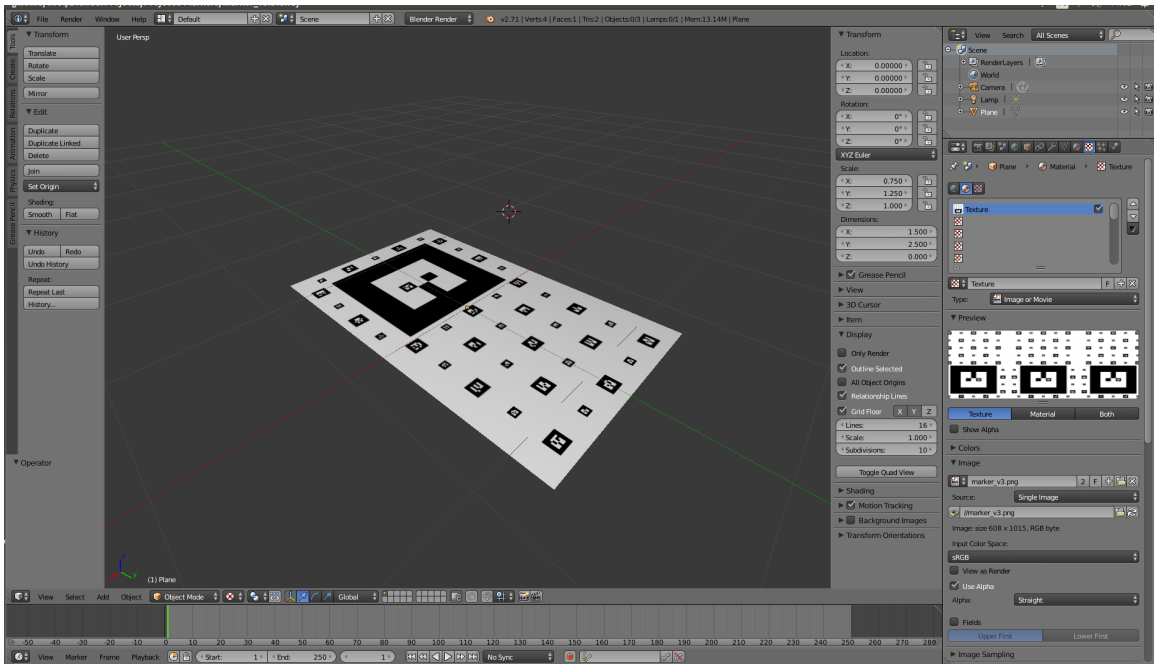
O Gazebo descreve seus objetos e ambientes, como robôs, terrenos, física e atores através de arquivos SDF. Esses arquivos são uma extensão do XML focados para a

Figura 38 – Ambiente de simulação Gazebo



Fonte: Elaborada pelo autor

Figura 39 – *Software* de modelagem Blender



Fonte: Elaborada pelo autor

simulação em Gazebo. Assim, de posse da forma e da textura do marcador, projeta-se um arquivo SDF para embarcá-lo na simulação. São inseridas características como massa, centro de massa, reflexibilidade e contatos de colisão. É projetado também um modelo de câmera baseado na *webcam Widecam* F100, da empresa *Genius*. Essa câmera possui

massa de aproximadamente $100g$ e ângulo efetivo de abertura de 90^o. Posteriormente, o modelo é anexado ao centro de massa da aeronave.

Para que as imagens sejam acessíveis pelo *Ar Track Alvar*, torna-se necessária a utilização de *plugins* ao SDF do modelo da câmera que executam o papel semelhante ao nó *UVC Camera* abordado na Seção 3.5. *Plugins* são bibliotecas dinâmicas programadas em *C++* que permitem que objetos simulados no Gazebo sejam comandados. Para interação com o ROS, uma API denominada Gazebo-ROS permite que dados sejam externados ao *framework*. Um *plugin* de movimentação bidimensional é inserido sobre o modelo do marcador para execução de pouso em objetivos móveis.

A aeronave simulada é controlada pelo *firmware PX4* emulado. Como abordado na Seção 3.5, o *MAVROS* permite que algoritmos atuem sobre a aeronave. Para o SITL, o *MAVROS* tem sua porta de comunicação direcionado pelo protocolo UDP (Seção 2.4):

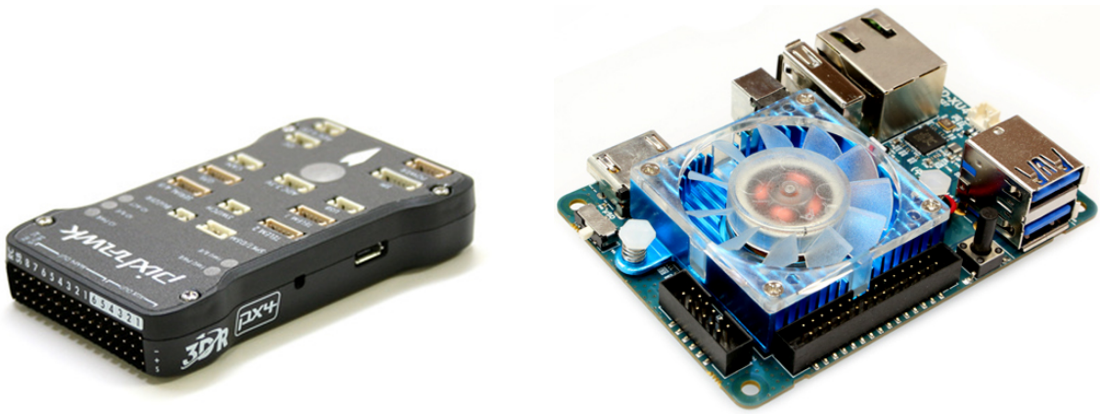
```
1    <arg name="fcu_url" default="udp://:14554@192.168.1.100:14555/>
```

3.9 *PROTÓTIPO*

3.9.1 *Hardware*

Para a execução dos resultados práticos, determina-se a *Flight Control Unit* (FCU) e o computador companheiro para o experimento de aterrissagem. Vale ressaltar que a utilização de um computador companheiro embarcado é desejada, fazendo com que não exista a necessidade de utilização de estações base de processamento, como proposto na Seção 1.2. Para a FCU adota-se a placa *Pixhawk* (Figura 40a (83)), enquanto para o computador companheiro a *Odroid XU4* (Figura 40b (84)) é a utilizada.

Figura 40 – FCU e computador companheiro utilizados



(a) *Pixhawk* (b) *Odroid XU4*

Fonte: Pixhawk.org e Odroid.com

A Pixhawk é uma FCU dentre os modelos comerciais existentes: *Ardupilot Mega*, *UAV Dev Board*, *FlexiPilot*, *SLUGS Autopilot* ou *A2* da empresa *DJI*. Essa placa foi iniciada por Lorenz Meier em seu título de *Excellence Scholarship* do Instituto Federal de Tecnologia de Zurique em 2009 (85). Continuada por grupo de estudantes desta instituição, a *Pixhawk* é um *hardware* de desenvolvimento aberto e comercializada pela *3DR*®. O processador *ARM* presente na *Pixhawk*, em conjunto com sua arquitetura, são as vantagens principais desta placa que possui como foco a utilização do *firmware PX4*.

Com o *PX4* embarcado, a *Pixhawk* é compatível com sistemas multinodais do ROS e com sistemas Linux em computadores companheiros. A placa também apresenta desempenho elevado em associação a RTOS. A variedade de periféricos compatíveis, que variam desde sensores, interfaces e módulos de comunicação, além do fato da *Pixhawk* ser um *hardware* livre, a torna uma boa opção perante as placas de controle aéreo concorrentes.

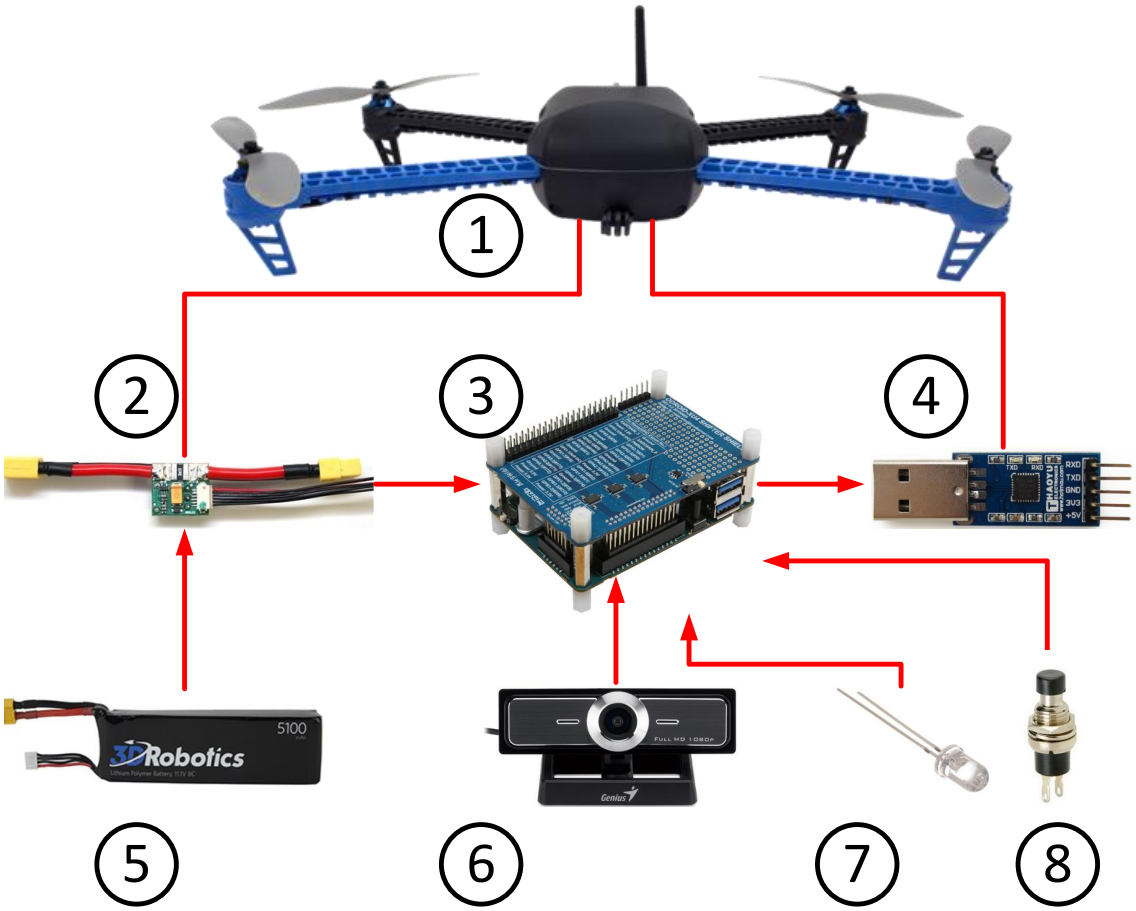
A *Odroid (Open Android)* é uma família de computadores embarcados inicialmente planejada como plataforma de desenvolvimento de *hardware* e *software* de entretenimento para sistemas *Android* (84). Contudo, devido ao seu desempenho, é muito utilizada com sistemas Linux para processamentos embarcados mais eficazes, incluindo até processamentos de imagem. Outros pontos importantes são: baixo consumo, $5V$ e $2A$, peso e tamanho reduzidos, em torno de $60g$ com dimensões de 83 x 58 x $22mm$. A variedade de portas e periféricos também torna a *Odroid XU4* uma opção viável para aplicação de aterrissagem embarcada sem o uso de estações base de processamento, ou *Ground Control Station* (GCS).

Determinadas as entidade processadoras, a aeronave *IRIS*, da empresa *3DR*® é a utilizada. Essa aeronave é um quadrotor, ou quadricóptero, com seus 4 motores predispostos em formato de "V". Vale ressaltar que ela possui a FCU *Pixhawk*. Assim, modificações são realizadas sobre sua estrutura para anexar o computador companheiro e a câmera. Informações complementares da estrutura do protótipo são apresentadas na Figura 41 e na Tabela 6.

A *Odroid* é inserida dentro de uma capa de proteção de confecção própria e anexada na parte inferior da aeronave, juntamente à câmera (Figura 42). A câmera utilizada é a *webcam Widecam* F100, da empresa *Genius*. Essa câmera possui o ângulo de abertura de 90^o, resolução de 1280 x 720, 30 *frames* por segundo e comunicação USB. O peso e dimensões desse dispositivo é de 82g e 48 x 150 x 49 mm, respectivamente.

Quanto a alimentação das entidades processadoras é utilizado um módulo conversor de tensão, que reduz a tensão da bateria do veículo, variante entre 12,4V a 10,4V, para 5,0V. Uma chave geral é inserida em série a alimentação. Um LED e um botão são soldados ao *shift* da *Odroid* para a interface humano-máquina. O botão inicializa a *Odroid* e o *firmware* de aterrissagem e o LED é um indicador de operação.

Figura 41 – Esquemático de montagem



Fonte: Elaborada pelo autor

Tabela 6 – Equipamentos do protótipo utilizado

Equipamento	Nome	Função
1	Quadrotor IRIS	Aeronave
2	Conversor de Tensão	Distribuir a tensão entre o quadrotor (10,4V∼12,4V) e o computador companheiro (5,0V)
3	*Odroid XU4* com *Shift*	Computador Companheiro
4	Conversor FTDI	Converter dados de USB em Serial
5	Bateria	Alimentação (10,4V∼12,4V)
6	*Webcam Genius WideCam*	Sensoriamento por visão
7	LED	Interface Humano-Máquina (Indicador de inicialização)
8	Botão	Interface Humano-Máquina (Botão para iniciar o *firmware*)

Fonte: Elaborada pelo autor

Figura 42 – Aeronave de teste

Fonte: Elaborada pelo autor

3.9.2 *Firmware*

O *firmware PX4* é inserido na *Pixhawk*, enquanto o sistema operacional Linux LUbuntu é o adotado na *Odroid XU4*. O ROS utilizado é o ROS *Indigo Bare Bones*, uma versão mais simplificada do *framework*, ideal para sistemas embarcados.

Para que ambas unidades processadores se comuniquem por intermédio do protocolo MAVLINK é utilizada uma conexão serial. Essa conexão é adaptada através do barramento *UART* da *Odroid* até o barramento serial 4 da *Pixhawk*. Utiliza-se um conversor FTDI para a conversão, vide Figura 41. A escolha de uma conexão cabeada permite uma comunicação mais eficiente e segura para a aplicação aérea. Vale ressaltar que o *baud rate* utilizado nessa comunicação é de 115200. Modificações nos sistemas do *PX4* e LUbuntu foram realizadas para essa comunicação. No *PX4*, no diretório "/etc" é instanciada a comunicação *Mavlink* através do comando:

mavlink start -d /dev/ttyS6 -b 115200

Na Odroid a instância da comunicação é realizada no *Launch* de execução do MAVROS:

```
1    <arg name="fcu_url" default="/dev/ttyACM0:115200" />
```

São desenvolvidos também *scripts* de inicialização do sistema Linux. Esses *scripts* executam e carregam todo o *firmware* de aterrissagem proposto assim que a aeronave e

a *Odroid* são ligadas pelo botão inserido. Mais especificamente, o serviço, ou *daemon*, *Upstart* do *kernel* Linux é executado assim que o sistema é inicializado (86). Em seguida, este *daemon* emite diversos eventos de sistema, inicializando funções básicas do sistema operacional. Um desses eventos, o *startup*, é o escolhido para inicialização do *firmware* de aterrissagem. Um arquivo de configuração (*.conf*) é inserido no diretório */etc/init* apontando o *script* de inicialização criado para o *Upstart*.

4 RESULTADOS E DISCUSSÕES

Neste capítulo são abordados e discutidos os testes realizados para verificação e validação da técnica de pouso da presente dissertação. São analisadas as técnicas de pouso utilizando a lógica *fuzzy* e a Rede Neural Artificial (RNA), discutindo comportamento e desempenho computacional.

Os testes preliminares e de desempenho são verificados em códigos de *Matlab*® e *C++*. Os procedimentos de aterrissagem são analisados em *Software in the Loop* (SITL) e em experimentos reais para locais de pouso estáticos e dinâmicos.

4.1 ANÁLISE PRELIMINAR

Como descrito na Seção 1.2, a RNA implementada para a aterrissagem autônoma utilizando visão computacional deve absorver as características e inteligência intrínsecas do *fuzzy*, pois esse algoritmo é utilizado como entidade supervisionadora do seu processo de treinamento. De acordo com a Seção 3.7, o erro quadrático médio atingido pelo processo de treinamento foi de aproximadamente 0,005.

Uma observação preliminar aos testes de voo é a análise de atuação da RNA em comparação ao algoritmo *fuzzy*, dado entradas equivalentes. Essa análise é realizada sobre o software *Matlab*® após o projeto da lógica *fuzzy* e da RNA. A Tabela 7 apresenta alguns resultados de atuação, dado o posicionamento (X e Y) do local de pouso em valores relativos que variam de -1 a 1. Esses valores são normalizados em relação a imagem adquirida pela câmera conforme abordado no Capítulo 3. Por meio dessa tabela é possível inferir que a RNA atinge, ou se aproxima com pequenos erros, dos valores esperados do *fuzzy*. As diferenças são mais evidentes nas atuações de descida, ativadas pelas regras referentes ao conjunto "Perto" (de -0,1 a 0,1) das variáveis *fuzzy* ΔX e ΔY, Seção 3.6. Exemplo é a entrada $\{X : 0,06\ Y : -0,01\}$ que resultou nas saídas $\{V_x : 0,6\ V_y : 0\ V_z : -0,96\}$, para *fuzzy*, e $\{V_x : 0,51\ V_y : -0,01\ V_z : -0,95\}$, para RNA. Porém as diferenças de atuações dos dois algoritmos são pequenas, mostrando que a RNA foi capaz de absorver o procedimento de controle da entidade supervisionadora *fuzzy*. Não se verifica também valores discrepantes de atuação, que poderiam acarretar problemas no pouso.

Vale ressaltar que para o procedimento aterrissagem as velocidades adotadas são reduzidas e, assim, não são evidenciadas as pequenas diferenças de atuações entra a RNA e a lógica *fuzzy*. Outro ponto importante é que devido a dinâmica da aeronave e a erros de sensoriamento, diferenças sutis de velocidades acabam por ser imperceptíveis no momento da aterrissagem.

Tabela 7 – Saídas do *fuzzy* e da RNA dado entradas equivalentes

Entrada		**Saída do *Fuzzy***			**Saída da RNA**		
X	**Y**	**Vx**	**Vy**	**Vz**	**Vx**	**Vy**	**Vz**
1.00	1.00	1.00	1.00	0.00	1.00	1.00	-0.02
0.50	1.00	1.00	1.00	0.00	1.00	1.00	0
0.01	1.00	0.00	1.00	0.00	0.02	1.00	0
-0.06	1.00	-0.60	1.00	0.00	-0.53	1.00	0
-0.08	1.00	-0.90	0.99	0.00	-0.99	1.00	0
0.04	0.50	0.00	1.00	0.00	0.09	1.00	0
0.01	0.50	0.00	1.00	0.00	-0.01	1.00	0
0	0.50	0.00	1.00	0.00	-0.00	1.00	0
-0.08	0.50	-0.90	0.99	0.00	-0.99	1.00	0
-0.10	0.50	-1.00	1.00	0.00	-1.00	1.00	0
-0.50	0.50	-1.00	1.00	0.00	-1.00	1.00	0
1.00	0.10	1.00	1.00	0.00	1.00	1.00	0
0.50	0.10	1.00	1.00	0.00	1.00	1.00	0
0.10	0.10	1.00	1.00	0.00	1.00	1.00	-0.04
0.10	0.10	1.00	1.00	0.00	1.00	1.00	-0.04
0.08	0.10	0.90	0.99	0.00	1.00	1.00	-0.26
-0.06	0.10	-0.60	1.00	0.00	-0.51	1.00	-0.19
-0.08	0.10	-0.90	0.99	0.00	-1.00	1.00	-0.25
-0.10	0.10	-1.00	1.00	0.00	-1.00	1.00	-0.04
-0.50	0.10	-1.00	1.00	0.00	-1.00	1.00	0
0.04	0.08	0.00	0.90	-0.80	0.04	1.00	-0.83
0.01	0.08	0.00	0.90	-0.80	-0.02	1.00	-0.86
0	0.08	0.00	0.90	-0.80	0.00	1.00	-0.86
1.00	0.06	1.00	0.60	0.00	1.00	0.45	0
0.08	0.06	0.90	0.66	-0.80	1.00	0.51	-0.83
-0.04	0.06	0.00	0.60	-0.96	-0.04	0.51	-0.95
-1.00	0.06	-1.00	0.60	0.00	-1.00	0.57	0
0.50	0.04	1.00	0.00	0.00	1.00	0.02	0
0.10	0.04	1.00	0.00	0.00	1.00	0.03	-0.14
0.06	0.01	0.60	0.00	-0.96	0.51	-0.01	-0.95
0.04	0.01	0.00	0.00	-1.00	0.03	-0.01	-0.98
1.00	0	1.00	0.00	0.00	1.00	-0.01	0
0.08	-0.01	0.90	0.00	-0.80	1.00	0.02	-0.82
0.06	-0.01	0.60	0.00	-0.96	0.51	0.02	-0.95
0.04	-0.01	0.00	0.00	-1.00	0.03	0.02	-0.98
-0.08	-0.04	-0.90	0.00	-0.80	-1.00	-0.03	-0.82
-0.10	-0.04	-1.00	0.00	0.00	-1.00	-0.03	-0.13
-0.50	-0.04	-1.00	0.00	0.00	-1.00	-0.05	0
0.10	-1.00	1.00	-1.00	0.00	1.00	-1.00	0
0.08	-1.00	0.90	-0.99	0.00	1.00	-1.00	0
0	-1.00	0.00	-1.00	0.00	0.00	-1.00	-0.01
-0.50	-1.00	-1.00	-1.00	0.00	-1.00	-1.00	0
-1.00	-1.00	-1.00	-1.00	0.00	-1.00	-1.00	0

Fonte: Elaborada pelo autor

4.2 DESEMPENHO COMPUTACIONAL

Um ponto importante, discutido na Seção 1.2, é o desempenho computacional de ambos algoritmos: *fuzzy* e RNA. Para verificação dessa característica, cada algoritmo é implementado em *C++*, através de funções específicas. Logo, deseja-se estabelecer o desempenho do bloco de tomada de decisão de cada estratégia proposta. Para execução do teste de performance, a biblioteca SkyPat (87) para Linux é adotada. Essa biblioteca permite, através da eventos denominados *perf_events* do *kernel* Linux, estimar a performance de blocos de códigos em *C/C++* em tempo de execução.

Utilizando-se de um computador convencional, com 4GB de memória RAM, processador Intel® I5 3,10GHz e sistema operacional Linux Ubuntu 14.04 de 64bits, primeiro estima-se o número de instruções, em nível de *hardware*, necessárias para cada bloco de tomada de decisão. A Tabela 8 apresenta os valores médios de instruções dado algumas entradas equivalentes para o *fuzzy* e a RNA.

Tabela 8 – Instruções em nível de *hardware* realizadas por cada algoritmo proposto

Entrada X, Y	**Fuzzy**	**RNA**
	Instruções	**Instruções**
1,00 , 1,00	259734	39321
0,50 , 1,00	259886	39347
0,10 , 0,10	259886	39347
0,02 , 0,08	259886	39347
0,00 , 0,00	259886	39347
-0,04 , -0,04	258784	39383
-0,10 , -0,08	425448	39357
-0,10 , -0,10	271868	39357
-0,50 , -0,50	271286	39331
0,50 , -0,10	265940	39365
0,10 , -0,08	265940	39365
-0,08 , -0,04	405814	39383
0,04 , 0,00	258784	39383
-0,04 , 0,00	258784	39383
-1,00 , 1,00	265403	39348

Fonte: Elaborada pelo autor

Outro parâmetro de desempenho estimado é o de tempo de execução. Esse tempo é baseado pelo número de ciclos de CPU medidos durante cada tomada de decisão. Vale lembrar que o processador possui um temporizador de aproximadamente $3,10$GHz. Os valores médios do tempo de tomada de decisão de cada algoritmo e o número de ciclos da CPU são apresentadas na Tabela 9, dado entrada equivalentes.

Tabela 9 – Tempo de tomada de decisão e ciclos de CPU dos algoritmos propostos

Entrada	**Fuzzy**		**RNA**	
X, Y	**Tempo (ns)**	**Ciclos da CPU**	**Tempo (ns)**	**Ciclos da CPU**
1,00 , 1,00	45635	151876	10630	38822
0,50 , 1,00	44347	151207	10725	38764
0,10 , 0,10	44347	151207	10725	38764
0,02 , 0,08	44347	151207	10725	38764
0,00 , 0,00	44347	151207	10725	38764
-0,04 , -0,04	44301	151044	10769	39261
-0,10 , -0,08	75569	257364	11069	39099
-0,10 , -0,10	46964	160209	10992	39213
-0,50 , -0,50	46695	160253	11030	38574
0,50 , -0,10	46948	156004	11171	38871
0,10 , -0,08	73014	248721	10608	39357
-0,08 , -0,04	72920	247727	11039	38915
0,04 , 0,00	45708	151591	11190	38862
-0,04 , 0,00	44427	152376	11077	38818
-1,00 , 1,00	46660	159455	11049	39058

Fonte: Elaborada pelo autor

4.2.1 Discussões

Verifica-se a partir das Tabelas 8 e 9 que o algoritmo *fuzzy* consome mais instruções de máquina e possui um tempo maior de tomada de decisão em comparação a RNA. Outro fato é que o *fuzzy* varia com maior intensidade esses parâmetros de desempenho, ao contrário da RNA. Isso valida a premissa inicial relativa à redução da complexidade computacional proposta nesta dissertação, uma vez que as operações do *fuzzy* Mamdani variam de acordo com as regras ativadas exigindo das fases de implicação, agregação e defuzificação abordadas no Capítulo 2.6. Já a RNA é uma operação matricial fixa, que possui maior constância e exige menos computacionalmente.

Mais especificamente a RNA proporciona, em média, uma redução aproximada de: 6,57 a 10,81 vezes o números de instruções e de 3,87 a 6,65 vezes o tempo de tomada de decisão de acordo com os valores de entrada para o problema proposto, Tabela 10. Ressalta-se que o tempo de tomada de decisão, que demonstra a redução da complexidade computacional, não implica em rapidez de atuação, pois a frequência de atualização do sensoriamento proposto, em torno de 4Hz, é mais lento impedindo execuções em tempo reduzido.

Tabela 10 – Ganho em desempenho da RNA em relação ao *fuzzy*

Parâmetro de Performance	**Ganho**
Instruções	de 6,57 a 10,81 vezes
Tempo de Decisão	de 3,87 a 6,65 vezes

Fonte: Elaborada pelo autor

4.3 ESTUDO DE CASO: SITL

Para a verificação do comportamento da aterrissagem utilizando o *firmware* proposto, Seção 3.2, são estabelecidos testes em *Software in the Loop*. As configurações e especificações adotadas para esse procedimento são descritas na Seção 3.8.

São propostos os seguintes estudos de casos: alvos estático e dinâmicos. Nos alvos dinâmicos são testados os pousos com alvo em movimento retilíneo e circular. São apresentados o comportamento da estratégia de controle *fuzzy* e RNA para cada caso. O computador utilizado é o mesmo apresentado para os testes de desempenho computacional (Seção 4.2): 4 GB de memória RAM, processador Intel® I5 3,10 GHz, sistema operacional Linux Ubuntu 14.04 de 64bits e placa de vídeo com memória dedicada de 1 GB.

4.3.1 Caso 1: Alvo Estático

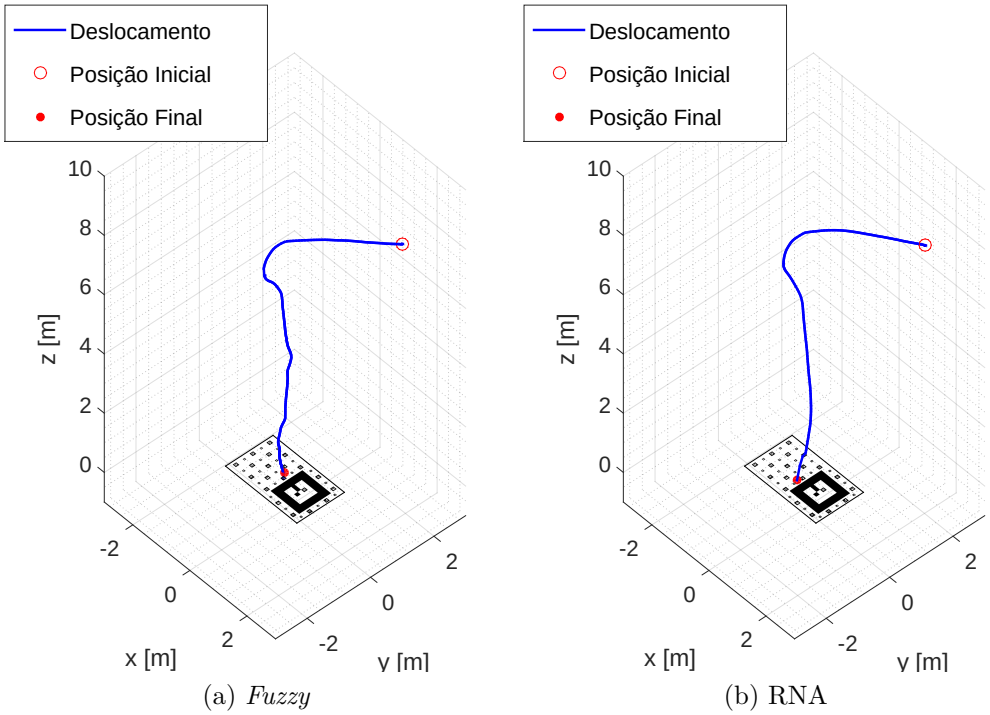
O primeiro teste consiste em dispor o marcador de pouso sobre uma posição fixa. A aeronave é colocada em uma localização relativa ao local de pouso de 8 m de altura e de 2 m de distância longitudinal e latitudinal. A priori, o marcador é visível dado o posicionamento adotado. A velocidade máxima estabelecida é de 20 cm/s para cada eixo de atuação. O "força pouso" é ativado a uma distância de 50 cm.

Durante a execução da aterrissagem, são armazenados dados de posicionamento baseado nas medidas sensoriais da aeronave em SITL. Além disso, as medidas fornecidas pelo Gazebo são armazenadas como *Ground Truth*. Os resultados para a aterrissagem utilizando a lógica *fuzzy* e RNA são apresentados na Figura 43. As reduções dos erros tridimensionais de distância relativa ao local de pouso (ΔX, ΔY e ΔZ) são expostas, individualmente, na Figura 44.

Um total de 30 repetições são realizadas para cada algoritmo. Assim, é estimada a distância radial média em relação ao ID 34, indicador central do local de pouso. A Tabela 11 apresenta a média e o desvio padrão dessas medidas. Todas a aterrissagens, de ambos os algoritmos, atingiram a região interna do marcador de pouso para as velocidades máximas adotadas.

O posicionamento do local de pouso, determinado pela estratégia adotada da Seção 3.4, é ilustrado pela Figura 45. Nessa figura são apresentados o posicionamento do

Figura 43 – Pouso em alvo estático em SITL



(a) *Fuzzy* (b) RNA

Fonte: Elaborada pelo autor

Tabela 11 – Erro radial em relação ao centro do marcador estático

Algoritmo	Erro Radial Médio [m]	Desvio Padrão [m]
Fuzzy	0,232	0,087
RNA	0,256	0,084

Fonte: Elaborada pelo autor

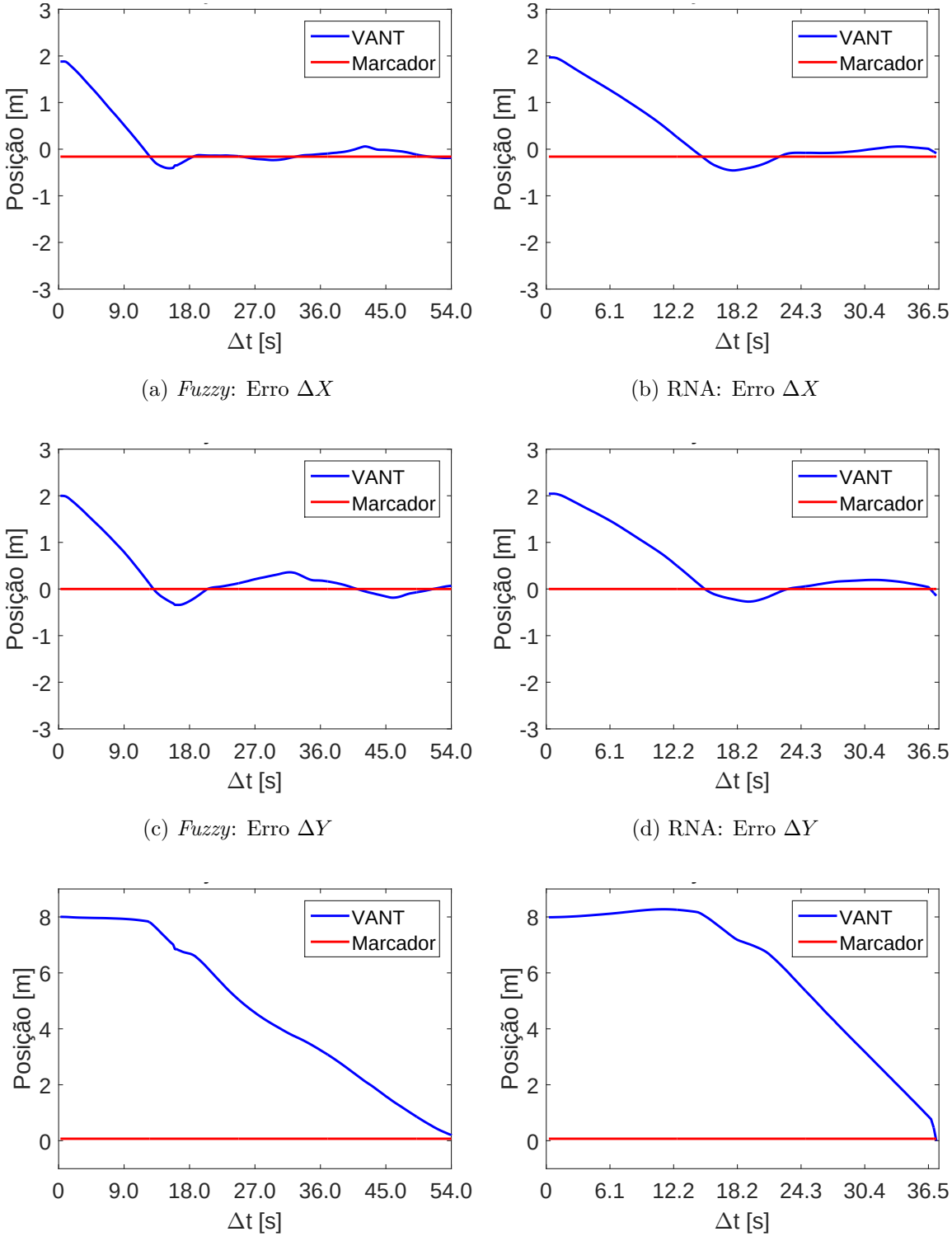
marcador em relação a aeronave, filtrado e não filtrado, além do *Ground Truth* informado pelo Gazebo.

4.3.2 Caso 2: Alvo Dinâmico com Movimento Retilíneo

O segundo teste consiste em dispor a aeronave em uma localização relativa ao local de pouso de 8 m de altura. Assim que o algoritmo de aterrissagem é ativado, o marcador desloca com uma velocidade retilínea uniforme de 20 cm/s. A priori, o marcador é visível dado o posicionamento adotado. A velocidade máxima estabelecida para a aeronave é de 40 cm/s para cada eixo horizontal e de 50 cm/s na vertical . O "força pouso" é ativado a uma distância de 1 m. Os resultados de ambos os algoritmos são apresentados na Figura 46. As reduções dos erros de distância relativa ao local de pouso são apresentadas na Figura 47.

A Tabela 12 apresenta o valor médio e o desvio padrão da distância radial ao ponto central do marcador. São realizadas 30 repetições de aterrissagens para cada estratégia de

Figura 44 – Redução dos erros ΔX, ΔY e ΔZ do VANT em relação ao local de pouso para o marcador estático em SITL



(a) *Fuzzy*: Erro ΔX

(b) RNA: Erro ΔX

(c) *Fuzzy*: Erro ΔY

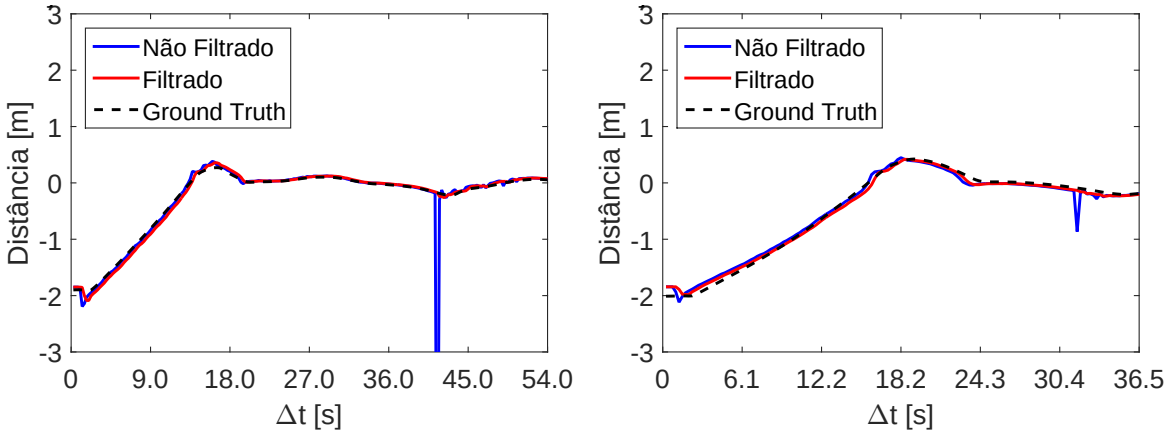
(d) RNA: Erro ΔY

(e) *Fuzzy*: Erro ΔZ

(f) RNA: Erro ΔZ

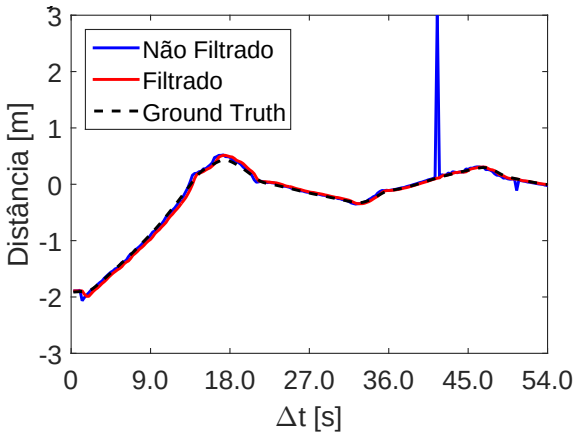
Fonte: Elaborada pelo autor

Figura 45 – Posição do local de pouso estático em SITL com referência ao VANT

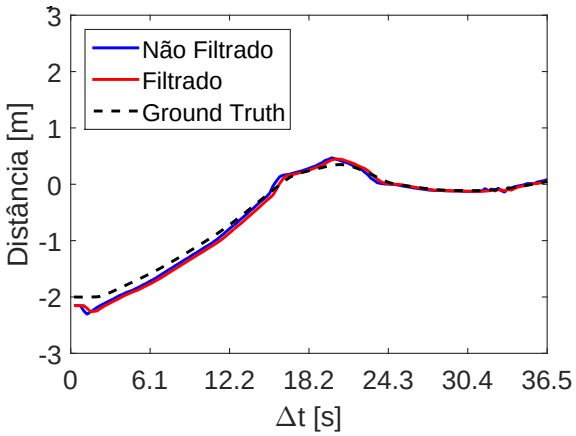


(a) *Fuzzy*: Posição X

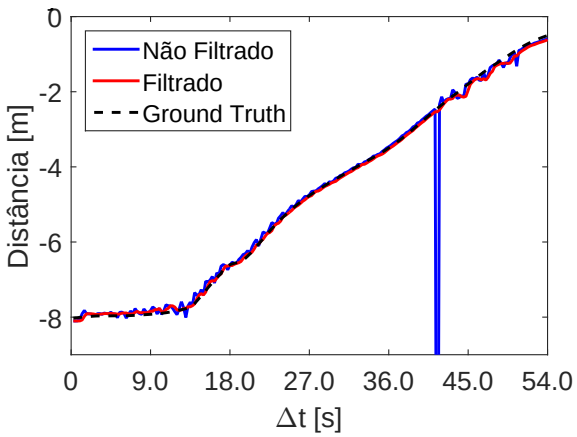
(b) RNA: Posição X



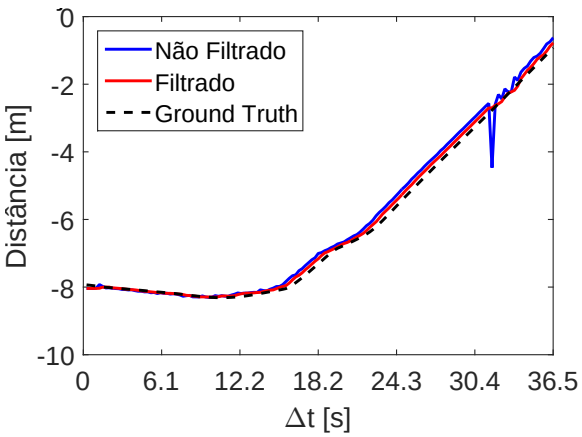
(c) *Fuzzy*: Posição Y



(d) RNA: Posição Y



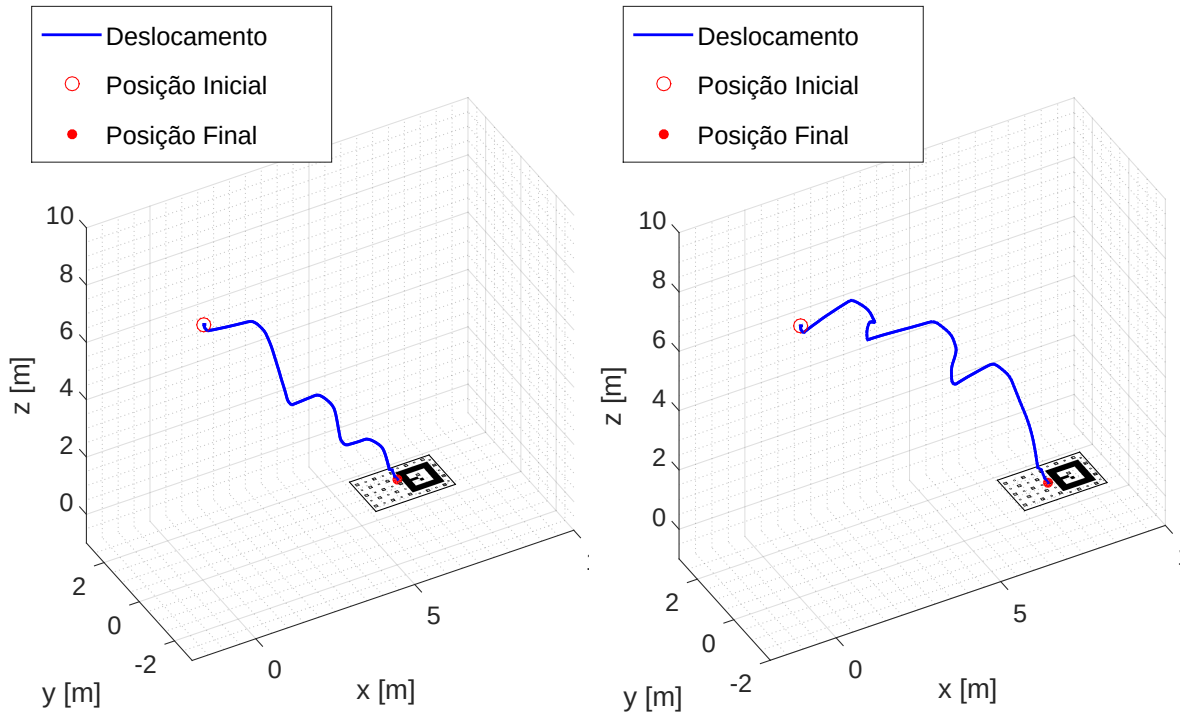
(e) *Fuzzy*: Posição Z



(f) RNA: Posição Z

Fonte: Elaborada pelo autor

Figura 46 – Pouso em alvo dinâmico retilíneo em SITL



(a) *Fuzzy* (b) RNA

Fonte: Elaborada pelo autor

controle. As posições iniciais de cada repetição, e velocidades adotadas, mantiveram-se iguais as discutidas. Todas as aterrissagens atingiram a região limitada pelo marcador de pouso.

Tabela 12 – Erro radial em relação ao centro do marcador de pouso com movimento retilíneo

Algoritmo	**Erro Radial Médio [m]**	**Desvio Padrão [m]**
Fuzzy	0,224	0,101
RNA	0,243	0,081

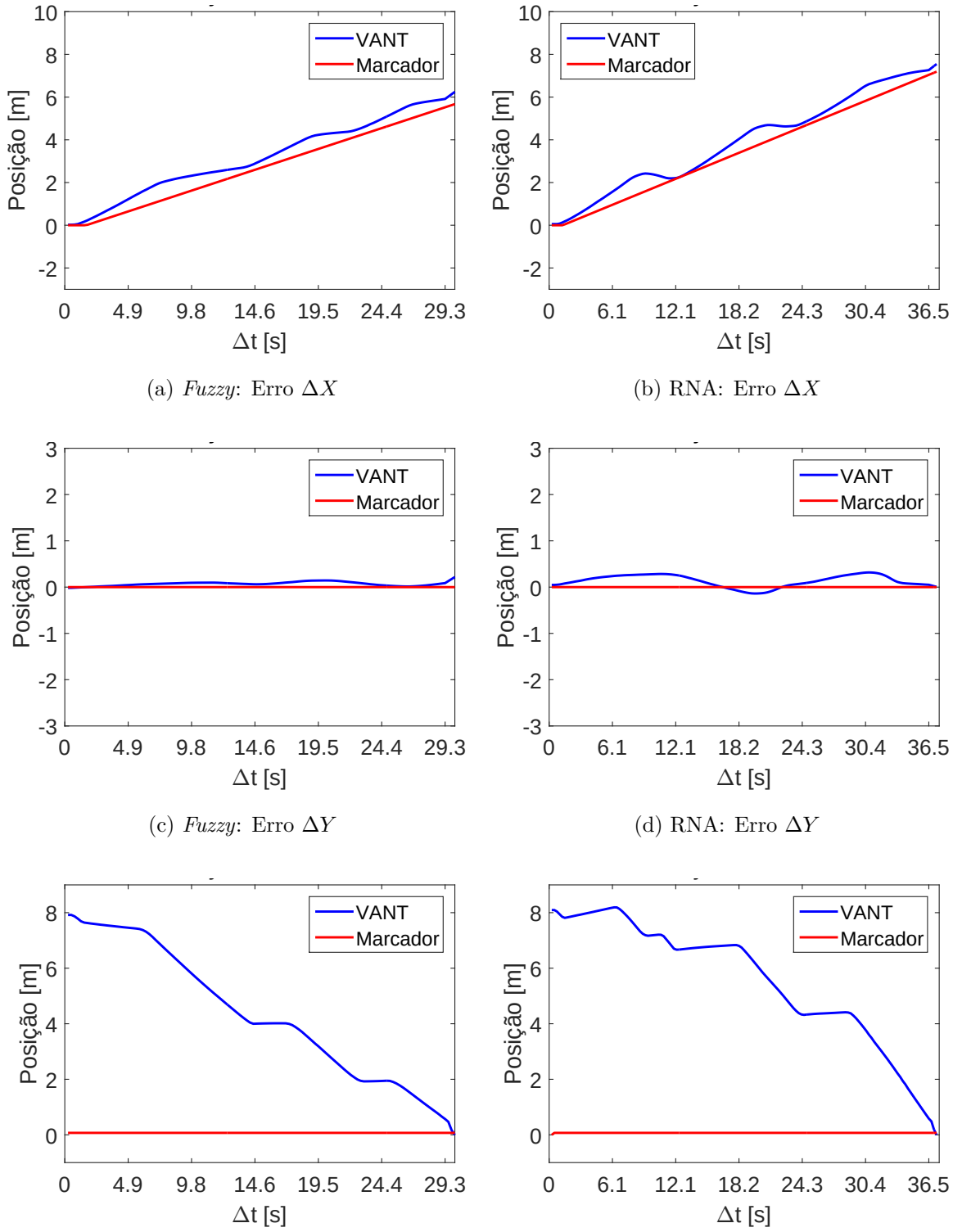
Fonte: Elaborada pelo autor

O posicionamento do local de pouso, para o caso de marcador com movimento retilíneo é ilustrado pela Figura 48. A estratégia adotada é discutida na Seção 3.4. Nessa figura são apresentados o posicionamento do marcador em relação a aeronave, filtrado e não filtrado, além do *Ground Truth* informado pelo Gazebo.

4.3.3 Caso 3: Alvo Dinâmico com Movimento Circular

O terceiro teste consiste em dispor a aeronave em uma localização relativa ao local de pouso de 8 m de altura. Assim que o algoritmo de aterrissagem é ativado, o marcador desloca com uma velocidade retilínea uniforme de 20 cm/s e uma velocidade angular de

Figura 47 – Redução dos erros ΔX, ΔY e ΔZ do VANT em relação ao local de pouso para o marcador dinâmico com movimento retilíneo em SITL



(a) *Fuzzy*: Erro ΔX

(b) RNA: Erro ΔX

(c) *Fuzzy*: Erro ΔY

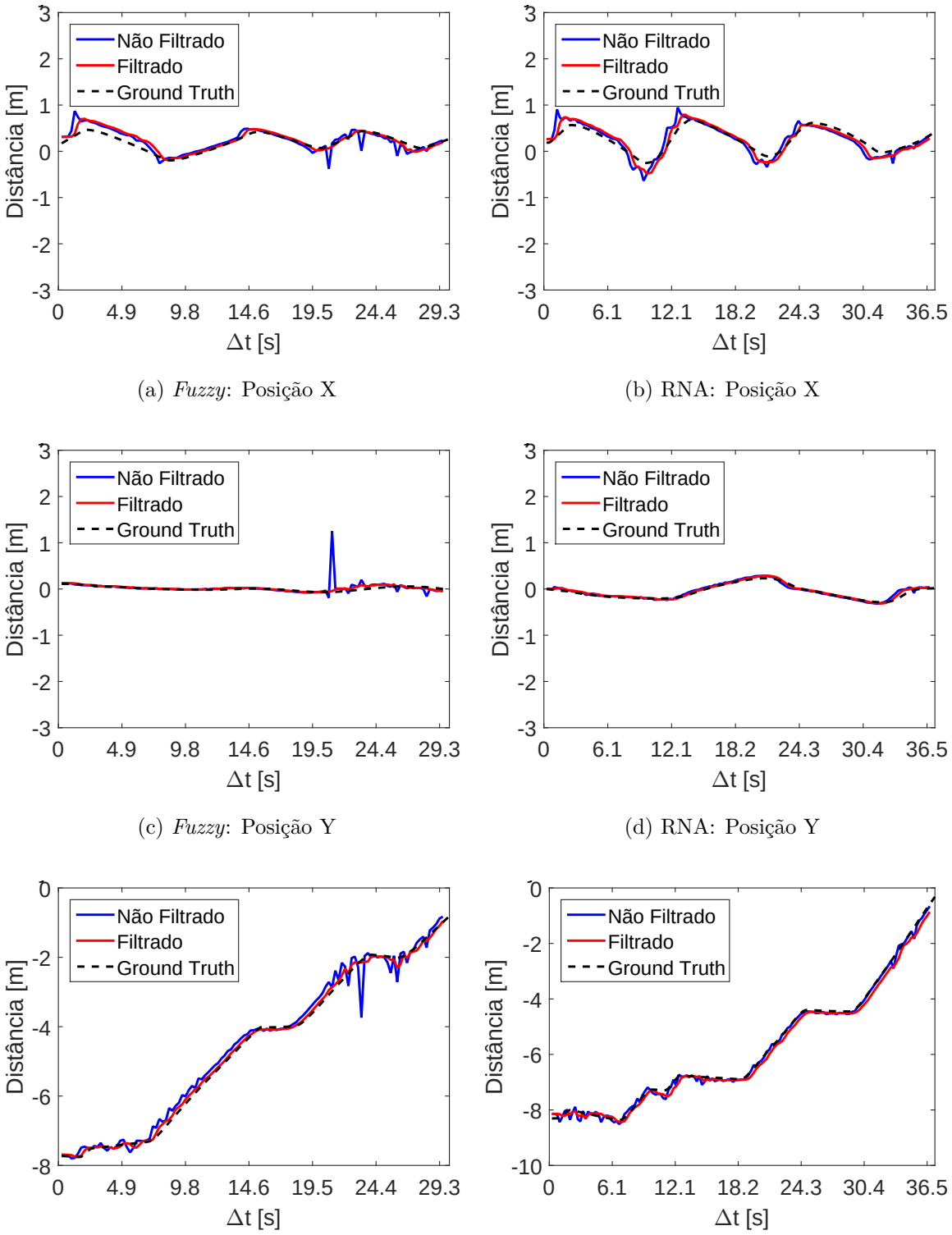
(d) RNA: Erro ΔY

(e) *Fuzzy*: Erro ΔZ

(f) RNA: Erro ΔZ

Fonte: Elaborada pelo autor

Figura 48 – Posição do local de pouso com referência ao VANT para o alvo dinâmico retilíneo em SITL



(a) *Fuzzy*: Posição X

(b) RNA: Posição X

(c) *Fuzzy*: Posição Y

(d) RNA: Posição Y

(e) *Fuzzy*: Posição Z

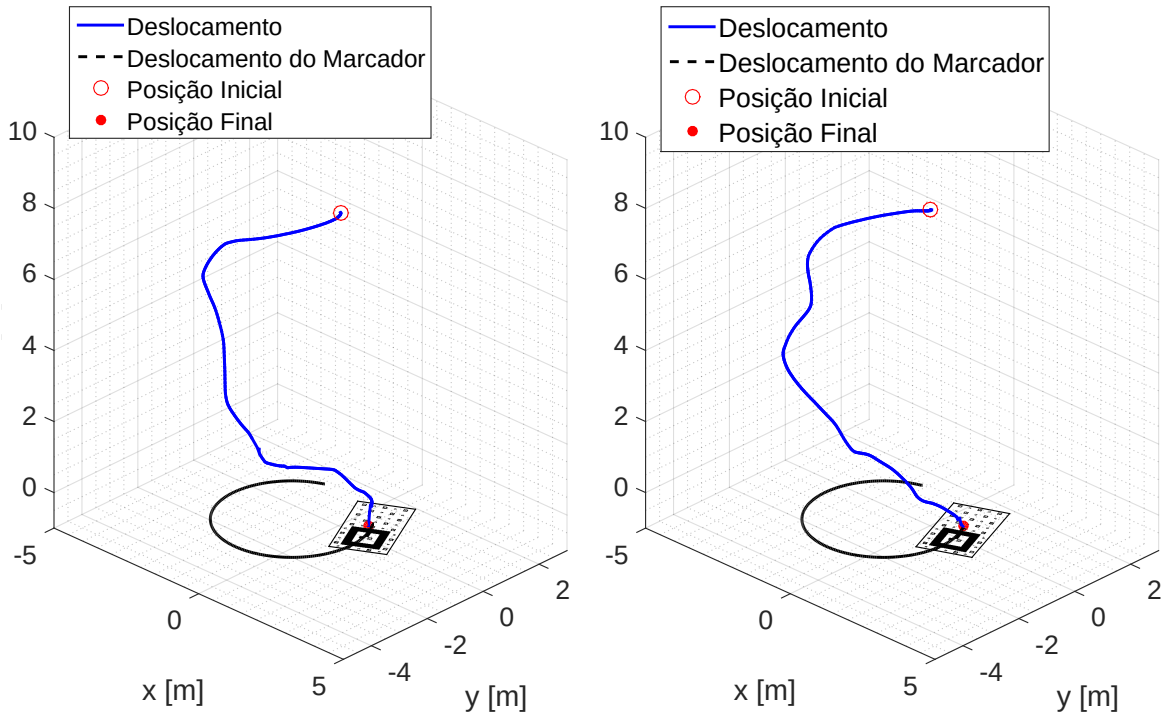
(f) RNA: Posição Z

Fonte: Elaborada pelo autor

10 rad/s. A priori, o marcador é visível dado o posicionamento adotado. A velocidade máxima estabelecida para a aeronave é de 20 cm/s para cada eixo horizontal e de 40 cm/s na vertical . O "força pouso" é ativado a uma distância de 1 m. Os resultados de ambos os algoritmos são apresentados na Figura 49. Os comportamentos da redução do erro tridimensional relativo à distância entre o VANT e o local de pouso são apresentados na Figura 50.

A Tabela 12 apresenta o valor médio e o desvio padrão da distância radial ao ponto central de aterrissagem. São realizadas 30 repetições de aterrissagens para cada estratégia de controle. As posições iniciais de cada repetição e velocidades adotadas, mantiveram-se iguais as discutidas. Todas a aterrissagens atingiram a região limitada pelo marcador de pouso.

Figura 49 – Pouso em alvo dinâmico com movimento circular em SITL



(a) *Fuzzy* (b) RNA

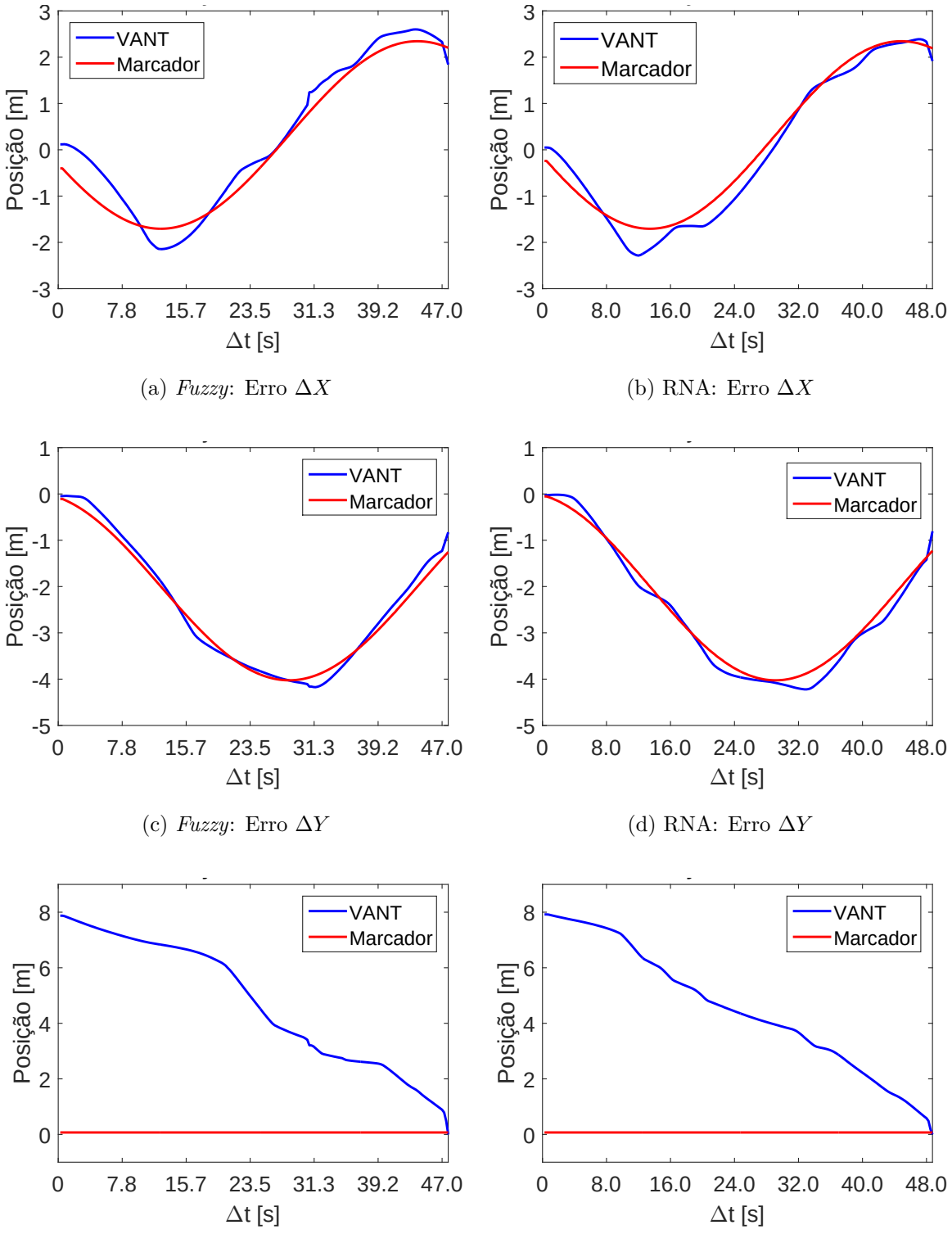
Fonte: Elaborada pelo autor

Tabela 13 – Erro radial em relação ao centro do marcador de pouso com movimento circular

Algoritmo	Erro Radial Médio [m]	Desvio Padrão [m]
Fuzzy	0,404	0,099
RNA	0,406	0,088

Fonte: Elaborada pelo autor

Figura 50 – Redução dos erros ΔX, ΔY e ΔZ do VANT em relação ao local de pouso para o marcador dinâmico com movimento circular



(a) *Fuzzy*: Erro ΔX

(b) RNA: Erro ΔX

(c) *Fuzzy*: Erro ΔY

(d) RNA: Erro ΔY

(e) *Fuzzy*: Erro ΔZ

(f) RNA: Erro ΔZ

Fonte: Elaborada pelo autor

O posicionamento do local de pouso, determinado pela estratégia adotada da Seção 3.4, são ilustrados pela Figura 51. Nessa figura são apresentados o posicionamento do marcador em relação a aeronave, puro e filtrado, além do *Ground Truth* informado pelo Gazebo.

O vídeo com os experimentos em SITL pode ser acessado no *link* abaixo:

<https://goo.gl/AN1HZ5>

4.3.4 Caso 4: Obstáculos

O procedimento de aterrissagem quando o marcador de pouso não é encontrado, a priori, é apresentado na Figura 52a. Nesse exemplo, o VANT é posicionado inicialmente onde o local de pouso não é visível, porém se encontra próximo. O nó *SeekMarker*, apresentado na Seção 3.5, entra em execução (dado α igual a 0) até que seja detectado o marcador e a aterrissagem seja executada.

Um segundo procedimento é testado, Figura 52b. Nesse caso, existe a inserção de obstáculos que impedem a detecção do local de pouso durante a aterrissagem. Mais especificamente, o marcador em movimento retilíneo se desloca abaixo de um conjunto de caixas durante um trecho de seu deslocamento. O nó *SeekMarker* (dado α igual a 1) retoma a altura de segurança da aeronave, deslocando-a até a localização da última detecção do marcador, em seguida, executa o procedimento de busca relatado na Seção 3.5. Quando o marcador de pouso é identificado, a aterrissagem é retomada.

Os testes são realizados utilizando a RNA como algoritmo de aterrissagem em SITL. O vídeo dos procedimentos discutidos na presente seção pode ser visto em:

<https://goo.gl/5focsP>

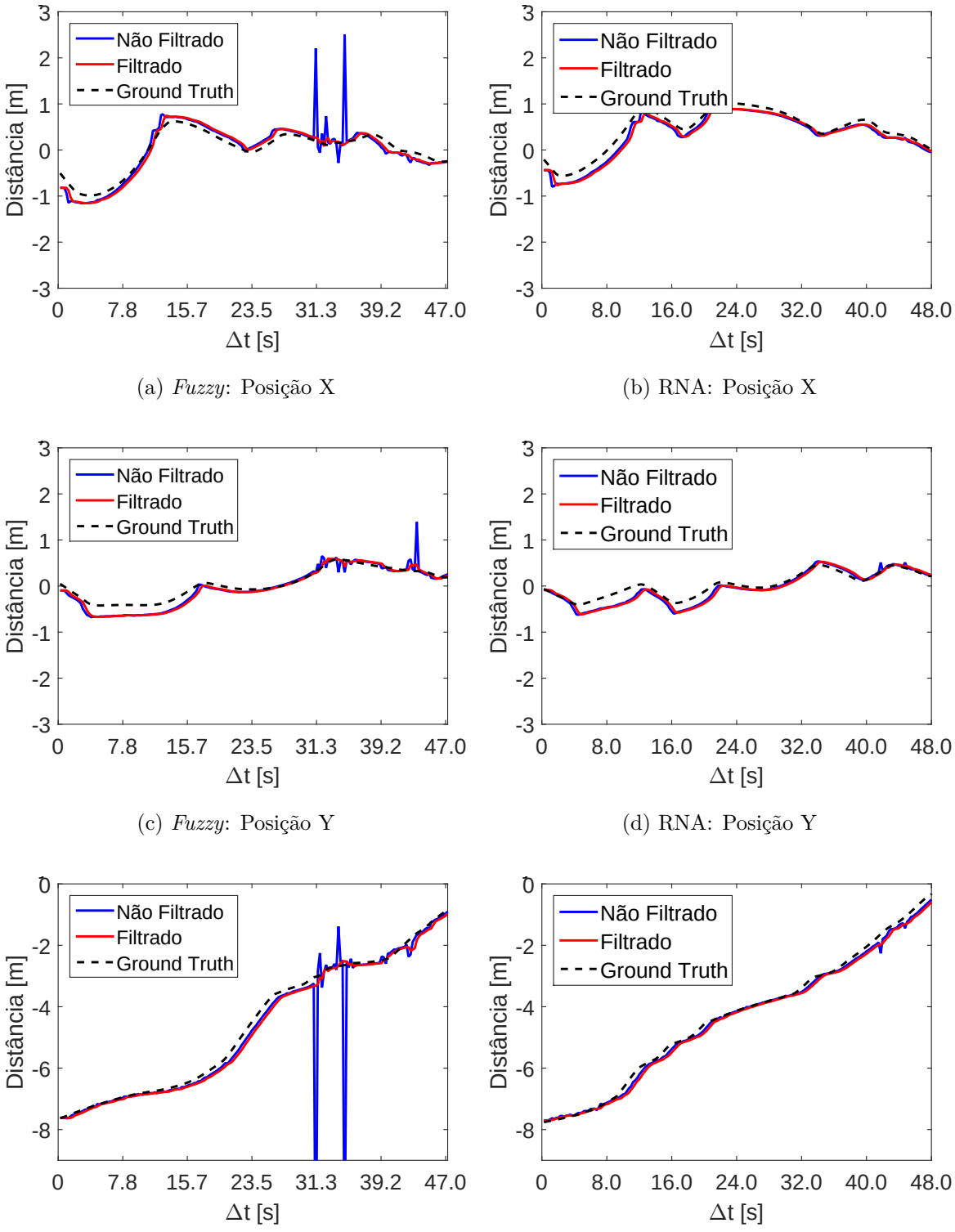
4.3.5 Teste de Hipóteses para Aterrissagem

O teste-t de duas amostras permite inferir suposições de duas amostras de dados independentes e verificar estatisticamente a sua validade. Este teste estatístico é representado pela Equação 4.1 (88, 89).

$$t = \frac{\overline{x} - \overline{y}}{\sqrt{\frac{{s_x}^2}{n} + \frac{{s_y}^2}{m}}}. \tag{4.1}$$

em que $\overline{x}$ e $\overline{y}$ são os valores médios das amostras X e Y e s_x e s_y são os desvios padrão das amostras. As variáveis n e m correspondem ao tamanho dos conjuntos de amostra X e Y, respectivamente. O grau de liberdade é definido como $n + m - 2$. Além da determinação de t, torna-se importante a inferência das hipóteses. As hipóteses são apresentadas na Equação 4.2.

Figura 51 – Posição do local de pouso com referência ao VANT para o alvo dinâmico circular em SITL



(a) *Fuzzy*: Posição X

(b) RNA: Posição X

(c) *Fuzzy*: Posição Y

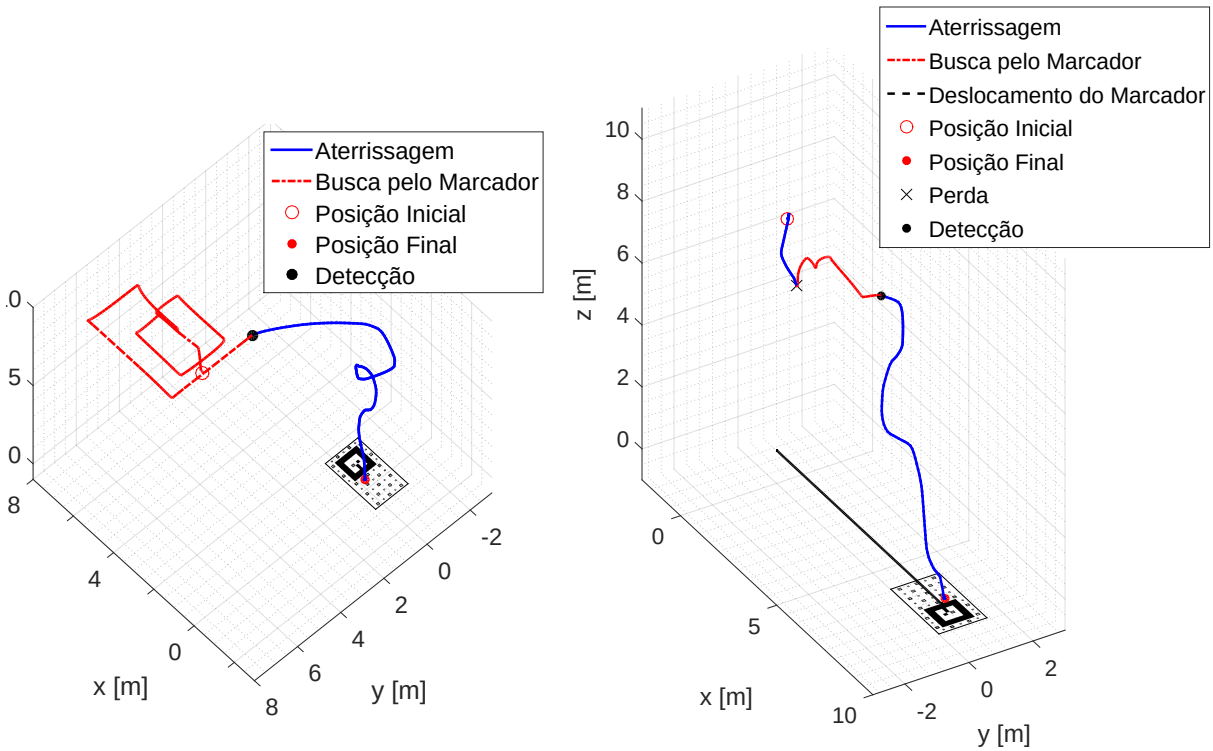
(d) RNA: Posição Y

(e) *Fuzzy*: Posição Z

(f) RNA: Posição Z

Fonte: Elaborada pelo autor

Figura 52 – Pouso com obstáculos



(a) *Caso* $\alpha = 0$ (b) Caso $\alpha = 1$

Fonte: Elaborada pelo autor

$$\begin{cases} H_0 : \overline{x} = \overline{y} \\ H_1 : \overline{x} \neq \overline{y} \end{cases} \tag{4.2}$$

Dado um nível de significância $\alpha_{significância}$, usualmente 0,05, o p-valor é calculado a partir de t, e representa o menor valor de $\alpha_{significância}$ para se rejeitar a hipótese nula (H_0), Assim, um p-valor abaixo de $\alpha_{significância}$ evidencia que a hipótese nula não é verdadeira (88, 89).

Para o problema de aterrissagem, objetiva-se através do teste-t verificar se as médias dos erros radiais médios obtidos pelos experimentos com a lógica *fuzzy* e a RNA são estatisticamente equivalentes. Para isso, os conjuntos de amostras X e Y da Equação 4.1 são os dados de erros radiais obtidos para os 30 pousos realizados em SITL para os algoritmos *fuzzy* e RNA, respectivamente. Para tal, foram considerados os seguintes experimentos: alvo estático e alvo com movimento retilíneo e circular. Como o número de graus de liberdade é considerado alto (igual a 58), logo não se torna necessária a verificação da normalidade da distribuição dos erros (88, 89). Na Tabela 14 são apresentados os p-valores calculados a partir do *software Matlab*. Vale ressaltar que foi utilizado $\alpha_{significância} = 0,05$. Se o p-valor de um teste for inferior a $\alpha_{significância}$, o teste rejeita a hipótese nula. Se o p-valor for maior do que $\alpha_{significância}$, não há provas suficientes para rejeitar a hipótese

nula. Com isso, é possível afirmar que o modelo baseado em RNA foi capaz de absorver a inteligência intrínseca do modelo *fuzzy*, validando a proposta da presente dissertação.

Tabela 14 – p-valor dado $\alpha_{significância} = 0,05$

Tipo de Movimento do Local de Pouso	**Estático**	**Movimento Retilíneo**	**Movimento Circular**
***p*-valor**	0,2911	0,4571	0,9643

Fonte: Elaborada pelo autor

4.3.6 Discussões

Baseando-se nas aterrissagens relatadas (Figuras 43, 46 e 49), no comportamento dos erros (Figuras 44, 47 e 50) e nas Tabelas 11, 12 e 13 é possível verificar que tanto o algoritmo *fuzzy* quanto a RNA obtiveram êxito no procedimento, já que o objetivo é realizar a aterrissagem da aeronave dentro dos limites do marcador de pouso. O erro da posição de pouso, destacado pelas tabelas para ambos algoritmos adotados, é pequeno em comparação às dimensões do marcador. Obviamente as velocidades máximas interferem nesse desempenho. O VANT deve ter uma velocidade relativa maior em relação ao movimento do local de pouso. Como não se utilizou de Gimbal como estrutura de câmera, a velocidade máxima também não pode ser muito alta. O limite verificado em testes foi de até $70cm/s$. Porém, para a aterrissagem essa magnitude de velocidade não é útil, tornando o procedimento arriscado. Essa restrição de velocidade é devido à perda da região de visão durante os movimentos de *pitch* e *roll* executados pela aeronave.

Através dos comportamentos de aterrissagem verificados pelos experimentos, infere-se que a RNA foi capaz de absorver a técnica de controle de aterrissagem da lógica *fuzzy*. Em linhas gerais, o erro radial médio da aterrissagem para a RNA foi ligeiramente maior para os casos estático e de movimento retilíneo (Tabelas 11 e 12) e basicamente igual para o movimento circular (Tabela 13). Os desvios padrão também se aproximaram. Apesar das diferenças das médias, o teste-t (Seção 4.3.5) não obteve provas suficientes para rejeitar a equivalência estatística entre as médias dos erros gerados pelas aterrisagens dos algoritmos RNA e *fuzzy*. Vale lembrar que foram executadas 30 amostras de testes. É verificado que o VANT apresenta maiores dificuldades, devido a sua dinâmica e inércia, em comutar velocidades compostas em ambos eixos com uma frequência elevada. Dessa forma, os erros para alvo com movimento circular foram maiores.

O erro inerente aos pousos executados é a distância da posição de aterrissagem atingida pelo centro de massa do VANT em relação ao ponto de pouso, determinado pelo marcador de ID 34. Vale lembrar que o veículo possui as dimensões de $45,0$ cm de comprimento por $62,5$ cm de largura. O sensoriamento realizado pelo veículo para o seu controle de baixo nível apresenta erros que acarretam na atuação da aeronave. Além disso, a margem não determinística dos grupos *fuzzy* permite uma tolerância para execução da aterrissagem, caso contrário, devido as incertezas, o VANT não iria finalizar o

procedimento. A rotina de “força pouso”, discutida na Seção 3.5, também contribui para o erro, já que por um breve instante de tempo a descida é executada em malha aberta.

O nó *SeekMarker* permitiu que todos os testes executados convergissem para dentro do marcador de pouso, uma vez que retomava o procedimento de busca caso o marcador ficasse fora do alcance da visão computacional. O seu desempenho foi verificado na Seção 4.3.4.

Quanto à abordagem de identificação do local de pouso, discutida na Seção 3.4, por intermédio dos gráficos das Figuras 45, 48 e 51, infere-se que o filtro adotado foi capaz de atender as necessidades de impedir que leituras errôneas fossem aplicadas nos algoritmos de pouso propostos. Vale ressaltar, que o atraso foi pequeno não atrapalhando o procedimento de aterrissagem. Além do erro presente no *Ar Track Alvar*, a irregularidade das medidas, principalmente após os segundos finais, é devido ao constante chaveamento de detecção dos marcadores. Os pequenos erros de estimativa do *Ar Track Alvar* são propagados através das transformações relatadas nas Seções 3.3 e 3.4.

4.4 ESTUDO DE CASO: EXPERIMENTO REAL

Para a execução do experimento de aterrissagem em ambiente externo, são utilizados os equipamentos relatados na Seção 3.9. O local de pouso, confeccionado no software *Corel Draw*®, é impresso com as especificações relatadas na Seção 3.4 sobre um *banner*. A Figura 53 apresentam a configuração de teste.

Figura 53 – Configuração de teste

Fonte: Elaborada pelo autor

O VANT é pilotado até uma altura fixa e deslocado em relação ao local de pouso para ambos os algoritmos de aterrissagem nos dois testes: local de pouso estático e

dinâmico, Figuras 54a e 54b, respectivamente. Vale ressaltar que por se tratar de um experimento real, o veículo apresenta erros em seu posicionamento inicial. O *firmware* é executado assim que a aeronave é ligada, através do *script* abordado na Seção 3.9.2. O procedimento de aterrisagem é iniciado quando o modo de voo *OFFBOARD* é ativado pelo rádio controle da aeronave. O *ground truth* utilizado é a leitura de posição adquirida pela própria aeronave, cuja estimativa é realizada por leituras de GPS e IMU.

Os dados gerados são gravados em arquivos na própria *Odroid* e nenhum código de atuação do *firmware* de aterrissagem proposto é executado em estação base. Ao término dos experimentos os *logs* gerados são coletados.

Figura 54 – Aterrissagens do caso prático

(a) Pouso em alvo estático

(b) Pouso em alvo dinâmico

Fonte: Elaborada pelo autor

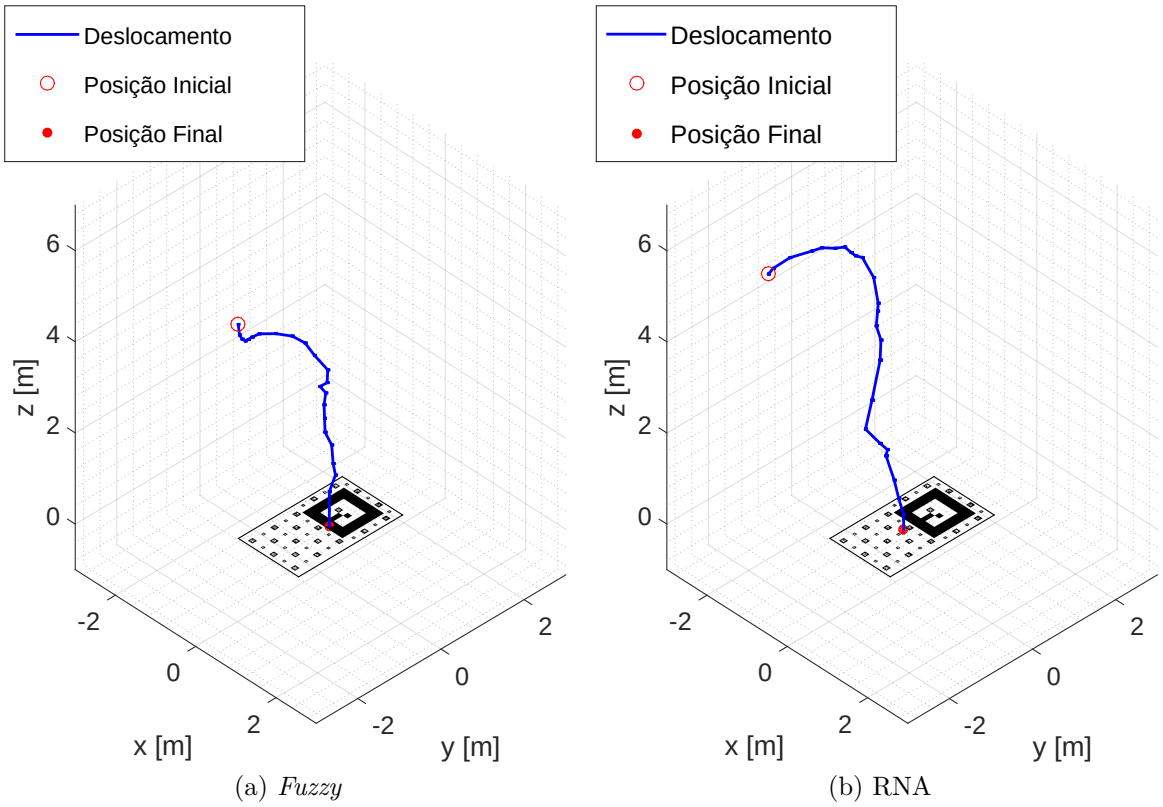
4.4.1 Caso 1: Alvo Estático

Os experimentos com o local de pouso estático são realizados em um dia com céu nublado e ventos medianos, Figura 54a. Os procedimentos de pouso, tanto para o algoritmo *fuzzy* quanto RNA, são apresentados na Figura 55. São adotadas velocidades de 20 cm/s para todos os eixos de atuação e o "força pouso" utilizado é de 50 cm. O marcador não apresenta velocidade relativa. Os comportamentos das reduções do erro tridimensional relativo à distância entre o VANT e o local de pouso são apresentados na Figura 56. A Figura 57 ilustra o posicionamento do local de pouso, determinado pela estratégia adotada da Seção 3.4.

O vídeo com os experimentos reais com alvo estático encontra-se no *link* abaixo:

<https://goo.gl/91jM8V>

Figura 55 – Pouso em alvo estático



(a) *Fuzzy* (b) RNA

Fonte: Elaborada pelo autor

4.4.2 Caso 2: Alvo Dinâmico

Os experimentos envolvendo o local de pouso em movimento foi realizado em um dia sem nuvens e ventos medianos. Para a movimentação do local de pouso é adotado o robô terrestre diferencial P3DX da empresa *Pioneer* como rebocador. O marcador de aterrissagem se encontrava sobre uma estrutura de madeira com rodas, Figura 54b.

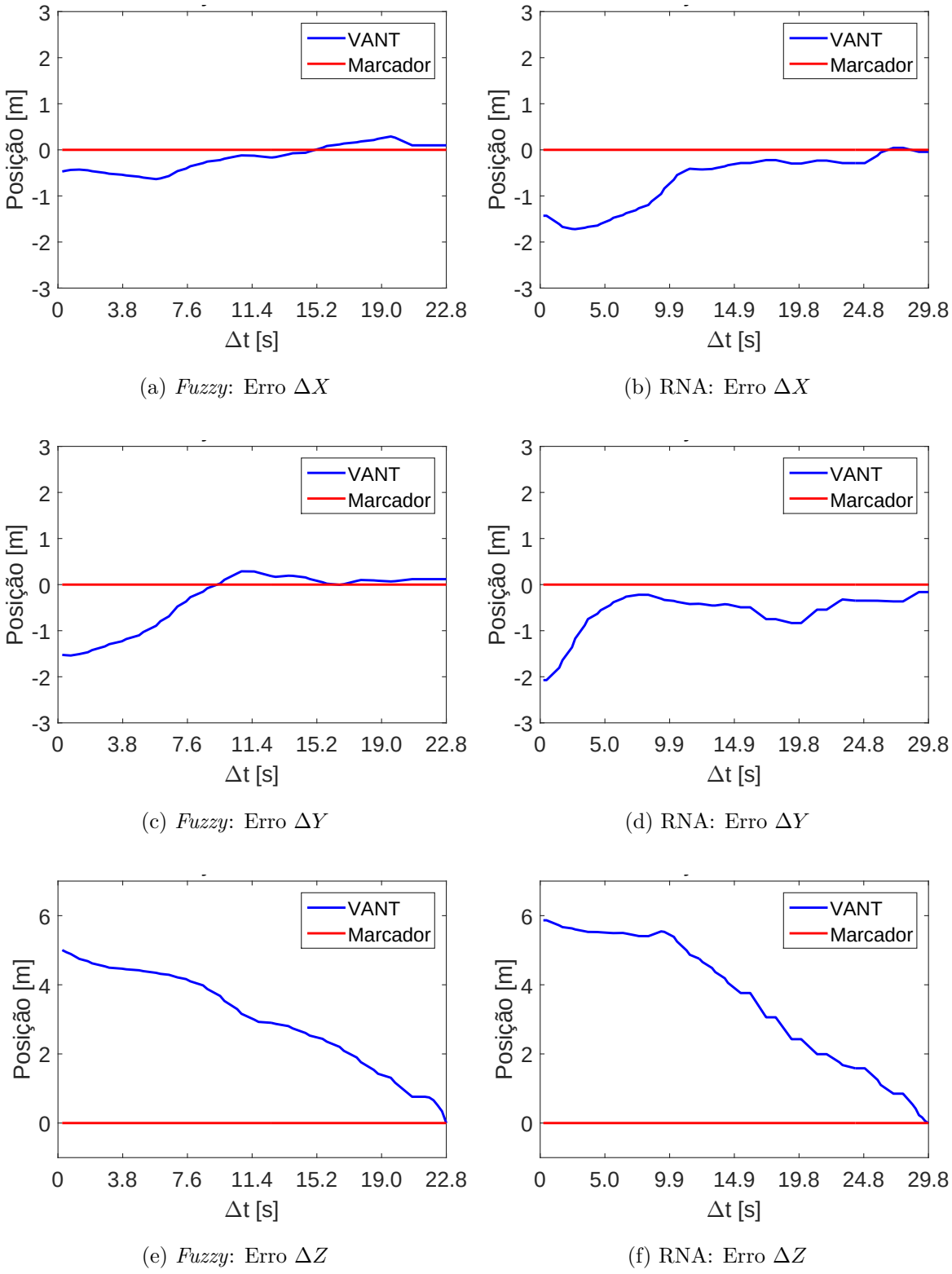
Para o VANT são adotadas as velocidades horizontais e verticais de 50 cm/s e 30 cm/s, respectivamente. Para o local de pouso é utilizada uma velocidade linear de 30 cm/s. O "força pouso" é de 50 cm. O procedimento de pouso, tanto para o algoritmo *fuzzy* quanto RNA, são apresentados na Figura 58

Os comportamentos das reduções do erro tridimensional relativo à distância entre o VANT e o local de pouso dinâmico são apresentados na Figura 59. A Figura 60 ilustra o posicionamento do local de pouso.

O vídeo com os experimentos reais com alvo dinâmico encontra-se no *link* abaixo:

<https://goo.gl/91jM8V>

Figura 56 – Redução dos erros ΔX, ΔY e ΔZ do VANT em relação ao local de pouso para o marcador estático



(a) *Fuzzy*: Erro ΔX

(b) RNA: Erro ΔX

(c) *Fuzzy*: Erro ΔY

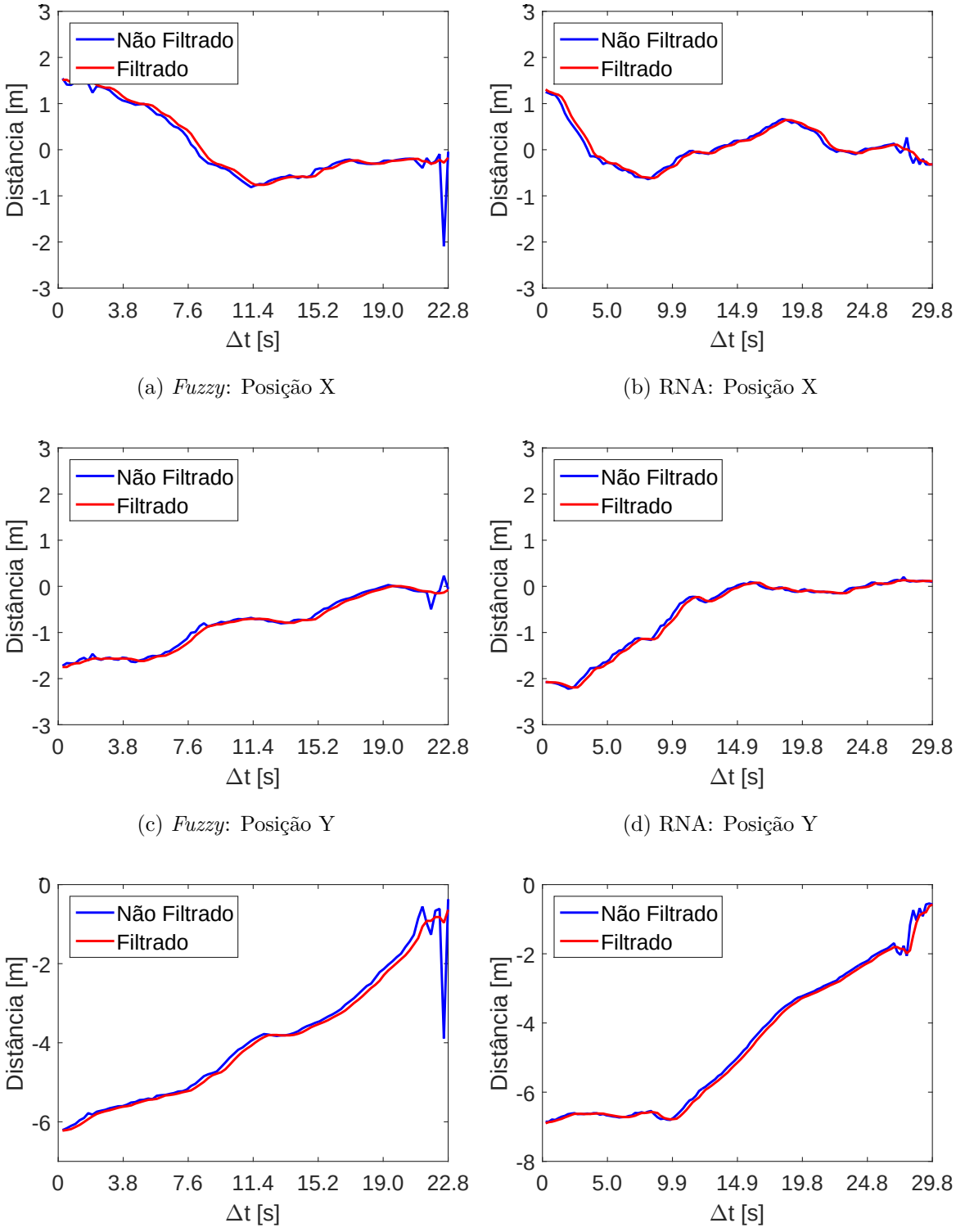
(d) RNA: Erro ΔY

(e) *Fuzzy*: Erro ΔZ

(f) RNA: Erro ΔZ

Fonte: Elaborada pelo autor

Figura 57 – Posição do local de pouso estático com referência ao VANT



(a) *Fuzzy*: Posição X

(b) RNA: Posição X

(c) *Fuzzy*: Posição Y

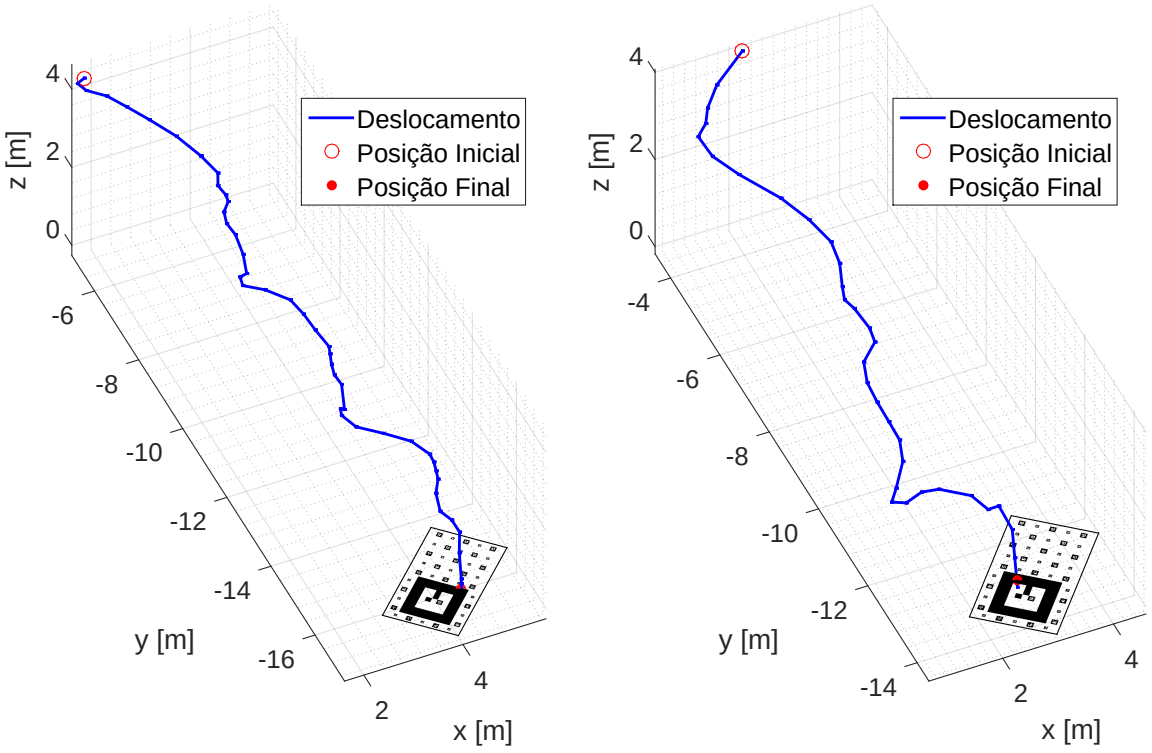
(d) RNA: Posição Y

(e) *Fuzzy*: Posição Z

(f) RNA: Posição Z

Fonte: Elaborada pelo autor

Figura 58 – Pouso em alvo dinâmico



(a) *Fuzzy* (b) RNA

Fonte: Elaborada pelo autor

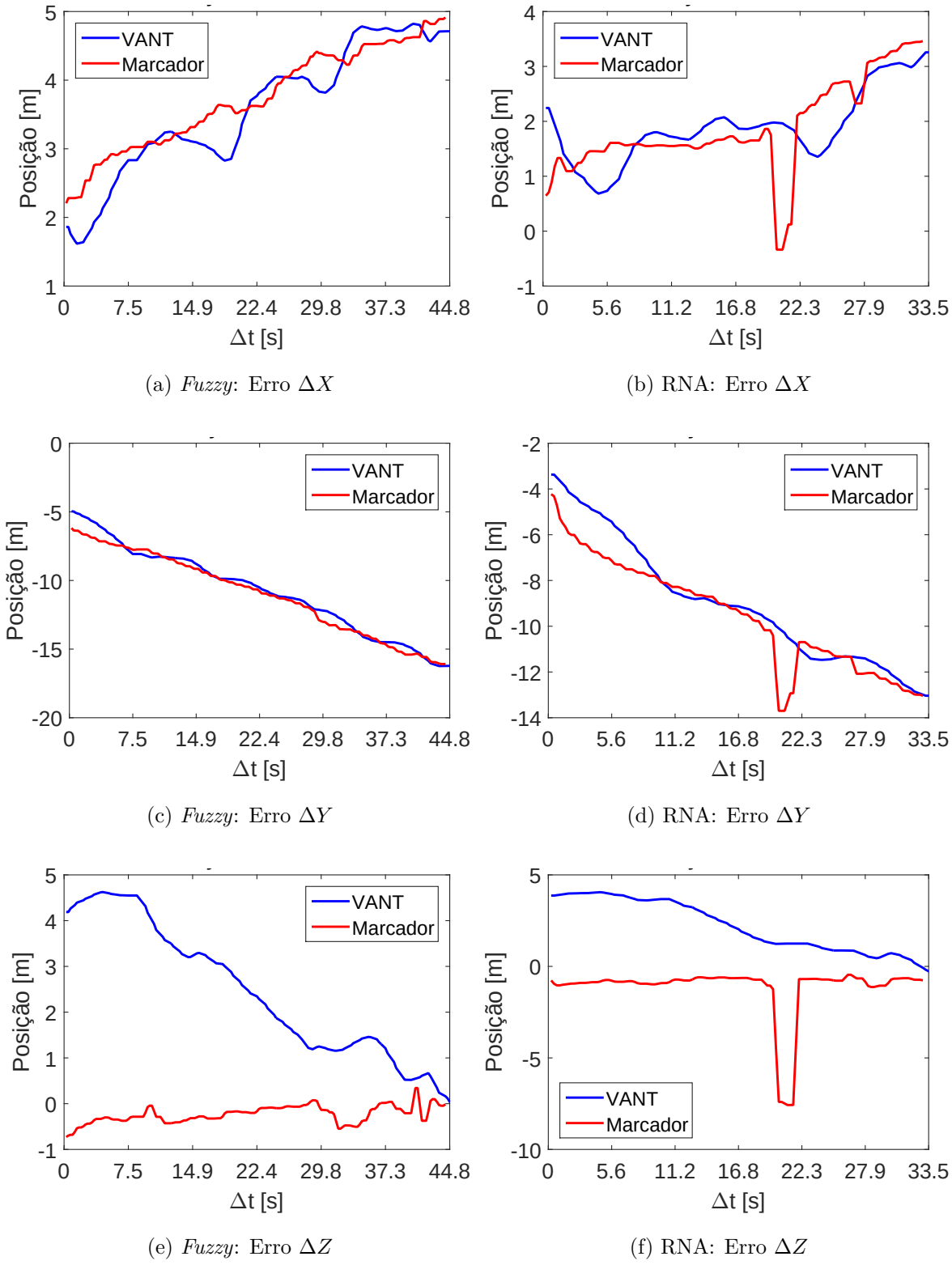
4.4.3 Discussões

Assim como os resultados apresentados em SITL, o *firmware* de aterrissagem proposto utilizando lógica *fuzzy* e RNA obteve sucesso no pouso em alvo estático e dinâmico, Figuras 55, 56, 58 e 59. O VANT foi capaz de aterrissar nos limites do marcador.

As análises de SITL, discutidas na Seção 4.3.6, foram verificadas em prática. O VANT foi capaz de contornar ventos medianos durante sua atuação de decida até o ponto de aterrissagem. Para o experimento com alvo dinâmico, a orientação do local de pouso devido ao declive lateral no ambiente de testes não prejudicou o procedimento de aterrissagem, mostrando a capacidade de pouso dos algoritmos. Foi verificada também a movimentação suave do veículo. O "força pouso" exerceu o seu papel para finalização da missão, sem grande perda de precisão, dado que sua execução foi realizada relativamente perto do local de pouso. O controle de baixo nível, executado pelo *firmware PX4*, também atendeu as espectativas.

Os procedimentos de aterrisssagem, tanto lógica *fuzzy* quanto RNA, na prática não se diferiram em comportamento. Assim, a proposta desta dissertação é validada em experimento real, já que a rede neural conseguiu absorver o projeto da lógica *fuzzy* dado

Figura 59 – Redução dos erros ΔX, ΔY e ΔZ do VANT em relação ao local de pouso para o marcador dinâmico



(a) *Fuzzy*: Erro ΔX

(b) RNA: Erro ΔX

(c) *Fuzzy*: Erro ΔY

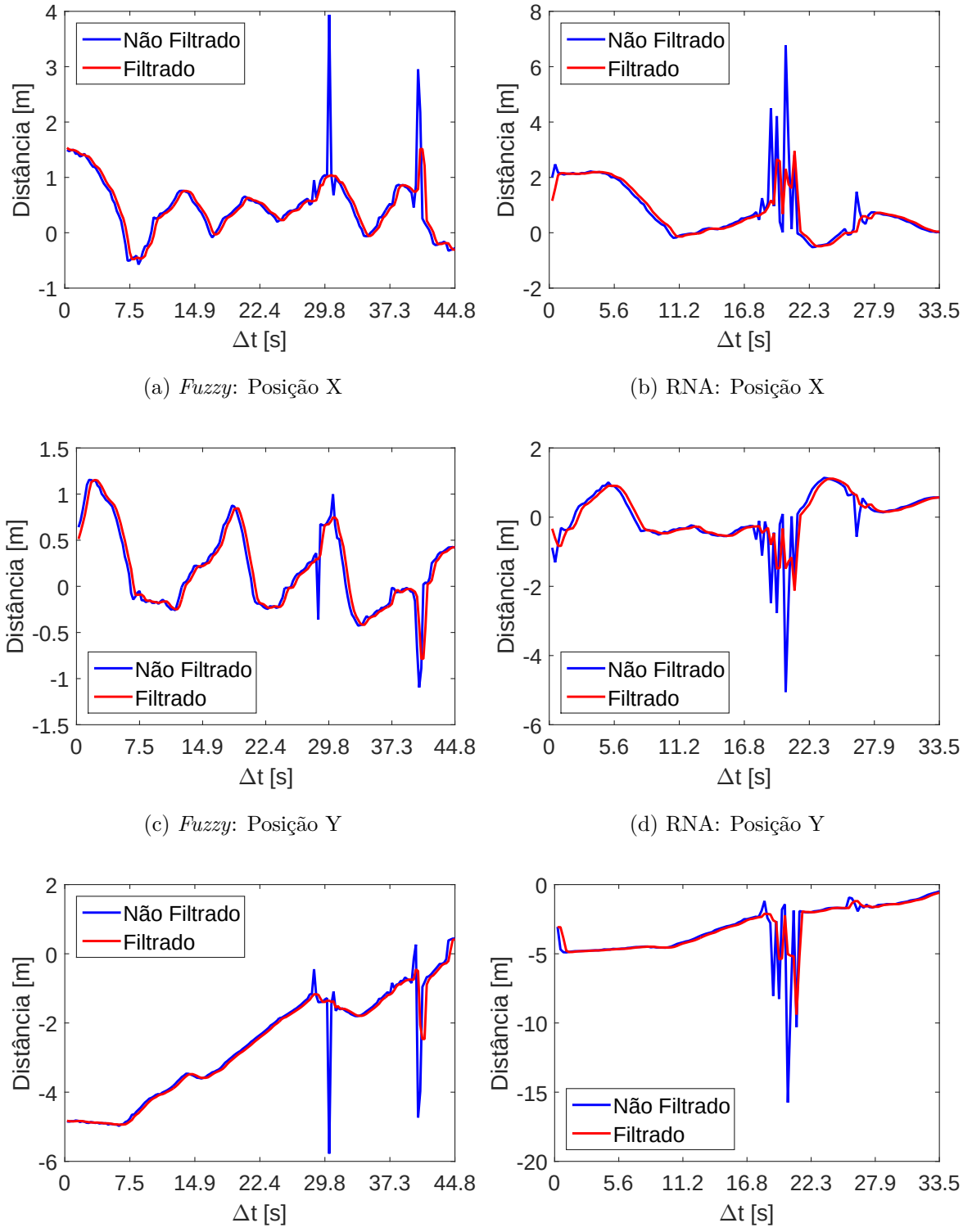
(d) RNA: Erro ΔY

(e) *Fuzzy*: Erro ΔZ

(f) RNA: Erro ΔZ

Fonte: Elaborada pelo autor

Figura 60 – Posição do local de pouso dinâmico com referência ao VANT



(a) *Fuzzy*: Posição X

(b) RNA: Posição X

(c) *Fuzzy*: Posição Y

(d) RNA: Posição Y

(e) *Fuzzy*: Posição Z

(f) RNA: Posição Z

Fonte: Elaborada pelo autor

o objetivo de pouso autônomo. Essa capacidade de absorção da inteligência *fuzzy* foi verificada também em SITL, Seção 4.3.

Quanto ao *firmware* embarcado, com o uso da *Odroid* como estação *onboard* de processamento, permite-se uma maior segurança sem perdas de comunicação entre as entidades processadoras. Vale lembrar também que essa proposta impede que a aeronave esteja sempre próxima a uma estação base processadora, ganhando maior liberdade em suas missões. Outro ponto importante é o custo, já que estações bases de processamento, em geral, são mais caras e necessitam de um *link* de comunição sem fio.

O ROS, concomitante ao protocolo MAVLINK (por intermédio do *MAVROS*), atingiu o seu objetivo no processamento grafo e troca de informações com o controle de baixo nível da aeronave. Assim o *firmware* de aterrissagem foi executado.

A estratégia de detecção do local de pouso, Seção 3.4, também foi verificada no experimento real, Figuras 57 e 60. Assim como o analisado em SITL, o *Ar Track Alvar* apresentou erros devido ao constante chaveamento dos IDs vistos pelo algoritmo. A filtragem por mediana foi satisfatória, impedindo que leituras errôneas fossem aplicadas sobre os algoritmos de controle. Como um compromisso entre atraso e filtragem foi adotado, em momentos de ruídos constantes, o filtro não eliminou completamente, porém conseguiu minimizar consideravelmente essas leituras, permitindo a retomada da detecção. Esse fato pode ser verificado nas Figuras 61b, 61d e 61f, entre os segundos 20 e 22. O atraso do filtro não afetou o procedimento de aterrissagem.

5 CONCLUSÃO

Esta dissertação teve como principal objetivo o desenvolvimento e aplicação de técnicas de aterrisagem embarcadas em VANTs, sem a necessidade de estações bases de processamento. Mais especificamente é proposto um *firmware* de aterrissagem de VANTs utilizando visão computacional para a detecção do local de pouso. O algoritmo de pouso é executado por uma Rede Neural Artificial. O método de treinamento da RNA também é um dos pontos abordados no trabalho, cuja elaboração e projeto conta com a supervisão de um controlador *fuzzy*.

São apresentadas no Capítulo 2 as ferramentas e fundamentação técnica necessárias para a execução do *firmware* de aterrisagem proposto. São abordados o *Robot Operation System*, plataforma e *framework* que permite o gerenciamento e a comunicação dos nós de processamento dos algoritmos relativos ao pouso da aeronave e o protocolo de comunicação de micro veículos aéreos, o MAVLINK. O MAVLINK exerce o papel fundamental para a troca de dados e informações entre os algoritmos do ROS com o *firmware* de controle aéreo, *PX4*, também abordado no capítulo.

Os pacotes *MAVROS* e *Ar Track Alvar* desempenham papéis importantes, sendo o primeiro citado responsável pela interação do ROS com o protocolo MAVLINK e o segundo responsável pelo sensoriamento e detecção do local de aterrisagem através de marcadores artificiais. Ambos pacotes são apresentados também no Capítulo 2.

Ainda no Capítulo 2, são abordados os conceitos inerentes às técnicas de inteligência artificiais utilizadas nesta dissertação: *fuzzy* e Rede Neural Artificial (RNA).

No Capitulo 3 é relatado toda a proposta de desenvolvimento de *firmware* de aterrisagem. Inicialmente é definida a relação de coordenadas referenciais do problema, dado que sua má formulação desvalidaria todo o sistema. Para a aterrissagem, conta-se com quatro sistemas de coordenadas distintos: inercial, veículo, câmera e local de pouso. Vale ressaltar que a identificação do local de pouso é o principal processo de sensoriamento utilizado na aterrissagem, logo é abordado o processo de criação e confecção desse objetivo. O local de pouso proposto apresenta diferentes IDs de marcadores artificiais, visando a robustez do processo de identificação. Em seguida, a estrutura do *firmware* de aterrissagem é apresentada, com seu sistema multinodal e fluxos de comunicação, além do fluxograma de operação.

Quanto ao algoritmo de aterrissagem, é apresentado inicialmente o projeto do controle *fuzzy* ainda no Capítulo 3. Esse controle é projetado visando a supervisão de treinamento do algoritmo RNA. O motivo da utilização do *fuzzy* como entidade supervisionadora é o seu projeto mais simplificado que impede que possíveis interpretações errôneas sejam inseridas por um banco de dados, ou por uma entidade humana, no processo de treinamento da RNA. Vale ressaltar que é importante a eficiência do controle, uma

vez que ele será utilizado em sistemas embarcados, assim o *fuzzy* é elaborado visando esse critério. Contudo, por ser do tipo Mamdani, o *fuzzy* intrinsecamente é mais complexo computacionalmente em suas tomadas de decisões, logo ele é utilizado no treinamento da RNA. Após o seu treino, a RNA é capaz de absorver o conhecimento da lógica *fuzzy* traduzindo-a em operações matriciais, ou seja, em comparação ao modelo Mamdani, a RNA apresenta um desempenho mais rápido e com custo computacional reduzido. O projeto da Rede Neural Artificial, do tipo *Multilayer Perceptron*, é apresentado no Capítulo 3.

Vale ressaltar que controladores clássicos, como o PID, poderiam apresentar o desempenho computacional mais simplificado, contudo os ajustes de seus ganhos devem ser realizados para cada tipo de aeronave utilizada. O controle de aterrissagem utilizando abordagens clássicas de controle é dependente da dinâmica da aeronave e do tipo de missão em que se deseja aplicar o pouso (47–49).. Assim, o processo torna-se custoso e perigoso, já que se trata de um veículo aéreo. O controle proposto nesta dissertação exige apenas que sejam determinadas velocidades máximas, facilitando seu uso em comparação a controladores clássicos. Vale ressaltar também que os algoritmos propostos são menos susceptíveis a erros de sensoriamento e atuação da aeronave, devido os grupos *fuzzy* não determinísticos.

Visando validar e testar o *firmware* de aterrissagem e ambos algoritmos, *fuzzy* e RNA, são apresentados ao fim do Capítulo 3 o conceito de *Software in the Loop* (SITL) e o protótipo para execução do experimento real. Quanto ao SITL são abordados os conceitos e a modelagem necessária para execução de uma simulação mais próxima da realidade com a utilização do simulador Gazebo e do *firmware PX4* emulado. Quanto ao protótipo são apresentados a confecção e estrutura do VANT utilizado para os experimentos.

No Capítulo 4 foram verificadas e comparadas as abordagens discutidas na dissertação para execução da aterrissagem. Inicialmente os algoritmos *fuzzy* e RNA são comparados em atuação e em processamento computacional. Verifica-se que a premissa de redução da complexidade computacional foi atendida, já que houve a redução do número de instruções executadas e na diminuição do tempo de tomada de decisão do algoritmo RNA em comparação ao *fuzzy*. Em seguida, a execução da aterrissagem é testada em SITL. São propostos 3 casos distintos de movimento do local de pouso: estático, movimento retilíneo uniforme e movimento circular uniforme. Ambos algoritmos executaram com sucesso a aterrissagem sobre a região limitada pelo marcador. Vale ressaltar também que o nó *SeekMarker* exerceu fundamental importância para que as repetições de aterrissagens em SITL alcançassem, em todos os casos, os limites interiores do marcador de pouso. Ainda no Capítulo 4 foram apresentados valores de erros radiais médio e o teste-t. O teste-t, baseado nesses erros, não obteve provas suficientes para rejeitar a equivalência estatística entre as médias dos erros gerados pelas aterrisagens dos algoritmos RNA e *fuzzy*. Foi averiguado, então, que a RNA conseguiu absorver a inteligência intrínseca do

modelo *fuzzy* supervisionador.

O comportamento da detecção do local de pouso também foi discutido no Capítulo 4. Verificou-se que a técnica de diferentes marcadores indicando o local de pouso, adicionalmente à filtragem do sinal, viabilizou a técnica de aterrissagem.

Dado que testes preliminares e em SITL foram satifatórios, os testes com experimentos reais validaram as técnicas e o *firmware* de aterrissagem. O VANT executou o procedimento com sucesso, para alvos de aterrissagem estático e dinâmico. É possível verificar, no Capítulo 4, que a Rede Neural Artificial foi capaz de absorver o comportamento do *fuzzy* na prática. A estratégia de detecção do local de pouso também funcionou como o esperado em SITL. Outro ponto importante é a verificação do processamento embarcado na aeronave. Essa abordagem *onboard* permite a redução de custos, dado que não existe a necessidade de estação base e nem *links* de comunicação. Em adição, o processamento embarcado permite a execução de missões sem a limitação de área de atuação.

A proposta deste trabalho é a execução da aterrissagem em locais de pouso que não se movimentem de maneira exacerbada. Sendo aplicável em barcos em correntezas leves e automóveis com velocidade reduzidas. Velocidades elevadas não são aplicáveis devido a dois fatores: ausência de Gimbal e à abordagem "*real landing*" adotada (30). O Gimbal permite a correção da orientação da câmera em movimento abruptos da aeronave. Esses movimentos são mais intensos em velocidades maiores.

A abordagem denominada "*real landing*" verifica ao longo de todo o processo de aterrissagem o local de pouso, inserindo atrasos no próprio sistema. Essa abordagem se difere da "*virtual landing*" que, após a detecção do local de pouso, a posição do alvo é prevista, permitindo uma atuação mais rápida. O revés dessa abordagem é a possível perda do local de pouso caso a estimativa seja feita erroneamente, ou casos em que imprevisibilidades não sejam consideradas (30).

Verificou-se que a incidência luminosa é um impecilho para o procedimento. Mais especificamente, uma luminosidade não uniforme ao longo do marcador compromete a detecção do local de pouso. Porém, uma luminosidade forte ou fraca, distribuída uniformimente não afetou a detecção. Durante os experimentos práticos, foi possível perceber, através de telemetria, a atuação momentânea do nó *SeekMarker* devido aos efeitos de luminosidade não uniforme. A abordagem do nó *SeekMarker* amenizou tal problema, permitindo novas leituras do local de pouso em melhores condições durante a aterrissagem.

5.1 TRABALHOS FUTUROS

Uma abordagem para continuidade da presente dissertação é a utilização do *firmware* de aterrisagem proposto como etapas em missões de monitoramento, mapeamento ou

localização de veículos aéreos. Torna-se importante também, devido as dificuldades materiais encontradas para execução dos experimentos práticos, testes em diferentes condições ambientais. Utilização de alvos em embarcações aquáticas ou em estruturas não planas também são propostas de trabalhos complementares.

Outro estudo importante é a melhoria do algoritmo de procura do local de pouso, os testes realizados nesta dissertação consideram, a priori, que o local de pouso encontra-se visível ou próximo. A fusão sensorial visando estabelecer a localização do marcador de pouso seria uma etapa complementar útil, uma vez que atingida a região próxima do local de pouso o *firmware* de aterrisagem poderia ser executado. Essa fusão pode ser feita utilizando, inclusive, as abordagens de detecção de local de pouso por marcadores artificiais e por *features* naturais.

No âmbito dos algoritmos de pouso, uma proposta de continuidade é a comparação dos métodos de aterrissagem discutidos nesta dissertação, com outras propostas de controle, como: PD-Fuzzy, Fuzzy Tipo 2 e algoritmos baseados em aprendizagem por reforço.

Visando uma maior velocidade de atuação na aterrissagem, além da inserção do Gimbal no protótipo, uma proposta válida seria a utilização de técnicas híbridas de aterrissagem por visão, envolvendo as abordagens "*virtual landing*" e "*real landing*" (30).

REFERÊNCIAS

1 Drone Show. Anac apresenta as regras para uso de drones e aeromodelos. http://www.droneshowla.com/anac-apresenta-as-regras-para-uso-de-drones-e-aeromodelos/. Acessado em 12/01/2017.

2 ANAC. Drones. https://www.anac.gov.br/assuntos/paginas-tematicas/drones. Acessado em 12/01/2017.

3 DECEA. Drone. http://www.decea.gov.br/drone/. Acessado em 27/01/2017.

4 Predator RQ-1. http://www.airforce-technology.com/projects/predator-uav/. Acessado em 23/01/2017.

5 Helicopters and UAV. http://www.goarmy.com/about/army-vehicles-and-equipment/army-helicopters-and-uavs.html. Acessado em 23/01/2017.

6 Watch LOCUST, the Navy's prototype launcher to send drones into the sky. $https://www.washingtonpost.com/news/checkpoint/wp/2015/04/14/watch-locust-the-navys-prototype-launcher-to-send-drones-into-the-sky/?utm\_term=.e82db7893983$. Acessado em 23/01/2017.

7 Reg Austin. *Unmanned aircraft systems: UAVS design, development and deployment*, volume 54. John Wiley & Sons, 2011.

8 FAB. Esquadrão de aeronaves não tripuladas completa 4 anos. http://www.fab.mil.br/noticias/mostra/22020. Acessado em 25/02/2017.

9 FAB. VANT monitora jogo de abertura em Brasília. http://www.fab.mil.br/noticias/mostra/15313. Acessado em 25/02/2017.

10 G1. FAB compra novo drone para vigiar estádios durante a Copa do Mundo. http://g1.globo.com/brasil/noticia/2014/03/fab-compra-novo-drone-para-vigiar-estadios-durante-copa-do-mundo.html. Acessado em 25/02/2017.

11 FAB. Esquadrão Hórus participa da vigilância aérea nos Jogos Olímpicos. http://www.fab.mil.br/noticias/mostra/26951. Acessado em 25/02/2017.

12 G1. FAB e PF apreendem drogas na fronteira. http://g1.globo.com/pr/parana/noticia/2013/05/em-1-acao-conjunta-de-drones-fab-e-pf-apreendem-drogas-na-fronteira.html. Acessado em 25/02/2017.

13 Amazon. Amazon Prime Air. www.amazon.com/primeair. Acessado em 30/01/2017.

14 Amazon. Revising the Airspace Model for the Safe Integration of Small Unmanned Aircraft System.

15 Amazon. Determining Safe Access with a BestEquipped, Best-Served Model for Small Unmanned Aircraft Systems.

16 Mark Zuckerberg. The technology behind Aquila. https://www.facebook.com/notes/mark-zuckerberg/the-technology-behind-aquila/10153916136506634/. Acessado em 31/01/2017.

17 Martin Luis Gomez and Andrew Cox . Flying Aquila: Early lessons from the first full-scale test flight and the path ahead. https://code.facebook.com/posts/268598690180189. Acessado em 31/01/2017.

18 DJI. http://www.dji.com/. Acessado em 1/02/2017.

19 3DR. https://3dr.com/. Acessado em 1/02/2017.

20 Nathan Michael, Daniel Mellinger, Quentin Lindsey, and Vijay Kumar. The grasp multiple micro-UAV testbed. *IEEE Robotics & Automation Magazine*, 17(3):56–65, 2010.

21 Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *International Conference on Robotics and Automation (ICRA)*, pages 2520–2525. IEEE, 2011.

22 Yiwen Luo, Meng Joo Er, Li Ling Yong, and Chiang Ju Chien. Intelligent control and navigation of an indoor quad-copter. In *International Conference on Control Automation Robotics & Vision (ICARCV)*, pages 1700–1705. IEEE, 2014.

23 Christian Neil R Katigbak, John Raymond B Garcia, Jonn Edric D Gutang, Jonathan B De Villa, Ace Dominic C Alcid, Ryan Rhay P Vicerra, Angelo R Dela Cruz, Edison A Roxas, and Kanny Krizzy D Serrano. Autonomous trajectory tracking of a quadrotor UAV using pid controller. In *International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, pages 1–5. IEEE, 2015.

24 Shuai An, Suozhong Yuan, and Huadong Li. Self-tuning of pid controllers design by adaptive interaction for quadrotor uav. In *Guidance, Navigation and Control Conference (CGNCC), 2016 IEEE Chinese*, pages 1547–1552. IEEE, 2016.

25 Gil-Young Yoon, Akito Yamamoto, and Hun-Ok Lim. Mechanism and neural network based on pid control of quadcopter. *International Conference on Control, Automation and Systems (ICCAS 2016)*, pages 19–24, 2016.

26 Lorenz Meier, Dominik Honegger, and Marc Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *International Conference on Robotics and Automation (ICRA)*, pages 6235–6240. IEEE, 2015.

27 Lorenz Meier, Petri Tanskanen, Friedrich Fraundorfer, and Marc Pollefeys. The pixhawk open-source computer vision framework for mavs. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38(1):C22, 2011.

28 Lionel Heng, Lorenz Meier, Petri Tanskanen, Friedrich Fraundorfer, and Marc Pollefeys. Autonomous obstacle avoidance and maneuvering on a vision-guided mav using on-board processing. In *International Conference on Robotics and automation (ICRA)*, pages 2472–2477. IEEE, 2011.

29 Friedrich Fraundorfer, Lionel Heng, Dominik Honegger, Gim Hee Lee, Lorenz Meier, Petri Tanskanen, and Marc Pollefeys. Vision-based autonomous mapping and exploration using a quadrotor mav. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 4557–4564. IEEE, 2012.

30 Alvika Gautam, PB Sujit, and Srikanth Saripalli. A survey of autonomous landing techniques for uavs. In *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1210–1218. IEEE, 2014.

31 Umberto Papa and Giuseppe Del Core. Design of sonar sensor model for safe landing of an UAV. In *Metrology for Aerospace (MetroAeroSpace), 2015 IEEE*, pages 346–350. IEEE, 2015.

32 Andrea Cesetti, Emanuele Frontoni, Adriano Mancini, Primo Zingaretti, and Sauro Longhi. A vision-based guidance system for UAV navigation and safe landing using natural landmarks. In *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*, pages 233–257. Springer, 2009.

33 Yang Gui, Pengyu Guo, Hongliang Zhang, Zhihui Lei, Xiang Zhou, Jing Du, and Qifeng Yu. Airborne vision-based navigation method for UAV accuracy landing using infrared lamps. *Journal of Intelligent & Robotic Systems*, 72(2):197, 2013.

34 Khaled Abu-Jbara, Wael Alheadary, Ganesh Sundaramorthi, and Christian Claudel. A robust vision-based runway detection and tracking algorithm for automatic UAV landing. In *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1148–1157. IEEE, 2015.

35 Marci Meingast, Christopher Geyer, and Shankar Sastry. Vision based terrain recovery for landing unmanned aerial vehicles. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 2, pages 1670–1675. IEEE, 2004.

36 José Raul Azinheira and Patrick Rives. Image-based visual servoing for vanishing features and ground lines tracking: Application to a UAV automatic landing. *International Journal of Optomechatronics*, 2(3):275–295, 2008.

37 Zeng Fucen, Shi Haiqing, and Wang Hong. The object recognition and adaptive threshold selection in the vision system for landing an unmanned aerial vehicle. In *International Conference on Information and Automation. ICIA'09*, pages 117–122. IEEE, 2009.

38 Iván F Mondragón, Pascual Campoy, Carol Martinez, and Miguel A Olivares-Méndez. 3d pose estimation based on planar object tracking for UAVs control. In *International Conference on Robotics and automation (ICRA)*, pages 35–41. Ieee, 2010.

39 Miguel A Olivares-Mendez, Somasundar Kannan, and Holger Voos. Vision based fuzzy control autonomous landing with UAVs: From v-rep to real experiments. In *Control and Automation (MED), 2015 23th Mediterranean Conference on*, pages 14–21. IEEE, 2015.

40 Patrick Benavidez, Josue Lambert, Aldo Jaimes, and Mo Jamshidi. Landing of an ardrone 2.0 quadcopter on a mobile base using fuzzy logic. In *World Automation Congress (WAC), 2014*, pages 803–812. IEEE, 2014.

41 Hanseob Lee, Seokwoo Jung, and David Hyunchul Shim. Vision-based UAV landing on the moving vehicle. In *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1–7. IEEE, 2016.

42 Francesco Cocchioni, Valerio Pierfelice, Alessandro Benini, Adriano Mancini, Emanuele Frontoni, Primo Zingaretti, Gianluca Ippoliti, and Sauro Longhi. Unmanned ground and aerial vehicles in extended range indoor and outdoor missions. In *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 374–382. IEEE, 2014.

43 Shuo Yang, Jiahang Ying, Yang Lu, and Zexiang Li. Precise quadrotor autonomous landing with srukf vision perception. In *International Conference on Robotics and Automation (ICRA)*, pages 2196–2201. IEEE, 2015.

44 JeongWoon Kim, YeonDeuk Jung, DaSol Lee, and David Hyunchul Shim. Landing control on a mobile platform for multi-copters using an omnidirectional image sensor. *Journal of Intelligent & Robotic Systems*, 84(1-4):529–541, 2016.

45 Daewon Lee, Tyler Ryan, and H Jin Kim. Autonomous landing of a vtol UAV on a moving platform using image-based visual servoing. In *International Conference on Robotics and Automation (ICRA)*, pages 971–976. IEEE, 2012.

46 Witold Pedrycz and Fernando Gomide. *Fuzzy systems engineering: toward human-centric computing*. John Wiley & Sons, 2007.

47 Serge Boverie, Bernard Demaya, and Andre Titli. Fuzzy logic control compared with other automatic control approaches. In *Decision and Control, 1991., Proceedings of the 30th IEEE Conference on*, pages 1212–1216. IEEE, 1991.

48 P Karuppanan and Kamala Kanta Mahapatra. Pi and fuzzy logic controllers for shunt active power filter—a report. *ISA transactions*, 51(1):163–169, 2012.

49 Radu-Emil Precup and Hans Hellendoorn. A survey on industrial applications of fuzzy control. *Computers in Industry*, 62(3):213–226, 2011.

50 João Pedro Carvalho, Marco Jucá, Alexandre Menezes, Leonardo Olivi, Alexandre Bessa, and André Marcato. *Lecture Notes in Electrical Engineering. 402ed*, chapter Autonomous UAV Outdoor Flight Controlled by an Embedded System Using Odroid and ROS, pages 423–437. Springer International Publishing, 2017.

51 Lucas Moraes, Luiz Carmo, Rafael Falci, João Pedro Carvalho, Marco Jucá, Alexandre Menezes, Daniel Discini, Thiago Coelho, Alexandre Bessa, and André Marcato. Autonomous quadrotor for accurate positioning. *IEEE Aerospace and Electronic Systems Magazine*, 2017.

52 João Pedro Carvalho, Marco Jucá, Alexandre Menezes, Leonardo Olivi, Alexandre Bessa, and André Marcato. Landing an UAV in a dynamical target using fuzzy control and computer vision. In *Congresso Brasileiro de Automática (CBA), 2016*, 2016.

53 Filipe Delgado, João Pedro Carvalho, A. Bessa, and Thiago Coelho. An optical fiber sensor and its application in UAVs for current measurements. *Sensors (Basel)*, 2016.

54 Moraes L., Campos. R., Carmo L., Moreira L., Carvalho J.P, Teixeira A., Discini. D., Coelho T., Marcato A., and Santos A. Desenvolvimento de um VANT híbrido autônomo para inspeção multiespectral em áreas no entorno de reservatórios hidrelétricos. In *IX Congresso de Inovação Tecnológica em Energia Elétrica - IX CITENEL*, 2016.

55 Ítalo Alvarenga, Filipe Delgado, João Pedro Carvalho, A. Bessa, and Thiago Coelho. Development of fiber-optic current sensor based on long-period fiber grating for uav applications. In *MOMAG 2016*, 2016.

56 About ROS. http://www.ros.org/about-ros/. Acessado em 12/08/2015.

57 ROS History. http://www.ros.org/history/. Acessado em 12/08/2015.

58 ROS robot operation system. http://http://wiki.ros.org/. Acessado em 12/08/2015.

59 Aaron Martinez and Enrique Fernández. *Learning ROS for Robotics Programming*. Packt Publishing Ltd, 2013.

60 R Patrick Goebel and ROS By Example-Volume. *A Do-It-Yourself Guide to the Robot Operating System*. 2012.

61 ROS Distributions. http://wiki.ros.org/Distributions. Acessado em 12/08/2015.

62 Catkin or Rosbuild. http://wiki.ros.org/catkin_or_rosbuild. Acessado em 12/08/2015.

63 Ar Track Alvar. http://wiki.ros.org/ar_track_alvar. Acessado em 12/08/2015.

64 Stereo image proc. http://wiki.ros.org/stereo_image_proc. Acessado em 12/08/2015.

65 MAVROS. http://wiki.ros.org/mavros. Acessado em 12/08/2015.

66 ROSARIA. http://wiki.ros.org/ROSARIA. Acessado em 12/08/2015.

67 Catkin Workspaces. http://wiki.ros.org/catkin/workspaces. Acessado em 12/08/2015.

68 OMG. Common object request broker architecture. http://www.omg.org/gettingstarted/corbafaq.htm. Acessado em 12/08/2015.

69 Peter Corke. *Robotics, vision and control: fundamental algorithms in MATLAB*, volume 73. Springer, 2011.

70 Qgroundcontrol. Mavlink micro air vehicle communication protocol. http://qgroundcontrol.org/mavlink/start. Acessado em 17/08/2015.

71 ESAcademy. Can bus. http://www.canbus.us/. Acessado em 1708/2015.

72 PX4 Autopilot. PX4 Development Guide. https://dev.px4.io/. Acessado em 08/03/2017.

73 Vladimir Ermakov. MAVROS - MAVlink to ROS gateway with UDP proxy for Ground Control Station. https://github.com/mavlink/mavros. Acessado em 31/08/2015.

74 Lorenzo Sciavicco and Bruno Siciliano. *Modelling and control of robot manipulators*. Springer Science & Business Media, 2012.

75 Witold Pedrycz and Fernando Gomide. *Fuzzy systems engineering: toward human-centric computing*. John Wiley & Sons, 2007.

76 Laécio Carvalho de Barros and Rodney Carlos Bassanezi. *Tópicos de lógica fuzzy e biomatemática*. Grupo de Biomatemática, Instituto de Matemática, Estatística e Computação Científica (IMECC), Universidade Estadual de Campinas (UNICAMP), 2006.

77 Juan Rada-Vilela. fuzzylite: a fuzzy logic control library, 2017.

78 Simon Haykin and Neural Network. A comprehensive foundation. *Neural Networks*, 2(2004):41, 2004.

79 Teresa Bernarda Ludermir, Antônio P Braga, and ACPLF Carvalho. Redes neurais artificiais: teoria e aplicações. *Livros Técnicos e Científicos Editora*, 2000.

80 Martin T Hagan and Mohammad B Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE transactions on Neural Networks*, 5(6):989–993, 1994.

81 Hongwei Liu. On the levenberg-marquardt training method for feed-forward neural networks. In *Sixth International Conference on Natural Computation (ICNC)*, volume 1, pages 456–460. IEEE, 2010.

82 IN da Silva, Danilo Hernane Spatti, and Rogério Andrade Flauzino. Redes neurais artificiais para engenharia e ciências aplicadas. *São Paulo: Artliber*, pages 33–111, 2010.

83 Pixhawk.org. Pixhawk autopilot. https://pixhawk.org/modules/pixhawk, 2015. Acessado em 09/09/2015.

84 Odroid Wiki. Odroid. http://odroid.com/dokuwiki/doku.php, 2015. Acessado em 18/09/2015.

85 Computer Vision and Geometry Lab. About Pixhawk. https://pixhawk.ethz.ch/overview, 2015. Acessado em 17/04/2017.

86 Christopher NEGUS. Linux: a bíblia. *Starlin Alta Consult. ISBN: 8576081792*, 2008.

87 Ping-Hao Chang, Kuan-Hung Kuo, Der-Yu Tsai, Kevin Chen, and Luba Tang. Skypat: C++ performance analysis and testing framework. In *Linux Symposium*, page 97. Citeseer.

88 J. Pinheiro, S. Cunha, G. Gomes, and S. Carvajal. *Probabilidade e Estatística: Quantificando a Incerteza*. Elsevier Brasil, 2013.

89 D.S. Moore. *The Basic Practice of Statistics*. The Basic Practice of Statistics. W.H. Freeman, 2007.