Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia CCET
Programa de Pós-Graduação em Engenharia Elétrica PPGEE

Mateus Gava Francisco

Ambiente de desenvolvimento de drones com ROS2, Linux Embarcado e Matlab

São Carlos - SP
2026

Mateus Gava Francisco

Ambiente de desenvolvimento de drones com ROS2, Linux Embarcado e Matlab

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração: Processamento Digital de Sinais

Orientador: Prof. Dr. André Carmona Hernandes

São Carlos - SP
2026

UFSCar

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia CCET

Programa de Pós-Graduação em Engenharia Elétrica PPGEE

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Mateus Gava Francisco, realizada em 25/05/2026.

Comissão Julgadora:

Prof. Dr. André Carmona Hernandes (UFSCar)

Prof. Dr. Jones Yudi Mori Alves da Silva (UnB)

Prof. Dr. Roberto Santos Inoue (UFSCar)

O Relatório de Defesa produzido pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação.

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Agradeço Primeiramente a Deus, que não me abandonou e me deu forças para continuar. Ao meu orientador Prof. Dr. André Carmona Hernandes, que guiou-me por este caminho, até então desconhecido. Ao Prof. Dr. Jones Yudi Mori Alves da Silva, por ter cedido a Kria KV260 para realização deste trabalho além de valiosos conheciemntos para a utilização da mesma.

Também agradesco a todos os familiares e amigos que, de alguma forma, colaboraram com este trabalho. Em especial, meus pais Sonia e Wanderley por todo o apoio e amparo emocional, a Carla e Pedro, amigos de trabalho que se extenderam para a vida e colaboraram com olhar tecnico sobre o texto oferendo uma valiosa contribuição

Destino um agradecimento especial ao grande amigo Anderson, que sempre foi fonte de inspiração e teve sua vida ceifada prematuramente.

"Somos essencialmente profissionais do sentido. Educamos, quando ensinamos com sentido. Educar é impregnar de sentido a vida. A profissão docente está centrada na vida, no bem querer."
(Prof. Gilberto Teixeira)

Resumo

Há tempos a humanidade sonha com robos desempenhando as mais diversas funçoes, nos dias atuais isso esta mais proximo da realidade do que se imagina. Diante da grande explosão e propagação no uso de robos, e o advento dos drones, construir e projetar essas maquinas têm se tornado cada vez mais desafiador. Com o intuito de remediar essa situação diversas tecnologias surgiram, como por exemplo Middwares, hardwares poderosos e compactos, além de tecnicas de desenvolvimento e testes que aceleraram e barateram a concepção de drones. Contudo testar de forma rápida e segura essas aeronaves durante seu desenvolvimento tem se tornado um desafio. Para mitigar esse problema, o presente trabalho busca a idealização de um ambiente capaz de auxiliar o desenvolvimento de drones, utilizando tecnicas de software e hardware in the loop, alinhados ao uso de sistemas embarcados, a simulações e a um modelo fisico de um dos eixos de um drone. Os resultados preliminares mostram que o Linux embarcado é capaz de controlar a simulação de forma estável e com valores de latencia toleráveis.

Palavras-chave: Middleware. Hardware. Testes. Drone. .

Abstract

Humanity has long dreamed of robots performing a wide variety of functions; nowadays, this is closer to reality than one might imagine. Given the explosion and spread of robot use, and the advent of drones, building and designing these machines has become increasingly challenging. To remedy this situation, various technologies have emerged, such as powerful and compact middleware and hardware, as well as development and testing techniques that have accelerated and reduced the cost of drone design. However, testing these aircraft quickly and safely during their development has become a challenge. To mitigate this problem, this work seeks to conceptualize an environment capable of assisting drone development, using software and hardware-in-the-loop techniques, aligned with the use of embedded systems, simulations, and a physical model of one of the drone's axes. Preliminary results show that embedded Linux is capable of controlling the simulation stably and with tolerable latency values.

Keywords: Middleware. Hardware. Tests. Drone. .

Lista de ilustrações

Lista de tabelas

Lista de siglas

ANAC Agencia nacional de Aviação Civil

ARM Advanced RISC Machine

AMD Advanced Micro Devices

API Interface de Programação de Aplicações

CAGED Cadastro Geral de Empregados e Desempregados

CBO Classificação Brasileira de Ocupações

CBT Companhia Brasileira de Tratores

CPU Unidade Central de Processamento

DARPA Defense Advanced Research Projects Agency

DDS Data Distribution Service

EOL End of Life

ESC Controlador Eletrônico de Velocidade

FPGA Field Programmable Gate Array

GPU Unidade de Processamento Gráfico

HIL Hardware in the loop

IA Inteligência Artificial

IFR International Federation of Robotics

I/O Entrada e Saída

MPSoC Multi-Processor System-on-Chip

PID Proporcional-Integral-Derivativo

QoS Qualidade de Serviço

RBAC-E Regulamento Brasileiro da Aviação Civil Especial

ROS Robot Operating System

ROS2 Robot Operating System 2

RTOS Real Time Operational System

RTT Round Trip Time

RMW ROS Middleware

SISANT Sistema de Aeronaves não Tripuladas

SIL Software in the loop

SOM System on Module

SOC System on Chip

TCP Transmission Control Protocol

UAV Unmanned Aerial Vehicle

UDP User Datagram Protocol

USB Universal Serial Bus

UART Universal Asynchronous Receiver/Transmitter

VANT Veículo Aéreo Não Tripulado

Sumário

Capítulo 1

Introdução

Desde os primórdios da humanidade, os seres humanos exploram ferramentas e técnicas que facilitem a execução de suas atividades, principalmente as repetitivas ou que demandam grande força física. É do imaginário coletivo a concepção de um equipamento capaz de realizar, de maneira autônoma, as tarefas constantemente repetidas e monótonas, além de trabalhos pesados. Este equipamento tão encantador para as gerações passadas que tem se tornado real em nosso presente, são os robôs.

O conceito de robô surgiu de uma peça de teatro de 1920 chamada "A Fábrica de Robôs", criada pelo dramaturgista tcheco Karel Čapek. A história se centrava em uma fábrica que fabricava seres artificiais muito semelhantes aos humanos (TCHÁPEK; MACHAC; JOVANOVIC, 2022). O termo "robô" foi cunhado pelo seu irmão, Josef Čapek, a partir da palavra tcheca "robota", que significa "servidão".

O termo "robótica", por outro lado, foi cunhado pelo escritor Isaac Asimov, um dos clássicos autores de ficção científica, conhecido pela coletânea de histórias curtas "Eu, Robô" e pelas séries de histórias "Robôs" e "Fundação", além da criação das clássicas Três Leis da Robótica(ASIMOV, 2015).

Embora robôs e outros tipos de seres artificiais fossem parte de nossa literatura e até das antigas mitologias (onde eram conhecidos como "autômatos"), a robótica como ciência e tecnologia ainda demorou um tempo para se desenvolver plenamente.

O século XX trouxe avanços cruciais para a robótica com a introdução da eletrônica e dos computadores. A teoria da cibernética, proposta por Norbert Wiener na década de 1940, explorava a comunicação e o controle em animais e máquinas, estabelecendo as bases teóricas para a robótica moderna (WIENER; HILL; MITTER, 2019).

A primeira geração de robos industriais nasceu em meados de 1950 marcando um ponto de virada na história da robótica. O desenvolvimento de máquinas automáticas

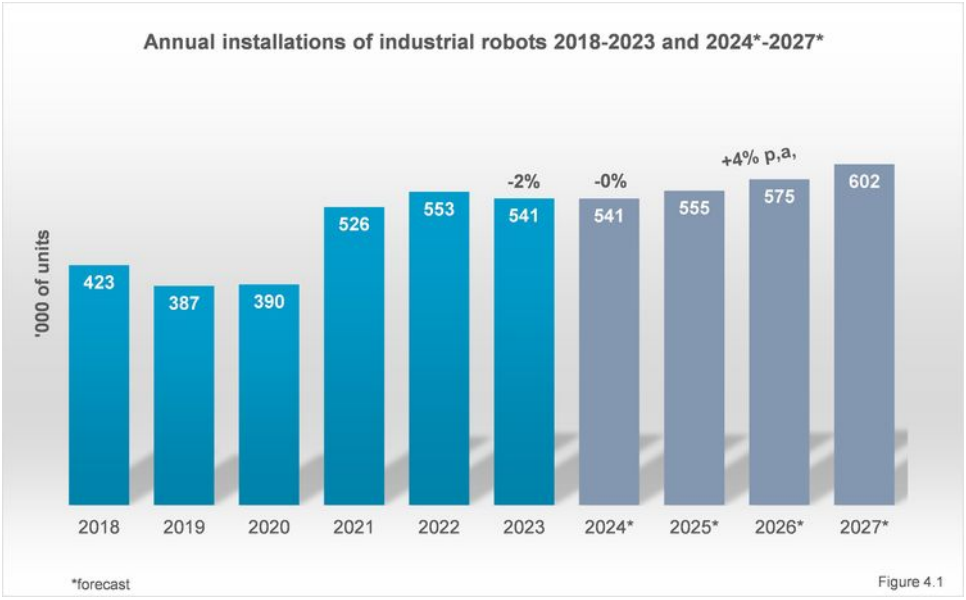
para fabricação impulsionou a necessidade de maior automação. A partir disso, varias empresas de fabricação de robos nasceram, fomentadas pelo grande potencial de aplicação, principalmete do setor automotivo. (GASPARETTO, 2019)

George Devol, em 1954, inventou o primeiro robô programável, chamado Unimate. Este robô foi posteriormente utilizado em linhas de montagem industriais, revolucionando a fabricação (MALONE, 2011). Na década de 1960, a robótica começou a se expandir rapidamente, com empresas como a General Motors implementando robôs industriais em suas fábricas.

Desde então a indústria vem experimentando o fenômeno da robotização, onde atividades designadas para forças de trabalho humana, vem sendo gradualmente substituídas por robôs.(MARR, 2017)

Esse fenômeno pode ser observado no estudo conduzido pela International Federation of Robotics (IFR) (IRF, 2026), que com base nas tendencias tecnológicas e de mercado, apresenta um gráfico com as métricas e previsão de instalação de novos robôs industriais conforme a Figura 1.

Figura 1 – Instalação anual de robos industriais



Fonte: (IRF, 2026)

No que tange o segmento aeronáutico, a robótica começa a ganhar os céus em meados de 1970. Abraham Karem, também conhecido como o pai da tecnologia Unmanned Aerial Vehicle (UAV), construiu o Albatross utilizando fibra de vidro e madeira. Posteriormente, sob financiamento do Defense Advanced Research Projects Agency (DARPA), recebeu melhorias e passou a ser chamado de Amber. (GOMES et al., 2022). Contudo os UAVs ainda eram de uso militar e os civis não tinham acesso à essa tecnologia.

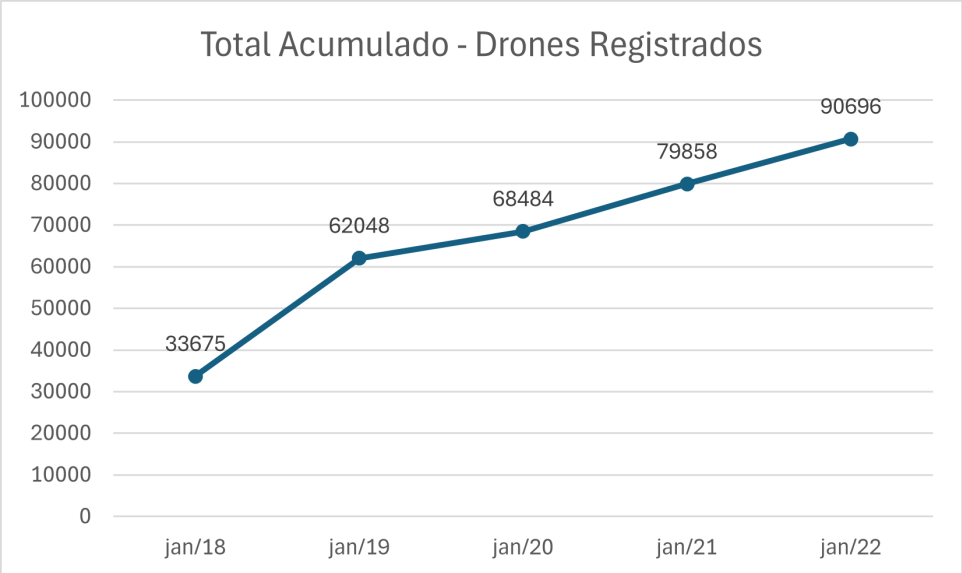
No Brasil, o primeiro registro de um drone foi do BQM1BR, fabricado pela Companhia Brasileira de Tratores (CBT) , que realizou seu primeiro voo em 1983 (MUNARETTO, 2015).

A partir da década de 1980, a integração de Inteligência Artificial (IA), aprendizado de máquina e visão computacional abriram novas fronteiras. UAVs autônomos surgiram, capazes de realizar tarefas complexas com mínima intervenção humana, abrindo portas para um futuro repleto de possibilidades.

Hoje, os UAVs estão em diversos setores, incluindo agricultura, logística e entretenimento. Avanços em IA e aprendizado de máquina têm permitido que esses dispositivos realizem tarefas antes inimagináveis, desde diagnósticos em lavouras até projeções de imagens no céu.

Diante de todos esses avanços a demanda por essas aeronaves é cada vez maior. A Agencia nacional de Aviação Civil (ANAC), por meio do Sistema de Aeronaves não Tripuladas (SISANT), realiza o registro dos drones para uso civil, seja recreativo ou profissional. Esse registro é regido pelo Regulamento Brasileiro da Aviação Civil Especial (RBAC-E) de número 94, que trata dos dos requisitos geraris para aeronaves não tripuladas de uso civil. A RBAC-E n° 94, em sua subparte D, item E 94.301, Parágrafos D e E, determina que: "As aeronaves não tripuladas de peso máximo de decolagem de até 250 gramas não precisam ser cadastradas junto à ANAC ou identificadas e o cadastro das aeronaves é válido por 24 meses. O cadastro não revalidado até 6 meses depois de vencido será inativado e não poderá mais ser revalidado". No período compreendido de 2018 a 2022 houve um crescimento na quantidade de registros, conforme o gráfico da Figura 2.

Figura 2 – Total Acumulado - Registros de Drones entre 2018 e 2022



Fonte: (ANAC, 2024)

Há um crescimento expressivo entre 2018 e 2019, e após esse período um crescimento mais brando,porém constante. Isso se deve ao fato dos drones registrados em 2018 e que não tiveram o cadastro revalidado pararam de integrar o total acumulado, uma vez que não estão aptos legalmente para voos.

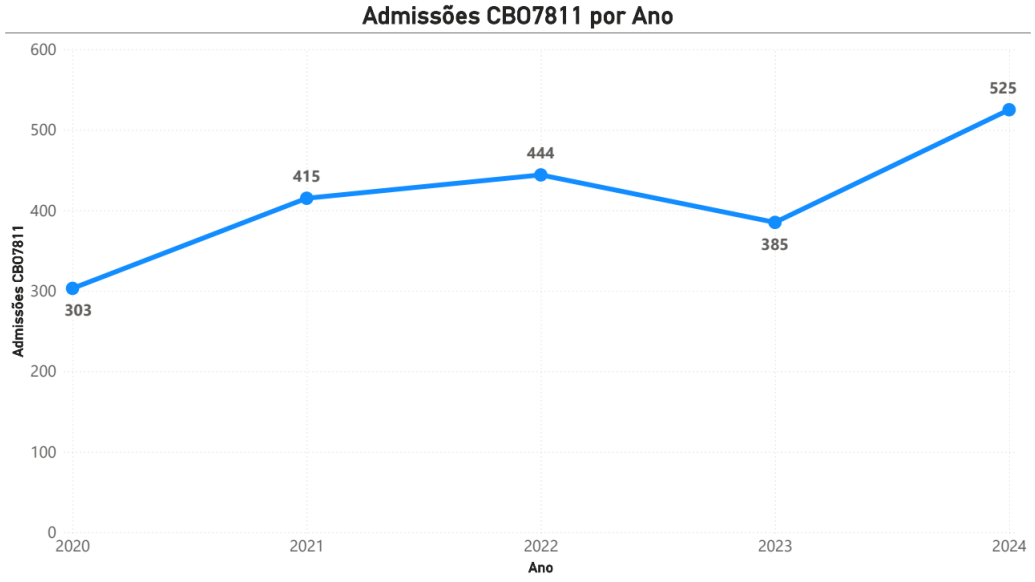
A partir de 2022 o acesso aos dados abertos da ANAC foi alterado, e as consultas não fornecem mais os totais acumulados anuais, agora o sistema fornece os cadastros validos de drones até a data da consulta.

Essa crescente demanda, tanto por robos industriais, quanto por drones, acaba aquecendo o mercado e impulsionando o desenvolvimento de equipamentos mais tecnológicos e sofisticados o que, consequentemente, aumenta a demanda de profissionais capazes de reparar e operar essas máquinas.

No Brasil, a identificação das ocupações no mercado de trabalho se dá pelo Classificação Brasileira de Ocupações (CBO), que prove uma classificação enumerativa e uma classificação descritiva.

Tomando como base os dados provenientes do Cadastro Geral de Empregados e Desempregados (CAGED), pode-se observar um aumento nas admissões para o CBO 7811 que representa o operador de robos industriais, profissional esses que, de acordo com a classificação descritiva, desempenham, dentre outras, as seguintes atividades: preparam, programam e realizam manutenção de rotina em robôs.(MTE, 2024) Esses aumentos podem ser observados respectivamente na Figura 3.

Figura 3 – Admissões CBO 7811



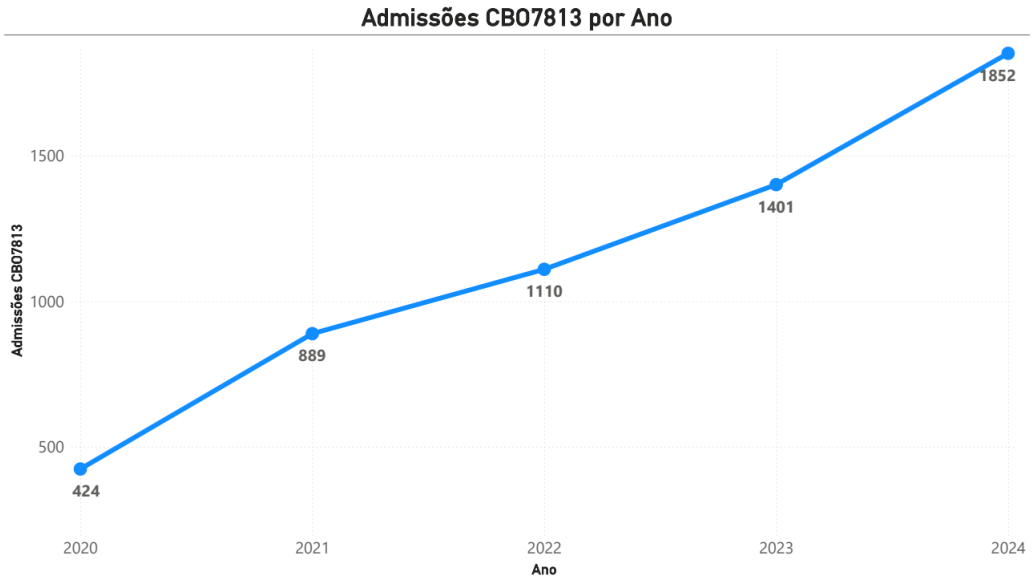
Fonte: MTE/SPPE/DES/CGET - CAGED LEI 4.923/65

De forma análoga, pode-se observar um aumento nas admissões para o CBO 7813

que representa o operador de equipamentos pilotados remotamente, profissional esses que, de acordo com a classificação descritiva, desempenham, dentre outras, as seguintes atividades: operam, preparam, programam e realizam manutenção de rotina em sistema rov e aeronaves não tripuladas.(MTE, 2024)

Esses aumentos podem ser observados respectivamente na Figura 4.

Figura 4 – Admissões CBO 7813



Fonte: MTE/SPPE/DES/CGET - CAGED LEI 4.923/65

Com base no aumento da procura, tanto por esses equipamentos quanto pelos profissionais que projetam, operam e realizam manutenção nos mesmo, instigar as pessoas a aprender sobre esse universo faz parte da solução.

Para isso pode- se adotar algumas técnicas, como por exemplo, o uso de uma bancada experimental que apresenta conceitos como a aero estabilização, alinhado ao uso de Middlewares como o ROS, plataformas de hardware flexíveis como o FPGA integrado a sistemas embarcados, simulações complementares as aplicações no mundo real por meio de softwares como o MatLab e de arquiteturas como a do Hardware in the loop.

Diante disso o presente trabalho pretende explorar soluções podem aprimorar o desenvolvimento de drones reduzindo o tempo e custo, por meio da adoção de Middlewares, praticas de simulação e testes envolvendo técnicas de Hardware in the loop (HIL).Essa abordagem busca promover um ambiente mais amistoso para o treinamento e capacitação na operação e manutenção de drones.

Capítulo 2

Objetivos

A crescente demanda pelos equipamentos e profissionais, evidenciado no capítulo anterior, abre a possibilidade para desenvolvimento de uma solução capaz de auxiliar a preencher essa lacuna.

Diante disso, alguns desafios vêm à tona, como por exemplo, a confiabilidade temporal e de movimentação para as simulações e para a bancada propostas, convergindo em um ambiente confiável e próximo a realidade. Tais desafios promovem questionamentos como: "Como aferir a confiabilidade temporal?"

A construção desse ambiente demanda recursos físicos, como a própria bancada experimental, e recursos virtuais, dentre os quais duas ferramentas se destacam: o ROS, por ser amplamente utilizado em diversas aplicações como robôs móveis, incluindo drones, e o MATLAB como uma ferramenta de simulação versátil e flexível, capaz de interagir com o ROS. Contudo, ROS alinhado ao MATLAB é uma boa escolha para o desenvolvimento e simulação de drones?

A elaboração do ambiente proposto limitado exclusivamente a computadores pode limitar a flexibilidade, adotar o uso de um SOM com Linux embarcado é uma opção viável e provê a flexibilidade desejada, porém levanta questionamentos como: "um SOM com Linux embarcado possui velocidade e confiabilidade suficientes para ser utilizado como componente chave em uma arquitetura HIL, executando ROS e comunicando com um simulador?".

Utilizando um SOM com Linux embarcado alinhado ao ROS, as taxas de comunicação são constantes, sem atrasos significativos a ponto de podermos utilizar em uma simulação as taxas de comunicação utilizadas em drones reais?

As respostas para esses questionamentos podem trazer ganho de tempo e redução de custos no desenvolvimento de novos drones e em testes, que aplicadas as indústrias da

região de São Carlos podem elevar a competitividade de tais empresas no cenário nacional. Além disso a bancada aero estabilizadora pode ser utilizada como recurso didático para o ensino das características intrínsecas dos drones, promovendo o interesse da comunidade em uma área em plena expansão.

Este trabalho se propõe a investigar se, com o uso de um SOM com Linux embarcado e ROS, podemos aferir a confiabilidade temporal e obter taxas de comunicação constantes e sem atrasos significativos.

O mesmo está organizado da seguinte forma: O terceiro capítulo apresenta uma revisão bibliográfica, com trabalhos correlatos e que apresentam o uso de ferramentas e técnicas que serão utilizadas no decorrer deste trabalho. O quarto capítulo apresenta uma coletânea dos materiais e métodos para a execução do trabalho. O quinto capítulo apresenta os resultados obtidos. E o sexto e último capítulo apresenta as conclusões e trabalhos futuros.

Capítulo 3

Revisão Bibliográfica

3.1 Desenvolvimento de Drones

Um drone é composto por diferentes elementos, sejam eles físicos como: estrutura, hélices, motores, sensores, etc. sejam eles lógicos, como softwares de controle, planejamento de voo dentre outros.

O trabalho desenvolvido em (OETTERSHAGEN et al., 2017), apresenta um drone de propulsão solar que pode ser lançado com as mãos. É possível apreciar no projeto a forma como o drone capta e armazena a energia, como os parâmetros de voo são definidos além do uso do hardware in the loop alinhado ao simulador X-Plane para teste da estrutura de controle e posterior refino com testes de voo.

Em (ZHAO et al., 2020) , também é apresentado o desenvolvimento de um drone de propulsão solar que pode ser lançado com as mãos. Diferindo do anterior este trabalho tem como foco o desenvolvimento para uso em condições de baixas temperaturas e pressão.

Os autores em (CHODNICKI et al., 2022), apresentam uma solução energeticamente eficiente para controlar aeronaves não tripuladas, utilizando ROS alinhado a um Real Time Operational System (RTOS), além disso foram utilizadas técnicas de hardware in the loop e um computador com processador baseado em Advanced RISC Machine (ARM).

Em (KöVARI; EBEID, 2021a), os autores propõem uma plataforma baseada em FPGA para operação de drones autônomos em tempo real. Para tal, foi utilizado pelos autores um Multi-Processor System-on-Chip (MPSoC), que combina FPGA, Unidade Central de Processamento (CPU) e Unidade de Processamento Gráfico (GPU), onde o FPGA é encarregado dos algoritmos de IA, e o CPU para executar o ROS, que processa a comunicação entre o controlador de voo e os sensores embarcados no drone. A plataforma desenvolvida foi testada tanto em ambiente simulado com o simulador Gazebo, quanto no

mundo real.

O trabalho desenvolvido em (NYBOE; MALLE; EBEID, 2022), apresenta um framework para a integração entre ROS2, PX4 e FPGA. Similar ao trabalho anterior, este utiliza um MPSoC, com Ubuntu e ROS2 sendo executado pela CPU, com a diferença na integração do piloto automático Open Soure PX4.

Em (PODLUBNE; GöHRINGER, 2019), os autores apresentam uma metodologia para uma arquitetura que permite a integração entre ROS e FPGA, de forma a obter uma arquitetura modular para aplicações reconfiguráveis compatíveis com ROS, módulos plug & play dentre outros tópicos.

Os autores em (WAN et al., 2022), expõe que a computação robótica em FPGAs emerge como uma solução crítica para superar o gargalo de von Neumann e a alta latência associados a CPUs e GPUs, permitindo o processamento paralelo massivo e determinístico necessário para tarefas de percepção e controle em tempo real. Contudo, a pesquisa nesta área enfrenta o desafio da lacuna de programabilidade: a dificuldade de traduzir algoritmos robóticos complexos, tipicamente escritos em C++ ou Python no ecossistema ROS, para linguagens de descrição de hardware.

O trabalho desenvolvido em (MAYORAL-VILCHES et al., 2022), explora uma arquitetura aberta para aceleração de hardware no ROS2, que por meio de testes de benchmark, mostrou que houve um ganho de velocidade de cerca de 24 % do FPGA sobre um CPU.

Em (MEGALINGAM et al., 2022), os autores apresentam uma simulação da estabilidade de um drone, utilizando ROS e o simulador Gazebo, através de um controlador Proporcional-Integral-Derivativo (PID) o drone é estabilizado para atingir uma coordenada de destino, com o Gazebo atuando como simulador 3D e o ROS como interface para o drone.

Os autores em (JAIN; GUPTA; MATHUR, 2021), apresentam a concepção de um drone autônomo baseado em ROS, que tem como objetivo principal vigilância e mapeamento 3D baseado em satélite.

De forma geral, a vertente de desenvolvimento de drones pode atingir diferentes segmentos, mas compartilham algumas característica, como por exemplo: Uso do ROS ou ROS2, uso de simuladores como o X-plane ou o Gazebo e crescente interesse no uso de FPGA e tempo real.

Em síntese, o panorama apresentado neste tópico demonstra que a convergência entre o ecossistema ROS2, a aceleração de hardware via FPGA e as metodologias de simulação HIL representam o estado da arte no desenvolvimento de drones, alinhando-se diretamente aos objetivos deste trabalho.

3.2 ROS

Um robô é uma máquina equipada com sensores, atuadores (motores) e uma unidade de processamento que opera com base em comandos do usuário ou de maneira autônoma, de acordo com os dados fornecidos pelos sensores. A unidade de processamento pode ser considerada o "cérebro"do robô, sendo representada por um microcontrolador ou um computador. As decisões e ações do robô são inteiramente determinadas pelo programa que está em execução nessa unidade de processamento (JOSEPH; JOHNY, 2022).

A programação de robôs é um subgrupo da programação computacional e consiste no desenvolvimento do código que permite ao robô criar inteligência para tomada de decisões, seja pela implementação de um controlador, automação de tarefas repetidas dentre outras frentes. Essa programação pode ser realizada com qualquer linguagem de programação, porém Python e C++ são as mais comuns e amplamente utilizadas, principalmente devido a fatores como performance e bom suporte da comunidade (JOSEPH; JOHNY, 2022).

Dentro desse contexto encontra-se o ROS, que é um framework para o software de robôs, gratuito e de código aberto (OPEN-ROBOTICS, 2024b).

O ROS teve sua origem com dois estudantes da faculdade de Stanford: Eric Berger e Keenan Wyrobek, que por volta de 2005 buscavam uma plataforma para implementar e testar sua pesquisa: o projeto de um robô pessoal intricadamente seguro (WYROBEK et al., 2008). Essa busca revelou que cerca de 90% do trabalho de desenvolvimento de um robô é gasto com prototipagem e rescrita de código. Isso impulsionou a criação do ROS, que nos dias de hoje possui uma comunidade poderosa e impossível de se evitar ao trabalhar com robôs (WYROBEK, 2017).

O ROS consolidou-se na última década como o padrão de fato para o desenvolvimento de software em robótica. Apesar do nome, o ROS não é um sistema operacional no sentido tradicional (como Linux ou Windows), mas sim um middleware que fornece serviços como abstração de hardware, controle de dispositivos de baixo nível, passagem de mensagens entre processos e gerenciamento de pacotes (QUIGLEY et al., 2009).

3.2.1 ROS vs. ROS2: Arquitetura e Middleware

Uma das escolhas projetuais fundamentais deste trabalho é a utilização do ROS2. Para justificar essa decisão, é necessário compreender as limitações da primeira versão (ROS) frente aos requisitos de sistemas modernos de Veículo Aéreo Não Tripulado (VANT)

O ROS baseia-se em uma arquitetura centralizada que depende de um Roscore (nó mestre) para descoberta e estabelecimento de conexões. A comunicação utiliza majoritariamente protocolos Transmission Control Protocol (TCP) e User Datagram Protocol (UDP) com serialização customizada (TCPROS/UDPROS). Embora eficiente para pesquisa acadêmica em ambientes controlados, essa arquitetura apresenta gargalos: ponto

único de falha (se o Roscore cai, todo o sistema para) e falta de suporte nativo a tempo real (MARUYAMA; KATO; AZUMI, 2016).

O ROS2 é baseado em Data Distribution Service (DDS), um padrão aberto utilizado em infraestruturas críticas, como militar, aeroespacial e sistemas financeiros (PARDO-CASTELLOTE, 2003). Isso proporciona ao ROS2 a capacidade de trabalhar com sistemas de tempo real, comunicação multi robos, operação em ambientes de rede não ideias além do aprimoramento da segurança (MACENSKI et al., 2022a).

A influência do sistema operacional na previsibilidade temporal do ROS2 foi analisada por (SARKAR et al., 2021), utilizando um sistema de pêndulo invertido como caso de uso. Os resultados experimentais demonstraram uma distinção clara entre o uso de um Sistema Operacional de Propósito Geral e um Sistema Operacional de Tempo Real. Enquanto a execução sobre o kernel Linux padrão apresentou picos elevados na média e no desvio padrão do jitter sob alta demanda, resultando em instabilidade temporal , a implementação com o framework Xenomai exibiu um comportamento significativamente mais estável, mitigando a ocorrência de jitters excessivos que poderiam levar à perda de prazos críticos. Estes dados corroboram a necessidade de extensões de tempo real para garantir a segurança em aplicações robóticas críticas baseadas em ROS2.

Em (COSTA et al., 2017), os autores propõem a integração entre o Ptolemy, que modela, projeta e simula sistemas embarcados concorrentes e de tempo real, e o ROS para a criação de um ambiente multi robos.

As principais vantagens do ROS2 para este projeto incluem:

- ❏ Descentralização: Não existe Roscore. A descoberta de nós é feita automaticamente via DDS, aumentando a robustez do sistema.

- ❏ Qualidade de Serviço (QoS): O ROS2 permite configurar como os dados são transportados (ex: Best Effort vs. Reliable). Isso é crítico para telemetria de drones, onde perder um pacote de vídeo é aceitável, mas perder um comando de controle não é.

- ❏ Suporte a Tempo Real: A arquitetura foi desenhada para ser compatível com RTOS e garantir determinismo na entrega de mensagens, fator essencial para a estabilidade de malhas de controle de voo (MACENSKI et al., 2022b).

Dessa forma, a exploração do ROS e, especificamente, a adoção do ROS2, fundamentam os objetivos de versatilidade e confiabilidade temporal estabelecidos neste trabalho. A migração para o padrão DDS abordada neste tópico é o que permite ao sistema proposto gerir a comunicação entre a FPGA e os simuladores virtuais com o determinismo necessário para simulações HIL. Assim, o ROS2 deixa de ser apenas uma escolha de software para se tornar a ferramenta que viabiliza a criação de um ambiente de desen-

volvimento modular, capaz de integrar hardware real e modelos matemáticos de forma segura e eficiente.

3.3 Simuladores

A construção de qualquer equipamento no mundo real demanda tempo e dinheiro, produzir um protótipo para testes e calibragem muitas vezes se torna pouco viável, por isso realizar simulações computacionais possibilitam minimizar os custos e o tempo desprendido no desenvolvimento.

Uma simulação pode ser descrita como um modelo matemático de algo do mundo real, que pode ser reproduzida em ambiente virtual, e para isso alguns softwares podem ser utilizados.

Pode-se utilizar diferentes softwares para simulações dos mais diversos propósitos. Em (RODIć; MESTER, 2011), são utilizados os softwares Matlab/Simulink para modelar e simular um drone, obtendo informações acerca do comportamento e funcionamento da aeronave.

Além das simulações matemáticas, também são comumente utilizados simuladores para visualização e interação em ambientes 3D.

O trabalho desenvolvido em (GOEL et al., 2009), apresenta a modelagem e simulação de um drone com o simulink para a parte matemática e o simulador de voo open source Fligthgear (FLIGHTGEAR, 2024) para a visualização 3D.

De forma análoga, em (YIN et al., 2011) os autores propõem a modelagem e simulação de um drone utilizando o simulink para cálculos matemáticos e o Fligthgear para a visualização 3D.

Em (MEYER et al., 2012), os autores realizaram a simulação de um drone utilizando ROS e o simulador Gazebo, que está integrado ao ROS e fornece tanto soluções para implementação dos modelos, quanto recursos gráficos para exibição 3D.

Com base na dinâmica presente entre ROS e Gazebo, os autores em (HARIDEVAN et al., 2024) propõem um toolbox para simulação de drones com ROS2 e Gazebo, permitindo testes de planejamento de trajetória, algoritmos de controle e testes de percepção.

3.4 Linux embarcado e FPGA

FPGA, é um tipo de circuito integrado que pode ser configurado pelo usuário após a fabricação para realizar uma vasta gama de funções lógicas. Ao contrário dos circuitos integrados tradicionais, cujas funções são fixas após a fabricação, os FPGAs podem ser programados e reprogramados para executar diferentes tarefas. Este componente é composto por uma matriz de blocos lógicos programáveis, interconexões e blocos de entrada/saída. Esses componentes podem ser configurados para criar circuitos digitais

personalizados, permitindo a implementação de designs complexos (CHATTERJEE et al., 2002).

A literatura recente aponta que arquiteturas de computação heterogênea, especificamente System on Chip (SOC) que integram processadores application-class e tecido FPGA, são fundamentais para superar os gargalos de latência em robótica aérea (WAN et al., 2021). A integração nativa de aceleração de hardware no ecossistema ROS2 permite delegar tarefas críticas de controle e percepção para a FPGA, garantindo o determinismo que CPUs de propósito geral não conseguem oferecer sob carga (MAYORAL-VILCHES et al., 2023).

Em (PODLUBNE et al., 2021), os autores propõem o uso de um FPGA para aceleração de hardware em sistemas robóticos, onde os testes foram conduzidos em conjunto com o ROS2 e apresentaram resultados positivos.

O trabalho desenvolvido em (PODLUBNE et al., 2024), tem uma proposta análoga ao trabalho anterior, com o uso de um FPGA em conjunto com o ROS2 para múltiplos propósitos.

3.5 Hardware in the loop

O projeto e desenvolvimento de drones tem sido melhorado com a aplicação de técnicas de simulação, em especial simulações de HIL.

A simulação realista entre o hardware em teste e a simulação em tempo real proporciona uma maior segurança evitando danos ao hardware em testes, reduz o tempo e custo para depuração e reduz o trabalho de forma geral durante testes de validação (MIHALIč; TRUNTIč; HREN, 2022).

Além disso, HIL tem desempenhado um papel importante no campo as simulações de voo (DILLARD, 1998)

A validação de controladores para veículos aéreos não tripulados exige rigor temporal que simuladores baseados puramente em software, como o Gazebo, muitas vezes falham em fornecer devido ao jitter inerente a sistemas operacionais de propósito geral. Segundo (MORéAC et al., 2020), a utilização de arquiteturas HIL baseadas em FPGA permite a execução de simulações de dinâmica de voo em tempo real estrito, integrando-se nativamente ao ecossistema ROS. Pesquisas recentes de (NIETO et al., 2025) corroboram essa abordagem, demonstrando que a verificação de lógica crítica em FPGA através de interfaces ROS garante uma transição mais segura do ambiente simulado para o hardware físico, especialmente em cenários que demandam alta frequência de atualização dos sensores e atuadores (KöVARI; EBEID, 2021b).

3.6 Software in the loop

O Software in the loop (SIL) é definido como uma etapa crítica no ciclo de desenvolvimento de sistemas embarcados, onde o software de voo real (ou uma versão compilada para a máquina host) é executado em um computador de propósito geral, interagindo diretamente com modelos matemáticos que simulam a dinâmica da aeronave e os sensores. Segundo (BEARD; MCLAIN, 2012) em sua obra seminal sobre VANTs, o SIL permite a validação preliminar dos algoritmos de estimativa de estado e malhas de controle sem expor o hardware físico a riscos de acidentes. Esta abordagem desacopla a lógica de controle das incertezas do hardware, permitindo, conforme detalhado em (VALAVANIS; VACHTSEVANOS, 2015), que erros de implementação lógica e falhas de integração sejam detectados e corrigidos nas fases iniciais do projeto, reduzindo drasticamente os custos e o tempo de desenvolvimento antes da transição para testes em tempo real ou HIL.

Conforme estabelecido por (SHAH et al., 2017), ambientes de alta fidelidade visual permitem a validação segura de subsistemas de percepção e planejamento de trajetória, dissociando o desenvolvimento de software dos riscos físicos inerentes ao voo. A arquitetura do ROS2 facilita essa abordagem, permitindo, segundo(MACENSKI et al., 2022b), que os mesmos nós de controle sejam testados em ambientes simulados escaláveis antes do deploy. Contudo, como notado por (PANERATI et al., 2021), embora o SIL ofereça vantagens de velocidade para treinamento, ele carece da fidelidade temporal necessária para validar a resposta dinâmica fina do sistema embarcado, criando uma lacuna de realidade que deve ser posteriormente endereçada via HIL.

Capítulo 4

Materiais e Métodos

4.1 Diagrama geral da solução

A obtenção e análise dos dados, bem como a extração dos resultados dos mesmos, foram possíveis com a condução de diversos experimentos. Para isso diversos equipamentos e softwares foram utilizados, além da aplicação de técnicas e métodos distintos.

A partir disso uma solução foi delineada utilizando vários elementos, como ambiente de simulação, infraestrutura de comunicação, componentes físicos, dentre outros itens.

Os principais elementos empregados para este trabalho serão elencados no decorrer deste capítulo.

4.2 Kria KV260

A Kria KV260 é uma plataforma pronta para uso, ideal para o desenvolvimento de aplicativos de visão computacional sem exigir conhecimento complexo de design de hardware. O coração da Kria KV260 é o SOM em português Sistema no Módulo K26, fabricado pela Advanced Micro Devices (AMD), cujas especificações(AMD, 2023a) são descritas na Tabela 1.

Além disso o SOM K26 possui um FPGA, capaz de prover a possibilidade de desenvolvimento de aplicações aceleradas em Hardware. Todos esses recursos estão concentrados em uma pequena placa, com dimensões de 77 mm por 60 mm que é ilustrada pela Figura 5.

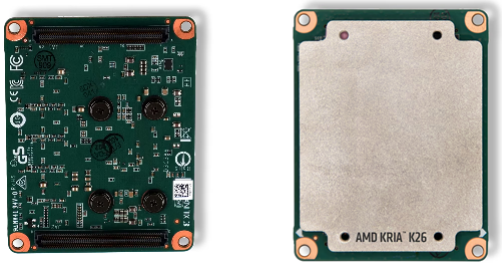
Para usufruir de todo esse potencial, utilizamos a placa Kria KV260(AMD, 2024), que fornece um conjunto de conexões que incluem, mas não se limitam a, portas Universal Serial Bus (USB), conexões para vídeo, rede, dentre outras. A Kria KV260 é ilustrada pela Figura 6.

Tabela 1 – Especificações do SOM K26

Componente	Especificação
Processador de Aplicação	Quad-core Arm® Cortex®-A53 MPCore™ até 1.5 GHz
Processador de tempo real	Dual-core Arm Cortex-R5F MPCore até 600 MHz
Unidade de processamento Gráfico	Mali™-400 MP2 até 667 MHz
Memória RAM (Random Access Memory)	4GB 64-bit DDR4 (Double Data Rate)

Fonte: (AMD, 2023a)

Figura 5 – SOM K26



Fonte: (AMD, 2023b)

Todas as entradas e saídas presentes na Kria KV260, bem como sua relação com o SOM K26 podem ser observadas no diagrama de blocos presente na Figura 7. Deve ser salientado que a Kria KV260 tem foco no desenvolvimento de aplicações de visão computacional, mas oferece a flexibilidade necessária para aplicações diversas, como no desenvolvimento de robôs, drones e demais equipamentos.

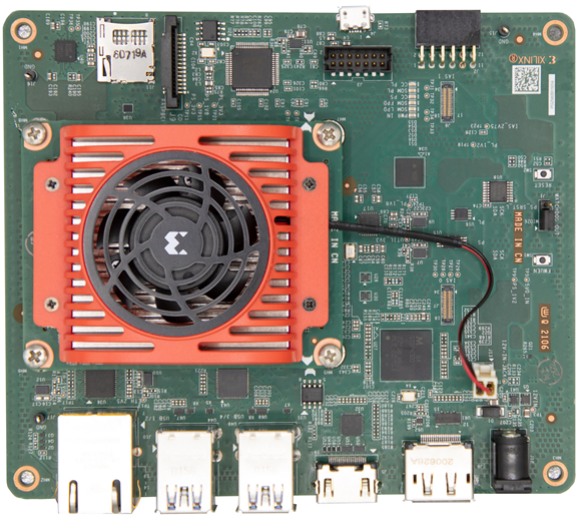
4.3 ROS2

Para o desenvolvimento de robôs, o middleware mais utilizado é o ROS (OPEN-ROBOTICS, 2024c), pois oferece uma plataforma de software padrão para desenvolvedores, que são utilizados a partir da pesquisa e da prototipagem até implantação e produção.

Mesmo com a adoção do ROS por grande parte da comunidade de entusiastas e estudantes, essa solução enfrenta resistência por parte da indústria, devido a pontos importantes, como por exemplo, questões de segurança e a falta de suporte para sistemas em tempo real. Para atender a essa demanda, os mantenedores do ROS, criaram o ROS2 com a premissa de sanar as deficiências do ROS que o impediam de ser adotado pela indústria.

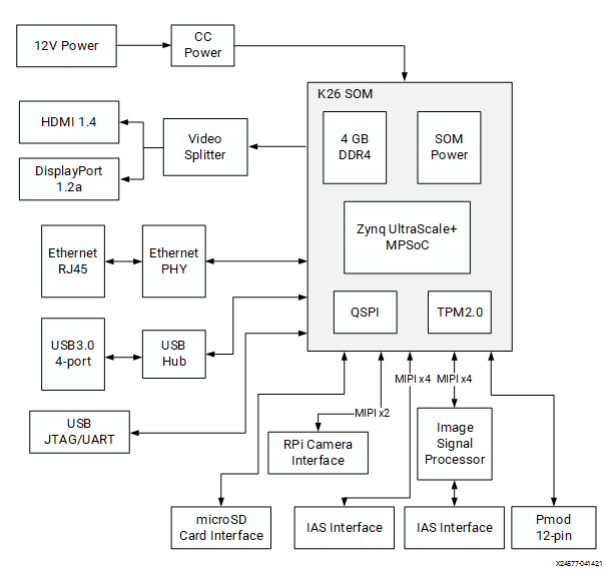
Recentemente, a Open Robotics, mantenedora do ROS, anunciou que o ROS seria descontinuado para focar os esforços de desenvolvimento no ROS2. Dessa forma a versão Noetic Ninjemys do ROS é a ultima versão do ROS(OPEN-ROBOTICS, 2024a), tendo

Figura 6 – Kria KV260



Fonte: (DIGIKEY, 2024)

Figura 7 – Diagrama de blocos da Kria KV260



Fonte: (AMD, 2024)

seu End of Life (EOL), em português fim da vida, anunciado. Essa versão é compatível oficialmente apenas com o Ubuntu 20.04.

Para este trabalho foi adotada a versão Humble Hawksbill do ROS2, uma vez que o ROSNoetic Ninjemys atingiu o EOL e o ROS2 traz significativas melhoras, como por exemplo segurança aprimorada, capacidade de trabalhar com sistemas de tempo real e suporte multiplataforma.

4.4 Linguagem de Programação

O ROS oferece bibliotecas que permitem desenvolver nós e componentes em diferentes linguagens. C++ e Python são as duas linguagens mais difundidas na prática e na documentação do ROS, sendo suportadas por bibliotecas oficiais que expõem a Interface de Programação de Aplicações (API) do sistema.

Python é frequentemente usado para protótipos rápidos, scripts de integração, testes e ferramentas de alto nível, devido à sua sintaxe concisa e ao ecossistema de pacotes. A estrutura de pacotes e a conveniência para desenvolvimento ágil tornam Python uma escolha natural para desenvolvimento exploratório e automação de testes.

C++ destaca-se nos seguintes pontos: maior desempenho de execução, melhor controle de alocação de memória, facilidade de integração com código nativo e bibliotecas em tempo real, e tipagem estática que ajuda a detectar erros em tempo de compilação. A API rclcpp é projetada para oferecer uma interface C++ eficiente e adequada a aplicações com requisitos de latência e determinismo

Além de C++ e Python, existem client libraries e bindings para outras linguagens (por exemplo, JavaScript, Rust, Ada), o que permite escolher a linguagem conforme requisitos específicos do projeto, embora com menor suporte comunitário e menos exemplos práticos do que C++/Python.

Neste trabalho foi escolhido C++ por razões de desempenho, conforme testes que serão apresentados no proximo capítulo.

4.5 Simuladores

A confecção de um protótipo físico diretamente a partir do projeto, muitas vezes é uma abordagem equivocada, que pode resultar em perdas financeiras e de tempo. Por isso um protótipo em ambiente simulado é comumente utilizado para confirmar as diretrizes e especificações do projeto, reduzindo significativamente os custos.

A simulação é parte fundamental deste trabalho, uma vez que o uso desta ferramenta permite a economia de tempo e dinheiro. Para isso a adoção de uma ferramenta eficiente se torna imprescindível

Existem diversas opções de software para simulação, com diferentes focos e abordagens. Para este trabalho priorizou-se a integração com o ROS2, facilidade de implementação e adequação ao ambiente proposto.

Dentre as opções disponíveis tres se destacaram: Gazebo, Webots e Matlab/Simulink. Sendo a última a opção escolhida para este trabalho.

4.5.1 Gazebo

Este é um dos simuladores mais tradicionais da comunidade ROS, com suporte oficial e ampla documentação. Possui pacotes que permitem comunicação bidirecional entre o ROSe a simulação. Sua aplicação típica inclui testes de robos móveis, drones e manipuladores em ambientes virtuais complexos.

4.5.2 Webots

Simulador multiplataforma que possui modelos prontos de robos e sensores, oferece suporte a ROS2. Comumente utilizado em prototipos e simulações de robos humanoides ou móveis.

4.5.3 Matlab/Simulink

Ambiente de modelagem e simulação numérica amplamente utilizado em engenharia de controle. Conecta-se ao ROS via ROS Toolbox, permitindo enviar e receber mensagens ROS diretamente. Suporta técnicas de SIL e HIL, permitindo desenvolvimento e validação de algoritmos de controle, especialmente em sistemas críticos como drones e veículos autônomos.

4.6 Software in the Loop

SIL é uma técnica de verificação em que o código gerado a partir de um modelo (por exemplo, Simulink) é compilado e executado no computador de desenvolvimento para comparar seu comportamento com a simulação do modelo original.

Neste trabalho, controlador é modelado como um nó ROS2, que é executado como processo separado no host (SIL). Esse processo recebe entradas (sensores simulados) e produz saídas (comandos ao atuador).

Dessa forma o modelo virtual da bancada é executado no Simulink, que envia as informações de angulo para o nó contendo o controlador, este por sua vez devolve a ação de controle para a simulação. Toda a comunicação ocorre por meio de tópicos e mensagens no ROS2 e é possivel observar o comportamento da simulação.

4.7 Hardware in the Loop

HIL é uma técnica de simulação usada principalmente no desenvolvimento e teste de sistemas embarcados, como os encontrados em automóveis, aeronaves e eletrônicos industriais. O conceito central do HIL envolve a utilização de um sistema real (ou parte dele) conectado a um simulador que emula o restante do ambiente ou sistema. Isso permite

testar e validar o hardware em condições controladas e reproduzíveis sem a necessidade de utilizar o sistema completo ou colocar em risco componentes críticos.

De forma análoga ao SIL, controlador é modelado como um nó ROS2, que pode ser executado em um computador ou no FPGA. A bancara real fornece a informação de angulo via tópicos e mensagens no ROS2, o controlador obtém essas informações e envia uma ação de controle para a bancada. O uso do FPGA permite que o Controlador Eletrônico de Velocidade (ESC) dos motores da bancada recebam a ação de controle diretamente das portas Entrada e Saída (I/O) do FPGA, uma abordagem de baixa latencia e tempo real.

4.8 Comunicação

A intercomunicação entre o equipamento que está executando a simulação e o equipamento que fará o controle da mesma requer uma interface de comunicação, uma rede para trafegar os dados.

Escolher esta interface requer alguns cuidados, como a latência, largura de banda e a presença da mesma interface em ambos os lados. Como a simulação será executada em um microcomputador e os controles serão executados na Kria KV260 as opções de interface de comunicação para implementação do HIL são limitadas, as principais disponíveis são as interfaces USB, Interface Universal Asynchronous Receiver/Transmitter (UART) e a interface Ethernet.

Com o objetivo de reduzir a complexidade e manter altas taxas de transferência e baixa latência, optou-se por utilizar a interface Ethernet, como a conexão ocorrerá entre os dispositivos e não será aberta a internet é possível adotar a comunicação em broadcast o que simplifica significativamente a implementação e reduz a latência, uma vez que a criptografia e segurança podem ser suprimidas.

Observando a variedade de configurações que podem ocorrer dentro de uma rede ethernet, algumas topologias possíveis foram adotadas, com o intuito de prover maior flexibilidade e explorar o cenário de menor latência.

Para este trabalho foi adotada a topologia de conexão direta via cabo de rede, sem equipamentos intermediários para gestão da rede. A escolha se baseia nos resultados de latencia, conforme testes que serão apresentados no proximo capítulo.

4.9 Computadores

O uso de computadores para este trabalho se faz necessário para execução da simulação no Matlab/Simulink, manutenção da rede ROSseus tópicos e nós. Além disso possibilitaram os testes em SIL e HIL. Uma variedade de equipamentos foi utilizada,

como computadores portáteis e estações de trabalho, a configuração destes equipamentos está listada na Tabela 2 a seguir.

Tabela 2 – Especificação dos computadores

Componente	Laptop	Desktop
CPU	AMD Ryzen 7 5800H	Intel Core i7 4790
RAM	16GB DDR4	16GB DDR3
DISCO	SSD 2TB	SSD 240GB
GPU	RTX 3060	HD Graphics 4600

Fonte: Autor

4.10 Modelo da Bancada

O uso de um modelo virtual da bancada faz-se necessário para o fechamento da malha nos testes de SIL e HIL. Para isso foram confeccionados dois modelos virtuais, sendo um deles um script no matlab e o outro uma implementação no simulink.

4.10.1 Modelo da Bancada - Matlab

O script de simulação representa um modelo bastante simplificado da bancada, um sistema dinâmico de corpo rígido com um grau de liberdade, modelado como uma barra homogênea pivotada no centro de gravidade. O objetivo é controlar o ângulo de inclinação θ através da atuação diferencial de dois propulsores.

A base da simulação é a Segunda Lei de Newton para Rotação. Para um corpo girando em torno de um eixo fixo, a soma dos momentos (torques) é igual ao produto do momento de inércia pela aceleração angular.

$$\sum \tau = I\ddot{\theta} \tag{1}$$

O momento de inércia (I) para uma barra delgada de massa m e comprimento L, girando pelo seu centro geométrico, é definido pela integração da distribuição de massa ao longo do comprimento.

$$I = \int r^2 \, dm = \frac{1}{12} mL^2 \tag{2}$$

Esta fórmula é derivada da mecânica clássica para barras uniformes e determina a resistência do corpo a mudanças na sua velocidade de rotação. Quanto maior o (I), mais torque é necessário para acelerar a barra (CORNWELL, 2012).

No passo da simulação, calculamos o torque líquido (τ_{net}) que atua no sistema naquele instante. Este torque é composto por três elementos: Atuação, Perturbação e Amortecimento.

$$\tau_{net} = \tau_{motores} + \tau_{vento} - \tau_{atrito} \tag{3}$$

O drone gera torque variando a força de empuxo (F) dos motores localizados nas extremidades. Seja d a distância do pivô ao motor ($d = L/2$).

$$\tau_{motores} = (\vec{F_{esquerda}} \cdot d) - (\vec{F_{direita}} \cdot d) \tag{4}$$

O torque é o produto vetorial da força pelo braço de alavanca. Como supõe-se que as forças são perpendiculares à barra, a relação torna-se escalar. O controle de atitude é obtido pela diferença de empuxo entre os motores (BEARD; MCLAIN, 2012).

O termo (τ_{vento}) é introduzido na equação dinâmica como uma perturbação aditiva exógena. Fisicamente, este torque representa o momento gerado pela força de arrasto aerodinâmico (F_d) que uma rajada de vento exerce sobre a área de superfície da barra e das hélices, atuando a uma distância (r) do pivô ($\tau = F_d \cdot r$).

$$\tau_{\text{vento}}[k] = \begin{cases} D, & \text{se } k_{\text{início}} \leq k \leq k_{\text{fim}} \\ 0, & \text{caso contrário} \end{cases} \tag{5}$$

Em Engenharia de Controle, a injeção de perturbações do tipo degrau é o método padrão para validar a robustez do sistema. O objetivo é verificar a Rejeição de Perturbação, analisando se o controlador consegue retornar o erro a zero mesmo na presença de forças externas constantes, ou se o sistema apresentará um erro de estado estacionário, o que ocorre em controladores puramente Proporcionais sem termo Integral(FRANKLIN; POWELL; EMAMI-NAEINI, 2014)

Na simulação, modelamos este fenômeno como uma função do tipo degrau temporário (pulso retangular):

Para evitar oscilações perpétuas (inconsistentes com a realidade física), introduz-se um termo de amortecimento proporcional à velocidade angular ($\dot{\theta}$).

$$\tau_{atrito} = b\dot{\theta} \tag{6}$$

Em sistemas mecânicos reais, a energia é dissipada. O modelo de amortecimento viscoso linear (b) é a aproximação padrão para pequenas velocidades em dinâmica de sistemas (OGATA, 2021).

4.11 Controlador

O controlador é responsável por receber o angulo atual da bancada e ajustar a força dos propulsores para atingir o angulo alvo, podendo ser implementado junto à simulação ou em forma de um nó independente. Ambas as abordagens serão exploradas.

Como parte dos objetivos deste trabalho visa a versatilidade, uma gama de controladores distintos podem ser utilizados para avaliar diferentes caracteristicas, como tempo de resposta, estabilidade, dentre outros.

Optou-se por um controlador do tipo proporcional por sua simplicidade. Opções mais complexas podem ser adotadas conforme a necessidade.

4.11.1 Controlador integrado ao Matlab

O sistema utiliza um controlador Proporcional para calcular o erro e determinar a ação necessária.

$$e_k = \theta_{\text{desejado}} - \theta_{k_{\text{atual}}} \tag{7}$$

Onde o erro (e_k) é utilizado para obter o torque de comando desejado (u_k).

$$u_k = K_p \cdot e_k \tag{8}$$

Para converter este torque abstrato em forças físicas para os motores, utilizamos uma Matriz de Alocação, considerando uma força de sustentação base (F_{base}).

$$F_{\text{esquerda}} = F_{\text{base}} + \frac{u_k}{2d} \tag{9}$$

$$F_{\text{direita}} = F_{\text{base}} - \frac{u_k}{2d} \tag{10}$$

Esta alocação lineariza a atuação em torno de um ponto de operação (F_{base}), permitindo que o controlador linear opere motores que só produzem força positiva (CORKE, 2017)

Como o computador não resolve equações diferenciais contínuas analiticamente em tempo real, utilizou-se o Método de Euler para discretizar o tempo. Isso transforma as equações diferenciais em equações de diferenças algébricas.

Dado um passo de tempo (Δt), o estado no instante futuro (k+1) é calculado baseando-se no estado atual (k). Isso é feito com o calculo da aceleração

$$\ddot{\theta}_k = \frac{\tau_{\text{net}}}{I} \tag{11}$$

seguido da atualização da velocidade por meio da integração da aceleração

$$\dot{\theta}_{k+1} = \dot{\theta}_k + (\ddot{\theta}_k \cdot \Delta t) \tag{12}$$

e finalizando com a atualização da posição por meio da integração da velocidade.

$$\theta_{k+1} = \theta_k + (\dot{\theta}_{k+1} \cdot \Delta t) \tag{13}$$

O Método de Euler é a aproximação de primeira ordem da Série de Taylor. Embora existam métodos mais precisos, para passos de tempo pequenos em dinâmicas de drones simples, o método de Euler oferece um equilíbrio adequado entre precisão e custo computacional para simulações em tempo real (CHAPRA, 2014).

4.11.2 Controlador em nó independente

Nesta etapa do projeto, o sistema de controle deixa de ser uma abstração matemática executada dentro da simulação e passa a ser um Controlador Digital Discreto independente.

Teoricamente, o nó opera como um sistema de dados amostrados. Diferente da simulação contínua, onde as variáveis existem em todos os instantes infinitesimais de tempo t, o nó de controle opera em instantes discretos k, definidos por uma taxa de amostragem ou a frequência de publicação das mensagens ROS.

Algebricamente, a Lei de Controle para um controlador Proporcional permanece idêntica na forma, mas muda no domínio, ao invés da Equação 8, temos :

$$u[k] = K_p \cdot e[k] \tag{14}$$

Onde (k) representa o k-ésimo instante de amostragem.

Embora a fórmula algébrica seja a mesma, a implementação no nó introduz dois fenômenos físicos que não existiam na simulação pura.

Latencia, agora existe o tempo de serialização, transporte DDS do ROS2 e processamento.

Segurador de Ordem Zero, o valor de torque calculado $(u[k])$ permanece constante até que a próxima mensagem $(u[k+1])$ chegue.

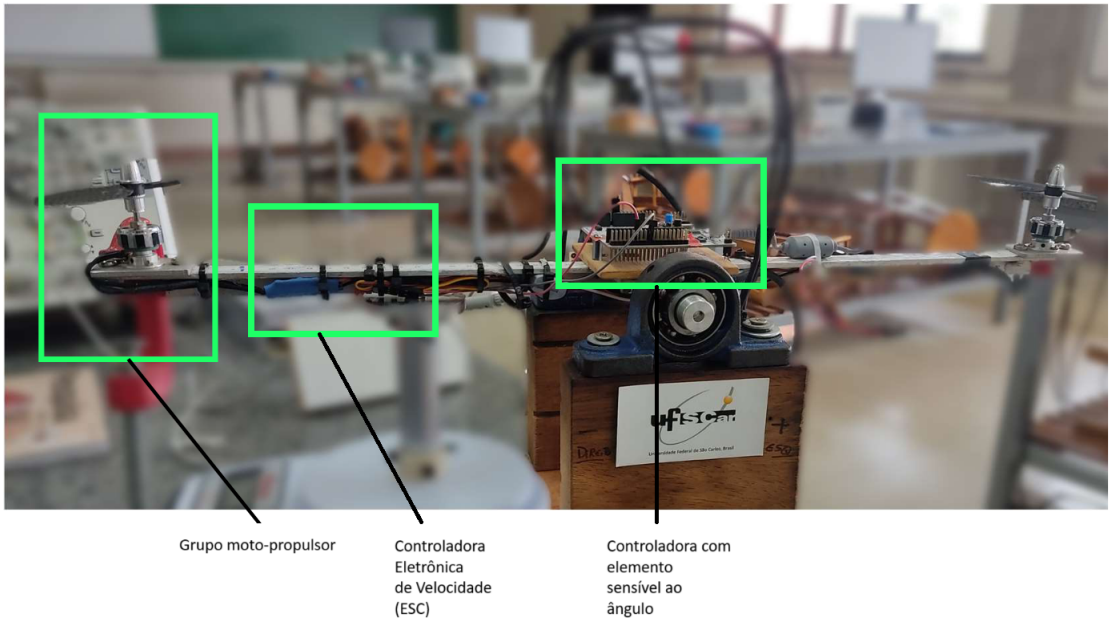
4.12 Bancada Experimental

A bancada foi projetada para simular o eixo de Pitch de um veículo multirotor. Ela é composta por três subsistemas principais: grupo motopropulsor (motor brushless + hélice); ESC e unidade de medição/controle de ângulo. Todos os componentes estão fixados em uma base rígida que garante repetibilidade e segurança durante os ensaios.

O objetivo principal é controlar os motores para garantir a estabilidade e atingir o angulo determinado.

A Figura 8 ilustra um protótipo da bancada e seus principais elementos.

Figura 8 – Bancada experimental e seus componentes



Fonte: Autor

Capítulo 5

Resultados

5.1 Avaliação de desempenho das linguagens

Como mencionado anteriormente, os nós ROS podem ser escritos em diferentes linguagens de programação, sendo C++ e Python as mais comuns. Para conduzir a avaliação de desempenho das linguagens foi construído um ambiente composto por dois nós e uma mensagem personalizada executado no sistema Ubuntu.

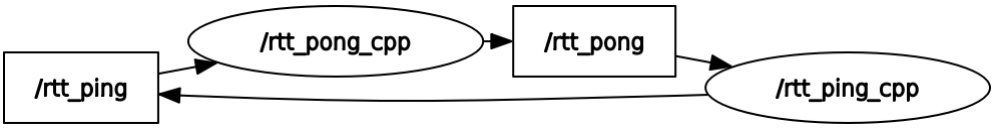
5.1.1 Estrutura dos nós

Para cada uma das linguagens foram escritos dois nós, denominados de "ping"e "pong". O fluxo ocorre com o envio de uma mensagem pelo nó ping, contendo um identificador da mensagem e um carimbo temporal que registra o momento da partida da mensagem com destino ao nó pong. Ao atingir o nó pong, um novo carimbo de tempo é registrado na mensagem e a mensagem é preparada para ser devolvida ao nó ping. Ao enviar a mensagem para o nó ping, um novo carimbo de tempo é registrado na mensgem, que ao ser rebida pelo nó ping registra um ultimo carimbo de tempo. Ao término do ciclo as informações contidas na mensagem são registradas em um arquivo de valores separados por vírgulas para analise posterior. O fluxo da mensagem atraves dos nós e tópicos pode ser visto em Figura 9 e Figura 10.

5.1.2 Metodologia de Coleta e Tratamento de Dados

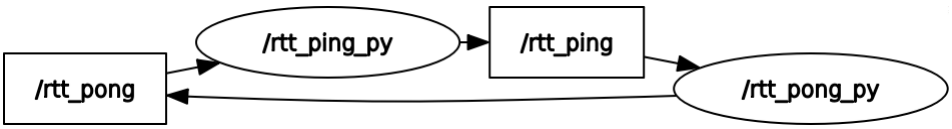
Para quantificar e comparar o desempenho dos nós desenvolvidos em C++ e Python, foram realizados experimentos controlados com uma frequência de amostragem de 10 Hz. O conjunto de dados totalizou N=100000 amostras por linguagem.

Figura 9 – Diagrama de nós e tópicos em C++



Fonte: Autor

Figura 10 – Diagrama de nós e tópicos em Python



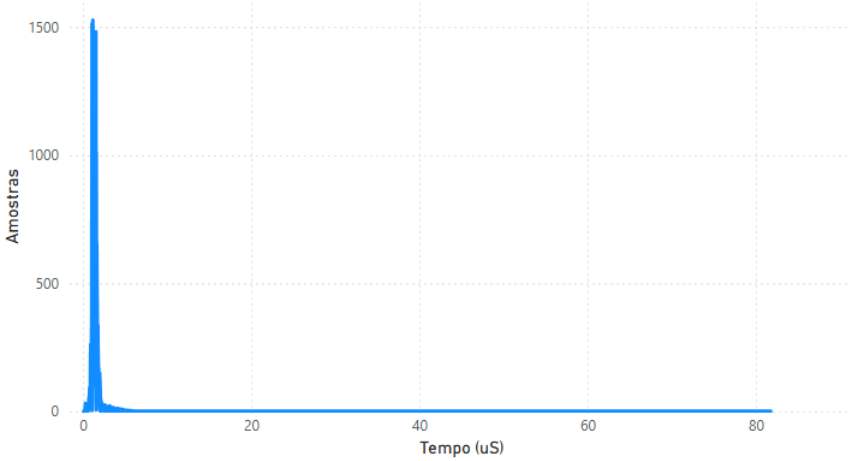
Fonte: Autor

Para garantir a análise em regime estacionário e mitigar o efeito de "Cold Start"(fase de descoberta da rede DDS e alocação dinâmica inicial), as primeiras 500 amostras de cada experimento foram descartadas. A unidade de medida temporal adotada para todas as métricas foi o microssegundo.

5.1.3 Distribuição das amostras

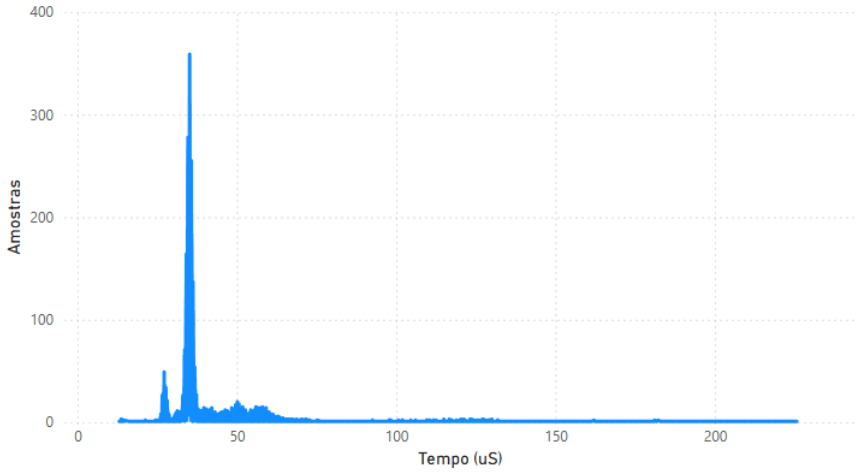
As amostras estão compiladas nos graficos a seguir, sendo a Figura 11 as amostras da linguagem c++ e a Figura 12 as amostras da linguagem python.

Figura 11 – Distribuição das amostras - C++



Fonte:Autor

Figura 12 – Distribuição das amostras - Python



Fonte:Autor

5.1.4 Análise do Tempo de Processamento

A análise do tempo de processamento permite avaliar diretamente a eficiência do Python versus C++, independente da infraestrutura de rede.

Esta métrica isola o overhead computacional introduzido pela linguagem de programação. Mede o intervalo de tempo entre a chegada da mensagem no nó pong e o envio da resposta, excluindo o tempo de transporte da rede.

$$T_{proc} = t_{pong\_tx} - t_{pong\_rx} \tag{15}$$

Os resultados presentes na tabela Tabela 3 demonstram uma disparidade significativa entre as implementações. O nó em C++ apresentou uma mediana de $1,35$ μs, enquanto a implementação em Python registrou $35,35$ μs.

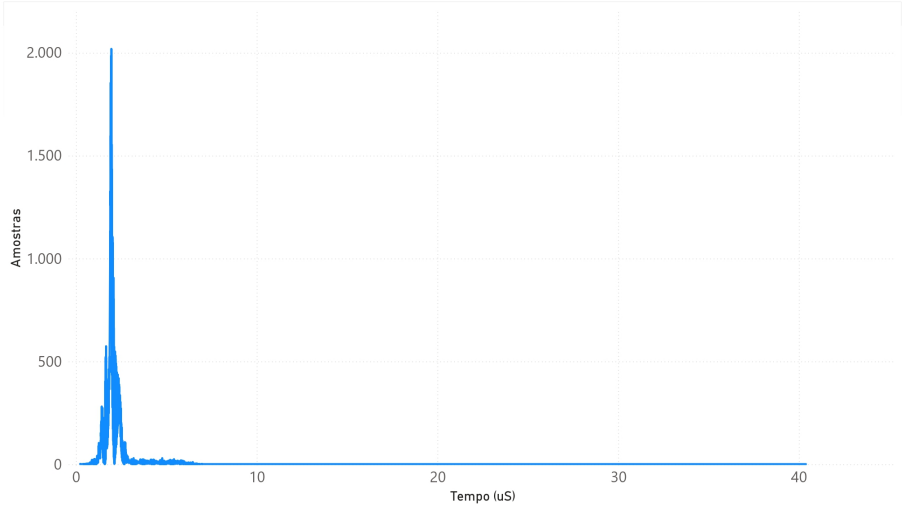
Tabela 3 – Estatísticas de Desempenho

Estatística	C++ (us)	Python (us)
Média	1,44	39,87
Mediana	1,35	35,35
Desvio Padrão	0,81	15,44
Mínimo	0,09	13,10
Máximo	81,77	225,71
99% (P99)	3,72	123,25

Fonte: Autor

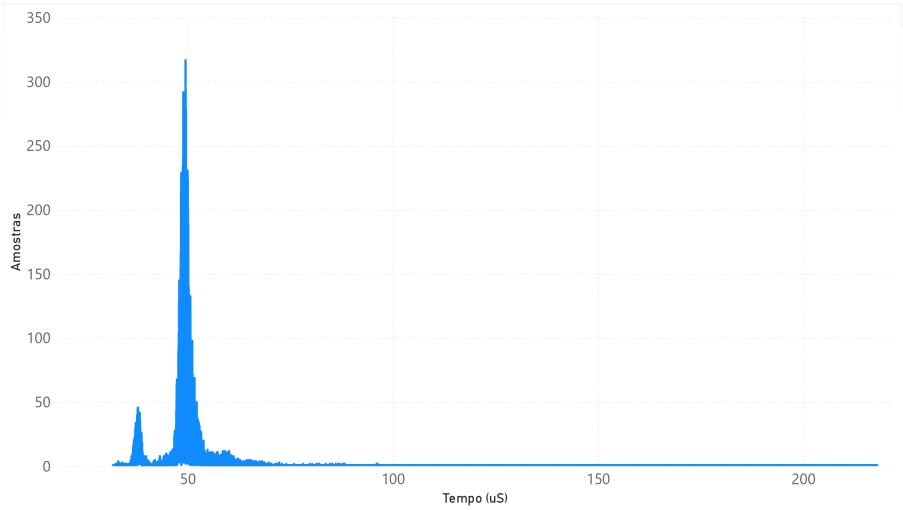
De forma análoga é possivel analisar o tempo de processamento quando há uma carga computacional dentro do nó pong. Os ensaios com carga têm como objetivo avaliar a robustez, a previsibilidade e o determinismo do ambiente de desenvolvimento quando o hardware é submetido a condições extremas de estresse de processamento. Para isso foi inserido o calculo de inversão de uma matriz 3x3. Os resultados estão compilados nos graficos a seguir, sendo a Figura 13 as amostras da linguagem C++ e a Figura 14 as amostras da linguagem Python.

Figura 13 – Distribuição das amostras - C++ com Carga



Fonte:Autor

Figura 14 – Distribuição das amostras - Python com Carga



Fonte:Autor

Com base nos dados analíticos apresentados na Tabela 4, a superioridade de desempenho do C++ em relação ao Python fica quantitativamente evidente através da redução drástica no tempo médio de processamento, no tempo máximo de execução e no desvio padrão.

Tabela 4 – Estatísticas de Desempenho com Carga

Estatística	C++ (us)	Python (us)
Média	2,13	49,38
Mediana	1,96	49,29
Desvio Padrão	0,72	5,77
Mínimo	0,23	31,74
Máximo	40,39	218,09
99% (P99)	5,63	65,66

Fonte: Autor

Com base nos dados obtidos, a linguagem com melhor desempenho para a aplicação proposta é C++.

5.2 Avaliação de desempenho das topologias de rede

Dentre as opções disponíveis de comunicação entre os dispositivos utilizados neste trabalho, o uso da rede local se mostrou a escolha mais racional, pela facilidade de uso, dsponibilidade em todos os dispositivos e confiabilidade. O uso da rede local pode ter diferentes topologias, isso define quais os meios para transmissão da informação e se equipamentos adicionais serão utilizados para gerenciar a rede. A avaliação das topologias disponíveis se dará por meio do Tempo de Ida e Volta, o RTT O RTT mede a latência ponta-a-ponta da comunicação, englobando a serialização da mensagem, o transporte pela camada ROS Middleware (RMW), a desserialização e o tráfego de rede (loopback). É calculado como a diferença entre o momento em que a mensagem ping é recebida de volta pelo nó remetente e o momento do envio original.

$$RTT = t_{ping\_rx} - t_{ping\_tx} \tag{16}$$

Esta métrica é crítica para sistemas de controle em malha fechada, onde o atraso total entre a leitura de um sensor e a atuação determina a estabilidade do sistema.

5.2.1 Metodologia de Coleta e Tratamento de Dados

Para quantificar e comparar o desempenho das topologias, foram realizados experimentos controlados com uma frequência de amostragem de 10 Hz. O conjunto de dados totalizou N=100000 amostras por linguagem.

Para garantir a análise em regime estacionário e mitigar o efeito de "Cold Start"(fase de descoberta da rede DDS e alocação dinâmica inicial), as primeiras 500 amostras de cada experimento foram descartadas. A unidade de medida temporal adotada para todas as métricas foi o microssegundo.

Todas as coletas foram realizada com os nós escritos em C++, devido a seu melhor desempenho comprovado anteriormente, também foi adotado o ambiente descrito no teste anterior.

Como base para comparação foi conduzido um teste com ambos os nós sendo executados no mesmo equipamento, resultando no menor RTT possivel, uma vez que não ha necessiadade da rede ou de quaisquer outros dispositivos para a aferição. Estes resultados serão utilizados como controle para avaliação das topologias disponíveis.

Os resultados do controle estão expostos na Tabela 5.

Tabela 5 – Estatísticas de Desempenho - RTT

Estatística	RTT (us)
Média	725,11
Mediana	723,15
Desvio Padrão	81,98
Mínimo	171,52
Máximo	9948,79
99% (P99)	921,10

Fonte: Autor

5.2.2 Topologia rede cabeada com roteador

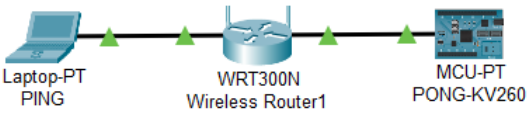
Esta é uma topologia tradicional das redes de computador, onde um roteador é responsavel por gerenciar todos os clientes de rede, direcionar os pacotes, atribuir endereços de rede, dentre outras responsabilidades.

Foram realizados dois testes nesta topologia, no primeiro dois computadores e um roteador foram utilizados, o nó ping foi executado em um computador e o nó pong no outro. No segundo teste o computador que executava o nó pong foi substituido por um SOM com Linux embarcado . Os diagramas podem ser vistos na Figura 15 e na Figura 16.

Figura 15 – Topologia rede cabeada com roteador e computadores

Fonte:Autor

Figura 16 – Topologia rede cabeada com roteador e SOM com Linux embarcado



Fonte:Autor

Como a gestão da rede é de responsabilidade do roteador, basta configura-lo corretamente e os clientes funcionarão automaticamente sem maiores complicações. Além disso todos os clientes são conectados atraves de cabos metálicos.

Ambos os resulltados podem ser vistos na Tabela 6.

Tabela 6 – Estatísticas de Desempenho - RTT - Topologia rede cabeada com roteador

Estatística	Computador (us)	FPGA (us)
Média	1184,67	793,52
Mediana	1121,50	784,13
Desvio Padrão	160,88	96,44
Mínimo	695,74	392,22
Máximo	1863,59	2383,77
99% (P99)	1659,06	1130,41

Fonte: Autor

Pode-se observar que o RTT do SOM com Linux embarcado está próximo ao do controle, enquanto o computador teve uma piora perceptível.

5.2.3 Topologia rede mista com roteador

Esta é uma topologia tradicional das redes de computador, onde um roteador é responsavel por gerenciar todos os clientes de rede, direcionar os pacotes, atribuir endereços

de rede, dentre outras responsabilidades.

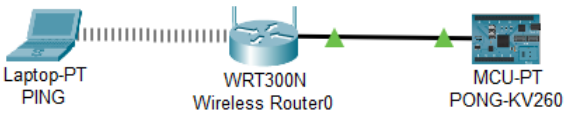
Foram realizados dois testes nesta topologia, no primeiro dois computadores e um roteador foram utilizados, o nó ping foi executado em um computador e o nó pong no outro. No segundo teste o computador que executava o nó pong foi substituido por um SOM com Linux embarcado . Os diagramas podem ser vistos na Figura 17 e na Figura 18.

Como a gestão da rede é de responsabilidade do roteador, basta configura-lo corretamente e os clientes funcionarão automaticamente sem maiores complicações. Além disso o cliente que executa o nó ping está conectado por meio de rede sem fio, enquanto o outro esta conectado atraves de cabos metálicos.

Figura 17 – Topologia rede mista com roteador e computadores

Fonte:Autor

Figura 18 – Topologia rede mista com roteador e SOM com Linux embarcado



Fonte:Autor

Ambos os resultados podem ser vistos na Tabela 7.

Tabela 7 – Estatísticas de Desempenho - RTT - Topologia rede mista com roteador

Estatística	Computador (ms)	FPGA (ms)
Média	72,03	74,05
Mediana	63,00	65,28
Desvio Padrão	69,97	68,09
Mínimo	1,27	0,88
Máximo	2050,23	1550,27
99% (P99)	196,30	229,83

Fonte: Autor

Pode-se observar que o RTT desta topologia é muito elevado, tornando esta opção inviável.

5.2.4 Topologia rede ponto a ponto

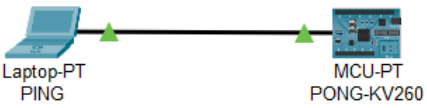
Nesta topologia não ha um equipamento intermediário controlando o trafego dos dados, ao invés disso, os clientes são ligados diretamente entre si por cabos metálicos, contudo exige configuração adicional das interfaces de rede, uma vez que não há intermediários na rede para esse gerenciamento.

Foram realizados dois testes nesta topologia, no primeiro dois computadores e um roteador foram utilizados, o nó ping foi executado em um computador e o nó pong no outro. No segundo teste o computador que executava o nó pong foi substituido por um SOM com Linux embarcado . Os diagramas podem ser vistos na Figura 19 e na Figura 20.

Figura 19 – Topologia rede ponto a ponto entre computadores

Fonte:Autor

Figura 20 – Topologia rede ponto a ponto entre computador e SOM com Linux embarcado



Fonte:Autor

Ambos os resultados podem ser vistos na Tabela 8.

Esta topologia apresentou o melhor resultado dentre as opçoes disponíveis e será adotada neste trabalho.

5.3 Avaliação de desempenho entre simulador e controlador

Com o intuito de averiguar se há perda de desempenho na comunicação entre o simulador e o nó de controle um cenário de teste foi construído, baseado no experimento

Tabela 8 – Estatísticas de Desempenho - RTT - Topologia rede ponto a ponto

Estatística	Computador (us)	FPGA (us)
Média	1184,53	741,12
Mediana	1120,35	727,90
Desvio Padrão	168,31	100,10
Mínimo	713,74	393,08
Máximo	2108,88	1902,17
99% (P99)	1596,87	1090,73

Fonte: Autor

anterior. O nó ping foi adaptado para ser exeutado dentro do Matlab, o nó pong será executado no SOM com Linux embarcado , em sua versão C++ e atuará como o nó do controlador. A topologia adotada será a ponto a ponto.

Para quantificar e comparar o desempenho da comunicação entre simulador e controlador, foram realizados experimentos controlados com uma frequência de amostragem de 10 Hz. O conjunto de dados totalizou N=100000 amostras.

Para garantir a análise em regime estacionário e mitigar o efeito de "Cold Start"(fase de descoberta da rede DDS e alocação dinâmica inicial), as primeiras 500 amostras de cada experimento foram descartadas. A unidade de medida temporal adotada para todas as métricas foi o microssegundo.

Similar ao teste de topologias, será adotado o RTT como parametro para analise do desempenho de comunicação.

Devido ao fato do Matlab ser executado em Windows, e a integração com oROS2 ser feita por meio de toolbox propietário do matlab, o teste RTT apresentou valores maiores, conforme a Tabela 9.

Tabela 9 – Estatísticas de Desempenho - RTT - Entre Simulação e Controle

Estatística	RTT (ms)
Média	6,95
Mediana	6,70
Desvio Padrão	1,03
Mínimo	6,29
Máximo	32,37
99% (P99)	12,20

Fonte: Autor

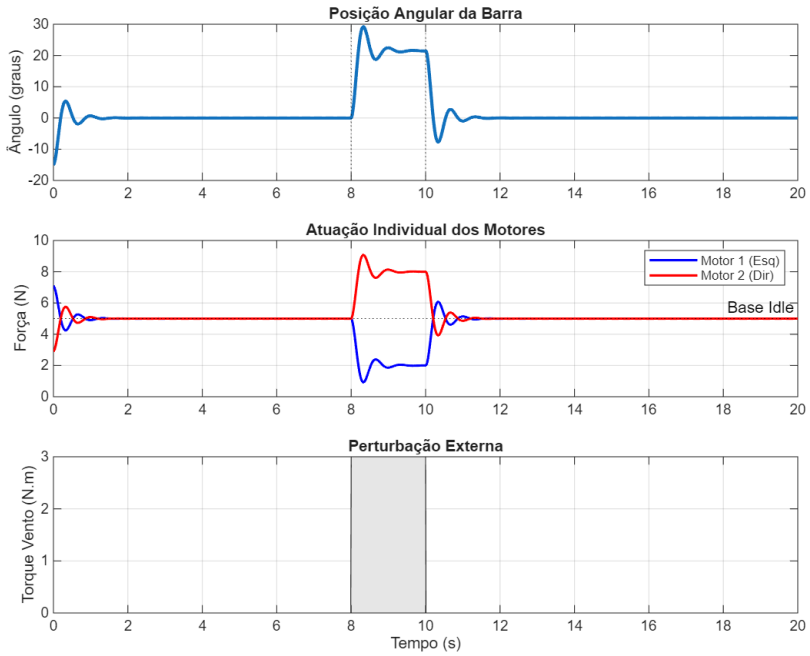
5.4 Validação do Modelo - Matlab com controlador integrado

A execução do modelo da bancada em Matlab, que contempla o controlador proporcional no mesmo script, fornece os resultados da ação de controle no melhor cenário possível, tempo contínuo e sem atrasos de comunicação.

Para estes resultados, a posição inicial da barra foi -15°, o tempo de simulação foi de 20 segundos e o valor do ganho proporcional Kp foi de 8.

Os gráficos presentes na Figura 21 Representam a posição angular da barra, a atuação dos motores e a perturbação externa.

Figura 21 – Resultados da Simulação



Fonte:Autor

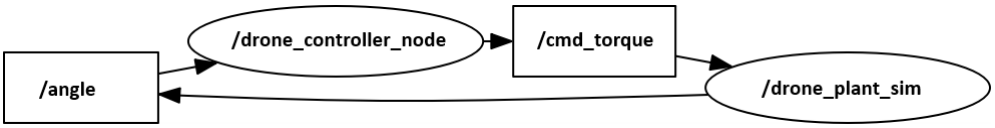
5.5 Validação do Modelo - Matlab com controlador em nó

A execuçãodo modelo em Matlab com um nó responsável pelo controle caracteriza uma implementação de SIL, uma vez que o nó não sabe a origem da informação nem o que controlará, não distinguindo entre uma simulação ou um cenário real.

Para este cenário o controlador foi implementado em um nó C++, executado em um SOM com Linux embarcado, com uma topologia de rede ponto a ponto.

O fluxo das mensagens atraves dos nós e tópicos pode ser visto em Figura 22

Figura 22 – Diagrama de nós e tópicos entre MATLAB e Controlador



Fonte: Autor

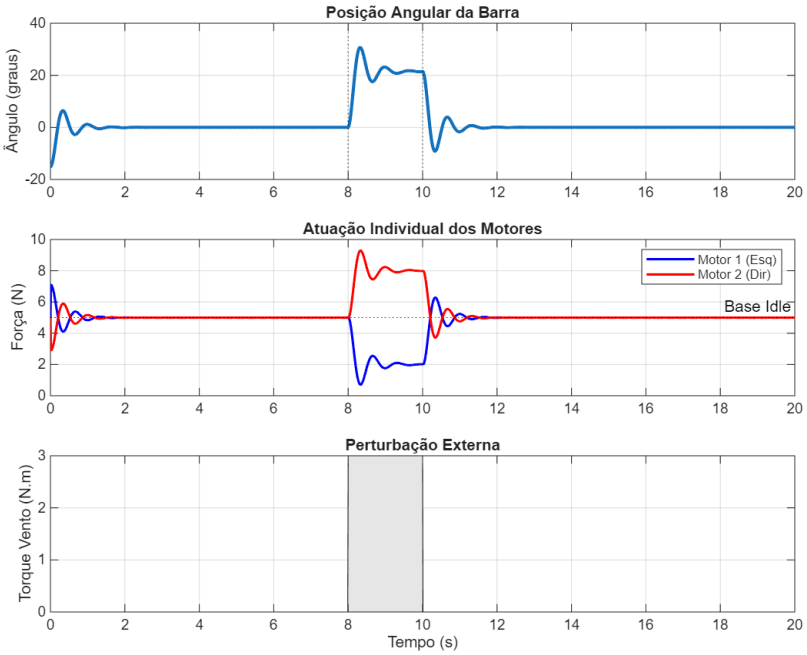
Os gráficos presentes na Figura 23 Representam a posição angular da barra, a atuação dos motores e a perturbação externa.

Para estes resultados, a posição inicial da barra foi -15°, o tempo de simulação foi de 20 segundos e o valor do ganho proporcional Kp, agora definido no nó, foi de 8. A frequencia de envio da mensagem para o nó de controle foi definida em 10 Hz.

Comparando os resultados podemos observar que, em amnbos os casos, o controlador foi capaz de atingir a estabilidade. Contudo no cenário em que o controlador é executado em um nó separado do modelo, a oscilação do angulo da barra e do torque dos motores é mais duradoura.

Essa diferença de resultados pode ser atribuída principalmente a latencia introduzida com a camada de comunicação ROS2.

Figura 23 – Resultados da Simulação



Fonte:Autor

5.6 Ajuste fino do controlador

Um dos objetivos do trabalho é auxiliar no desenvolvimento de drones, para isso o nó controlador pode ser ajustado para atender as necessidades do projeto, como estabilidade ou tempo de resposta.

O nó atual implementa um controlador proporcional, que pode ser ajustado alterando o valor do ganho proporcional Kp. Nós com diferentes tipos de controle podem ser implementados conforme a necessidade do projeto.

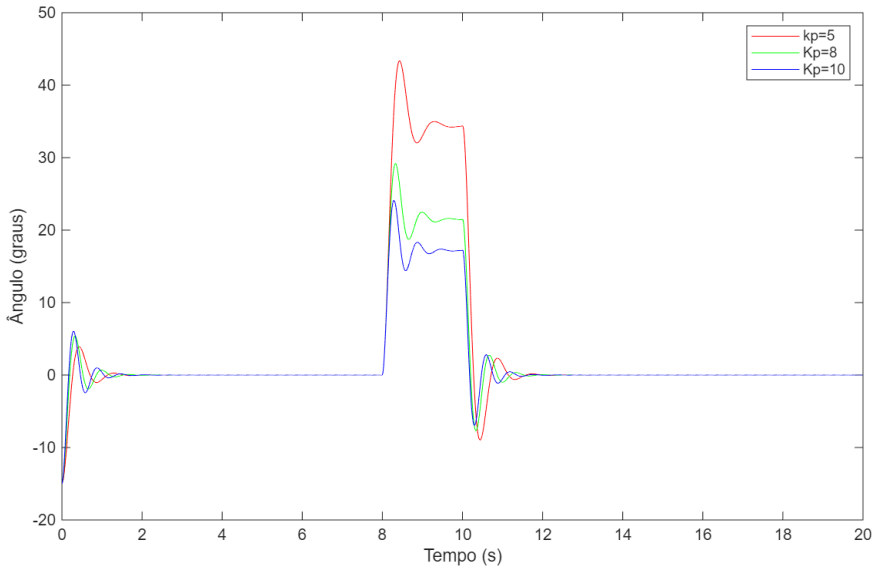
Com o ganho proporcional definido em 5, observa-se que as oscilações são menores e mais suaves, contudo o angulo atinge valores próximos a 45° mostrando que a ação do controlador foi suave.

Alterando o ganho proporcional para 8, o angulo da barra agora atinge valores menores, por volta de 30°, mostrando que a ação do controlador foi mais intensa.

Definindo o ganho proporcional em 10, obtém-se o cenário com o menor angulo da barra, em torno de 25°, aqui o controlador age com maior intensidade.

Os diferentes angulos dos respectivos valores de Kp estão na Figura 24

Figura 24 – Resultados da Simulação - diferentes Kp



Fonte:Autor

5.7 Simulação de diferentes frequencias de amostragem

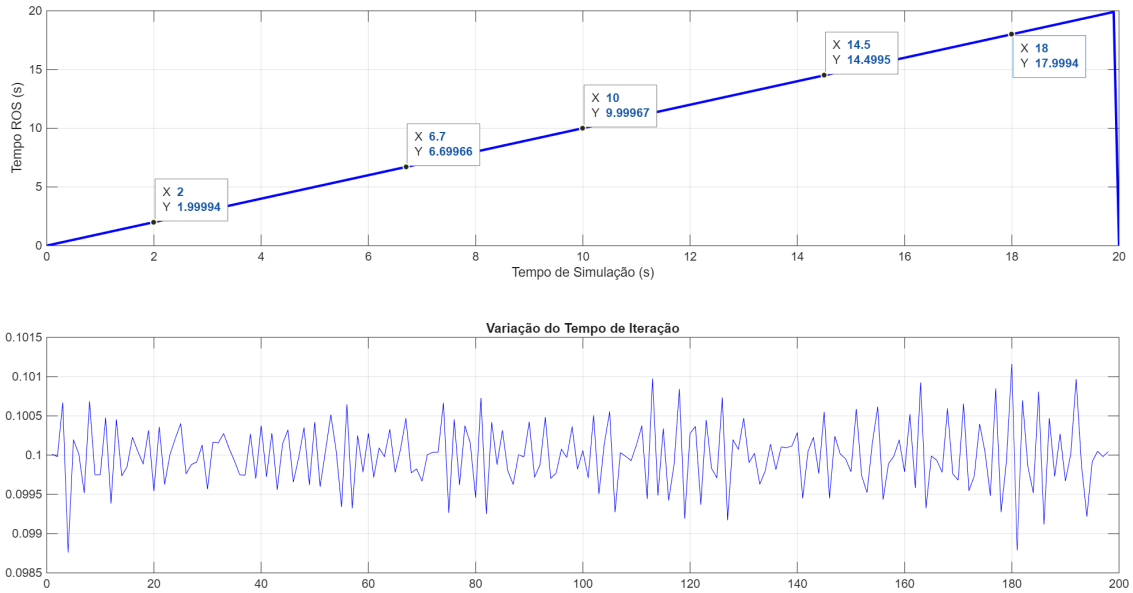
Esta seção apresenta e discute os resultados experimentais obtidos a partir dos ensaios realizados na plataforma embarcada. A análise divide-se em duas partes fundamentais: a

avaliação da estabilidade temporal (jitter) do ecossistema ROS2 em diferentes frequências de amostragem

5.7.1 10Hz

Para uma frequencia de 10Hz em uma simulação de 20 segundos, são esperadas 200 iterações com tempo máximo de 100 milissegundos por iteração. A Figura 25 ilustra o comportamento do sistema operando na frequência fundamental de 10Hz, cujo período nominal de iteração é de 100 milissegundos. No gráfico superior, observa-se uma relação perfeitamente linear entre o Tempo ROS e o Tempo de Simulação, com um erro acumulado praticamente desprezível ao atingir 17,9994 s no marco de 18 s de simulação. O gráfico inferior de variação do tempo confirma a excelente estabilidade do sistema nessa frequência, apresentando um jitter controlado que oscila estritamente entre 98,5 milissegundos e 101,2 milissegundos, o que representa uma variação percentual mínima frente ao período total e garante um comportamento altamente determinístico

Figura 25 – Resultados de Jitter - 10Hz

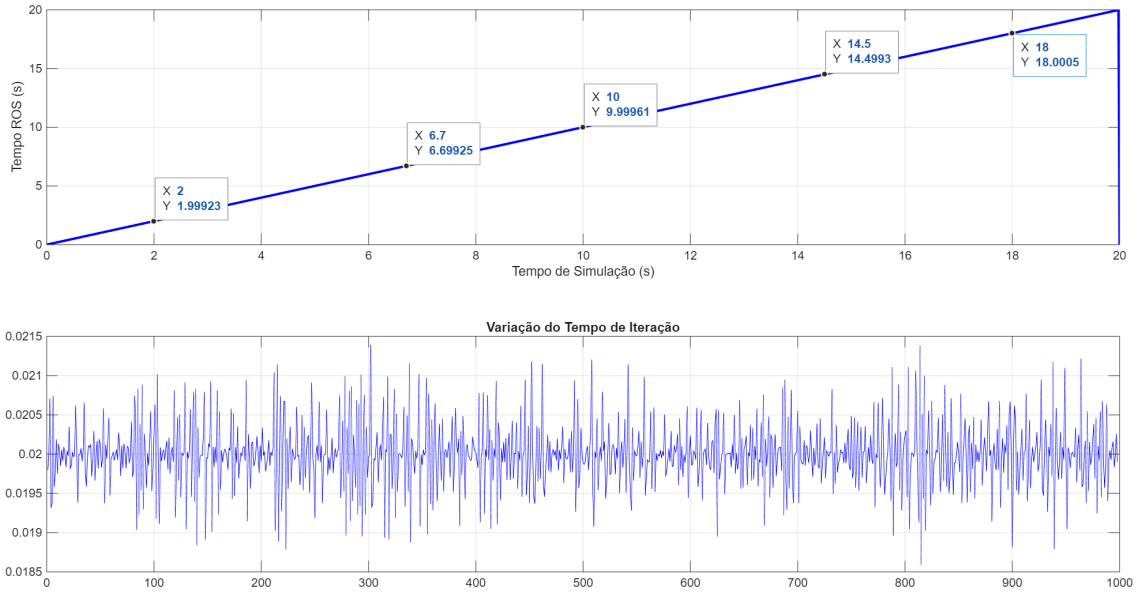


Fonte:Autor

5.7.2 50Hz

Para uma frequencia de 50Hz em uma simulação de 20 segundos, são esperadas 1000 iterações com tempo máximo de 20 milissegundos por iteração. A Figura 26 retrata o ensaio executado a 50 Hz, patamar no qual o período nominal de ciclo é reduzido para 20 milissegundos. A linearidade macroscópica do tempo (gráfico superior) permanece robusta, registrando 18,0005 s de Tempo ROS para os 18 s de simulação. Contudo, ao analisar a variação discreta das iterações no gráfico inferior, nota-se que a amplitude do ruído temporal passa a oscilar entre 18,5 milissegundos e 21,4 milissegundos; embora a magnitude absoluta do desvio mude pouco em relação ao teste anterior, ela começa a ganhar relevância proporcional frente ao período nominal de 20 milissegundos

Figura 26 – Resultados de Jitter - 50Hz

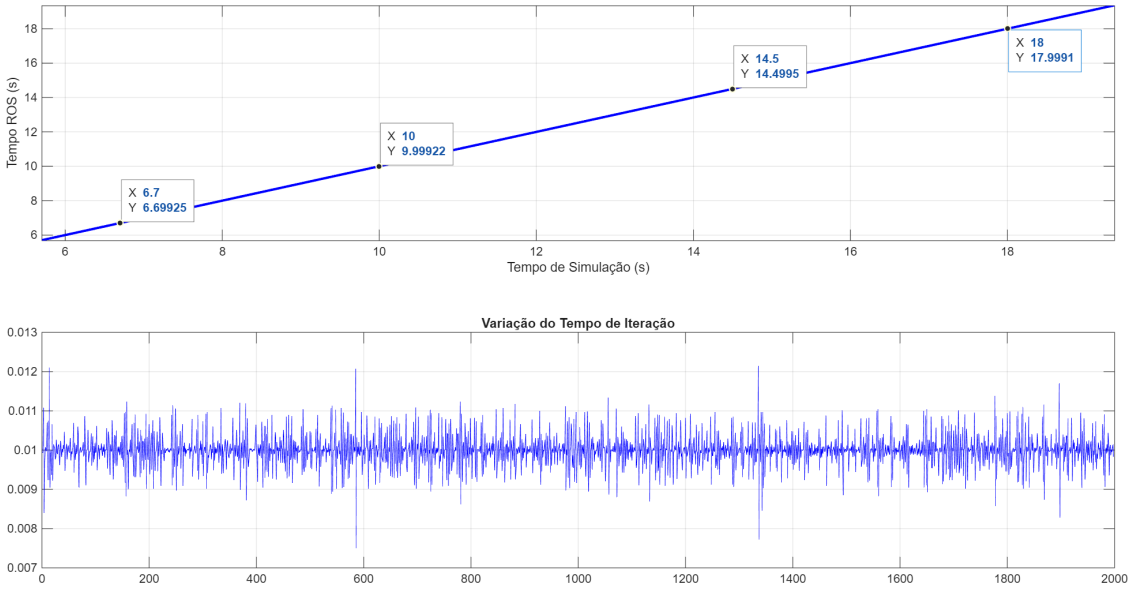


Fonte:Autor

5.7.3 100Hz

Para uma frequencia de 100Hz em uma simulação de 20 segundos, são esperadas 2000 iterações com tempo máximo de 10 milissegundos por iteração. A Figura 27 apresenta os dados coletados à taxa de amostragem de 100 Hz, cujo intervalo esperado entre as iterações é de 10 milissegundos. O subplot superior mantém a consistência da escala de tempo global, alcançando o valor de 17,9991 s no ponto de controle de 18 s. No entanto, o subplot inferior revela picos mais pronunciados e frequentes na variação do tempo de ciclo, com amplitudes que chegam a atingir a faixa de 7,5 milissegundos a 12,5 milissegundos, sinalizando que os atrasos de processamento internos e o escalonamento de threads do sistema operacional começam a impor flutuações perceptíveis na regularidade da malha.

Figura 27 – Resultados de Jitter - 100Hz

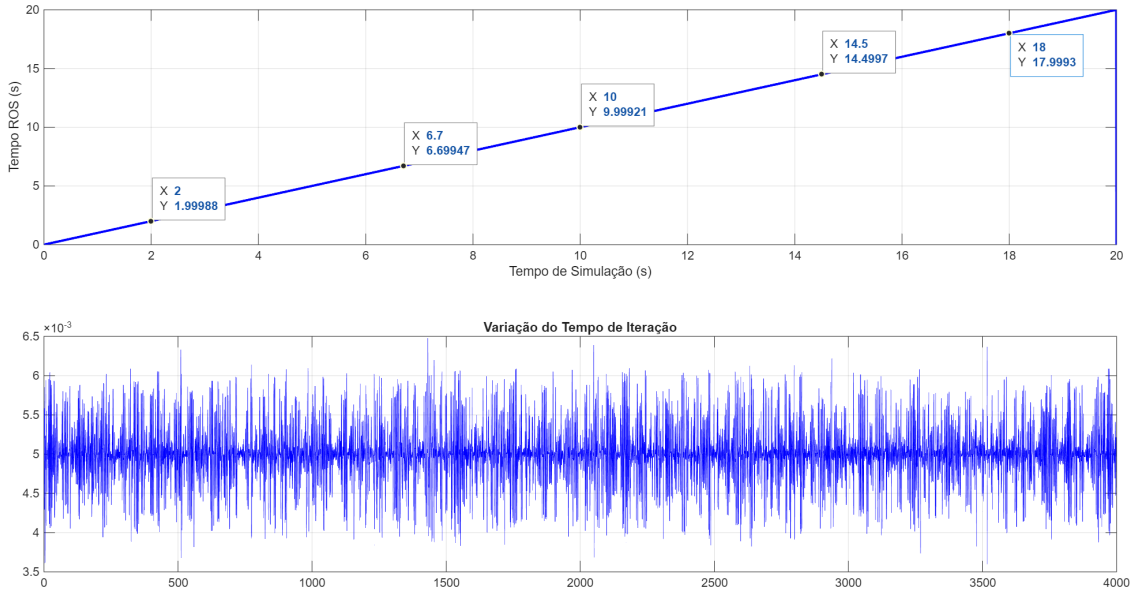


Fonte:Autor

5.7.4 200Hz

Para uma frequencia de 200Hz em uma simulação de 20 segundos, são esperadas 4000 iterações com tempo máximo de 5 milissegundos por iteração. A Figura 28 documenta a resposta temporal do nó de controle operando a 200 Hz, o que equivale a um período nominal de 5 milissegundos. A correspondência temporal de longo prazo no gráfico superior continua linear, marcando 17,9993 s aos 18 s. Por outro lado, a dispersão instantânea exibida no gráfico inferior se intensifica significativamente, criando uma banda densa de oscilações que varia de 3,5 milissegundos a 6,5 milissegundos, demonstrando que o erro dinâmico de tempo de ciclo passa a comprometer de forma mais agressiva a precisão do intervalo de amostragem.

Figura 28 – Resultados de Jitter - 200Hz

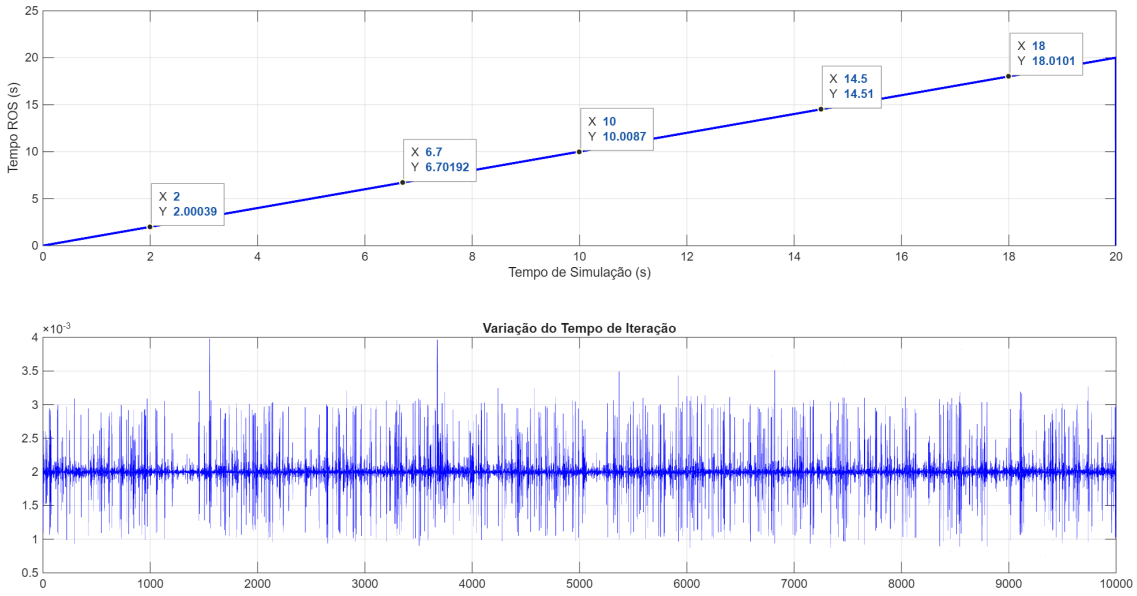


Fonte:Autor

5.7.5 500Hz

Para uma frequencia de 500Hz em uma simulação de 20 segundos, são esperadas 10000 iterações com tempo máximo de 2 milissegundos por iteração. A Figura 29 expõe os limites operacionais do sistema sob alta frequência a 500 Hz, onde o período de amostragem nominal é de apenas 2 milissegundos. Neste cenário crítico, o gráfico superior revela o surgimento de um desvio acumulado perceptível na linha de tendência, registrando um adiantamento no Tempo ROS que chega a 18,0101 s no instante de 18 s de simulação. Essa degradação fica evidente no gráfico inferior, onde o jitter sofre uma expansão severa, variando de 0,7 milissegundos até picos de 4 milissegundos, excedendo o próprio período nominal do loop.

Figura 29 – Resultados de Jitter - 500Hz



Fonte:Autor

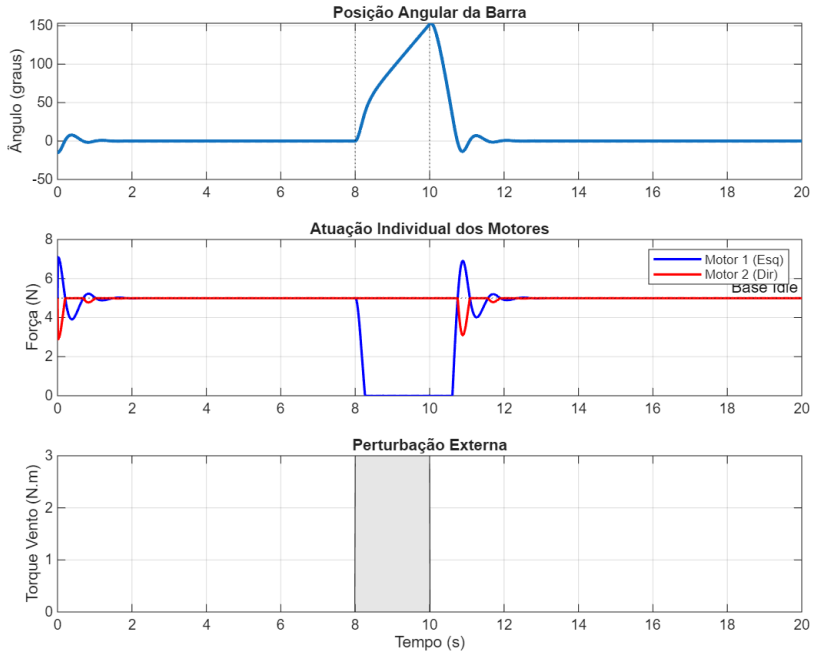
5.8 Simulação de erro

A simulação de erros e falhas é parte importante na formação de profissionais que pilotam e reparam drones. É possível simular falhas reais no modelo para explorar o comportamento da aeronave e a investigação da causa.

Uma das falhas a serem simuladas é a limitação do torque de um dos motores, o que afeta diretamente o controle. Na simulação a força máxima dos motores está definida em 20N, para representar um mau funcionamento, um dos motores pode ter sua força limitada em 5N e o ganho proporcional foi definido em 8. Os resultados dessa falha estão na Figura 30.

Comparando a Figura 23 com a Figura 30 é notório que a falha de um dos motores leva ao descontrole, levando a barra a atingir angulos proximos a 150°.

Figura 30 – Resultados da Simulação - falha no motor Direito



Fonte:Autor

Capítulo 6

Conclusão

O presente trabalho propôs e implementou um ambiente de desenvolvimento de drones integrando tecnologias de vanguarda, como o middleware ROS2, a aceleração de hardware em SOM (Kria KV260) e a simulação numérica em MATLAB. Através da metodologia de SIL e HIL, foi possível validar uma arquitetura que equilibra a complexidade algorítmica com o rigor do tempo real.

Os resultados demonstram que a utilização do ROS2 como camada de comunicação permite uma modularidade sem precedentes, facilitando a substituição de modelos matemáticos ou algoritmos de controle sem a necessidade de reestruturação completa do sistema. A integração do SOM mostrou-se eficaz para reduzir a latência de processamento, atacando diretamente a "lacuna de programabilidade"identificada na revisão bibliográfica. Além disso, o uso do MATLAB provou ser uma escolha acertada para a prototipagem rápida de malhas de controle, oferecendo uma fidelidade matemática superior aos simuladores puramente visuais.

Conclui-se que o ambiente desenvolvido cumpre o objetivo de ser uma plataforma segura, de baixo custo e altamente pedagógica, permitindo que pesquisadores e engenheiros testem cenários críticos e algoritmos avançados em um ambiente controlado antes da implementação em voo real.

6.0.1 Trabalhos futuros

Apesar dos avanços alcançados, o campo de drones e sistemas embarcados em tempo real oferece diversas oportunidades para expansão desta pesquisa. Como sugestões para trabalhos futuros, destacam-se:

- ❏ Implementação de Algoritmos de Inteligência Artificial em Hardware: Explorar as

unidades de processamento de IA presentes na plataforma Kria KV260 para realizar tarefas de visão computacional e desvio de obstáculos em tempo real diretamente na FPGA.

- ❏ Integração com Simuladores Fotorrealistas: Conectar a lógica de controle desenvolvida ao Unreal Engine ou AirSim, visando testar sensores visuais em ambientes virtuais de alta fidelidade visual, complementando a precisão matemática do MATLAB.

- ❏ Testes de Voo em Campo: Transpor o controlador validado no ambiente HIL para uma estrutura física de drone, comparando os dados de telemetria reais com os dados obtidos nas simulações para refinar ainda mais os modelos matemáticos utilizados.

Referências

AMD. **K26 PRODUCT BRIEF**. 2023. Disponível em: <https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/xilinx-k26-product-brief.pdf>.

_____. **Kria K26 System-on-Module Commercial**. 2023. Disponível em: <https://www.amd.com/en/products/system-on-modules/kria/k26/k26c-commercial.html>.

_____. **ds986-kv260-starter-kit**. 2024. Disponível em: <https://docs.amd.com/r/en-US/ds986-kv260-starter-kit>.

ANAC. **Quantidade de Cadastros - Drones**. 2024. Disponível em: <https://www.gov.br/anac/pt-br/assuntos/drones/quantidade-de-cadastros>.

ASIMOV, I. **Eu, Robô**. [S.l.]: Editora Aleph, 2015. (Série dos Robôs). ISBN 9788576572039.

BEARD, R. W.; MCLAIN, T. W. **Small Unmanned Aircraft: Theory and Practice**. [S.l.]: Princeton University Press, 2012.

CHAPRA, S. **Numerical Methods for Engineers**. McGraw-Hill Education, 2014. ISBN 9780077492168. Disponível em: <https://books.google.com.br/books?id=3GpzCgAAQBAJ>.

CHATTERJEE, P. K. et al. 20 - integrated circuits. In: MIDDLETON, W. M.; Van Valkenburg, M. E. (Ed.). **Reference Data for Engineers (Ninth Edition)**. Ninth edition. Woburn: Newnes, 2002. p. 20–1–20–113. ISBN 978-0-7506-7291-7. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780750672917500224>.

CHODNICKI, M. et al. Energy efficient uav flight control method in an environment with obstacles and gusts of wind. **Energies**, v. 15, n. 10, 2022. ISSN 1996-1073. Disponível em: <https://www.mdpi.com/1996-1073/15/10/3730>.

CORKE, P. **Robotics, Vision and Control: Fundamental Algorithms in MATLAB**. Springer, 2017. ISBN 9783319544120. Disponível em: <https://books.google.com.br/books?id=yO2hoAEACAAJ>.

CORNWELL, F. **Mecânica Vetorial para Engenheiros**. McGraw Hill Brasil, 2012. ISBN 9788580551440. Disponível em: <https://books.google.com.br/books?id=-yL3wwEACAAJ>.

COSTA, L. et al. **Integration of Robot Operating System and Ptolemy for Design of Real-Time Multi-robots Environments**. [S.l.: s.n.], 2017. 38–47 p.

DIGIKEY. $\mathbf{MFG}_S K - KV260 - G.2024.Disponívelem : <>$.

DILLARD, A. Real-time operational evaluations using advanced flight simulators. In: **17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No.98CH36267)**. [S.l.: s.n.], 1998. v. 1, p. E16/1–E16/7 vol.1.

FLIGHTGEAR. **FlightGear**. 2024. Disponível em: <https://www.flightgear.org/>.

FRANKLIN, G.; POWELL, J.; EMAMI-NAEINI, A. **Feedback Control of Dynamic Systems**. Pearson, 2014. (Always Learning). ISBN 9781292068909. Disponível em: <https://books.google.com.br/books?id=yO2hoAEACAAJ>.

GASPARETTO, L. S. A. A brief history of industrial robotics in the 20th century. v. 08, n. 01, p. 24–35, 2019.

GOEL, R. et al. Modeling, simulation and flight testing of an autonomous quadrotor. In: **IISc Centenary International Conference and Exhibition on Aerospace**. [S.l.: s.n.], 2009.

GOMES, N. et al. Análise da frequência radar na detecção de um rpa em função da rcs. In: . [S.l.: s.n.], 2022.

HARIDEVAN, A. D. et al. Ros2-gazebo simulator for drone applications. In: **2024 International Conference on Unmanned Aircraft Systems (ICUAS)**. [S.l.: s.n.], 2024. p. 1232–1238.

IRF. **World Robotics - Industrial Robots**. 2026. Disponível em: <https://ifr.org/wr-industrial-robots/>.

JAIN, N.; GUPTA, A. K.; MATHUR, P. Autonomous drone using ros for surveillance and 3d mapping using satellite map. In: GOYAL, D. et al. (Ed.). **Proceedings of the Second International Conference on Information Management and Machine Intelligence**. Singapore: Springer Singapore, 2021. p. 255–266. ISBN 978-981-15-9689-6.

JOSEPH, L.; JOHNY, A. Kick-starting robot programming using ros. In: _____. **Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy**. Berkeley, CA: Apress, 2022. p. 125–171. ISBN 978-1-4842-7750-8. Disponível em: <https://doi.org/10.1007/978-1-4842-7750-8_4>.

KöVARI, B. B.; EBEID, E. Mpdrone: Fpga-based platform for intelligent real-time autonomous drone operations. In: **2021 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)**. [S.l.: s.n.], 2021. p. 71–76.

_____. Mpdrone: Fpga-based platform for intelligent real-time autonomous drone operations. In: **2021 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)**. [S.l.: s.n.], 2021. p. 71–76.

MACENSKI, S. et al. Robot operating system 2: Design, architecture, and uses in the wild. **Science Robotics**, v. 7, n. 66, p. eabm6074, 2022. Disponível em: <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>.

_____. Robot operating system 2: Design, architecture, and uses in the wild. **Science robotics**, American Association for the Advancement of Science, v. 7, n. 66, p. eabm6074, 2022.

MALONE, B. **George Devol: A Life Devoted to Invention, and Robots**. 2011. Disponível em: <https://spectrum.ieee.org/george-devol-a-life-devoted-to-invention-and-robots>.

MARR, B. **The 4 Ds Of Robotization: Dull, Dirty, Dangerous And Dear**. 2017. Disponível em: <https://www.forbes.com/sites/bernardmarr/2017/10/16/the-4-ds-of-robotization-
>.

MARUYAMA, Y.; KATO, S.; AZUMI, T. Exploring the performance of ros2. In: **Proceedings of the 13th International Conference on Embedded Software**. New York, NY, USA: Association for Computing Machinery, 2016. ISBN 9781450344852. Disponível em: <https://doi.org/10.1145/2968478.2968502>.

MAYORAL-VILCHES, V. et al. Robotcore: An open architecture for hardware acceleration in ros 2. In: **2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. [S.l.: s.n.], 2022. p. 9692–9699.

_____. **RobotCore: An Open Architecture for Hardware Acceleration in ROS 2**. 2023. Disponível em: <https://arxiv.org/abs/2205.03929>.

MEGALINGAM, R. K. et al. Drone stability simulation using ros and gazebo. In: BIANCHINI, M. et al. (Ed.). **Advanced Computing and Intelligent Technologies**. Singapore: Springer Singapore, 2022. p. 131–143. ISBN 978-981-16-2164-2.

MEYER, J. et al. Comprehensive simulation of quadrotor uavs using ros and gazebo. In: NODA, I. et al. (Ed.). **Simulation, Modeling, and Programming for Autonomous Robots**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 400–411. ISBN 978-3-642-34327-8.

MIHALIč, F.; TRUNTIč, M.; HREN, A. Hardware-in-the-loop simulations: A historical overview of engineering challenges. **Electronics**, v. 11, n. 15, 2022. ISSN 2079-9292. Disponível em: <https://www.mdpi.com/2079-9292/11/15/2462>.

MORéAC, E. et al. Hardware-in-the-loop simulation with dynamic partial fpga reconfiguration applied to computer vision in ros-based uav. In: **2020 International Workshop on Rapid System Prototyping (RSP)**. [S.l.: s.n.], 2020. p. 1–7.

MTE. **CBO - Busca por Código**. 2024. Disponível em: <http://www.mtecbo.gov.br/cbosite/pages/pesquisas/BuscaPorCodigo.jsf>.

MUNARETTO, L. **Vants e Drones**. [S.l.]: Edição do Autor, 2015. 198 p. ISBN 9788591972906.

NIETO, R. et al. Open-source ros-based simulation for verification of fpga robotics applications. **Microprocessors and Microsystems**, v. 113, p. 105143, 2025. ISSN 0141-9331. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0141933125000110>.

NYBOE, F. F.; MALLE, N. H.; EBEID, E. Mpsoc4drones: An open framework for ros2, px4, and fpga integration. In: **2022 International Conference on Unmanned Aircraft Systems (ICUAS)**. [S.l.: s.n.], 2022. p. 1246–1255.

OETTERSHAGEN, P. et al. Design of small hand-launched solar-powered uavs: From concept study to a multi-day world endurance record flight. **Journal of Field Robotics**, v. 34, n. 7, p. 1352–1377, 2017. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21717>.

OGATA, K. **Modern Control Engineering**. [S.l.]: Pearson, 2021. v. 5th edition.

OPEN-ROBOTICS. **Distribuitons - ROS Wiki**. 2024. Disponível em: <http://wiki.ros.org/Distributions>.

_____. **Introduction - ROS Wiki**. 2024. Disponível em: <http://wiki.ros.org/Introduction>.

_____. **Robot Oterating System**. 2024. Disponível em: <https://www.ros.org/>.

PANERATI, J. et al. **Learning to Fly – a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control**. 2021. Disponível em: <https://arxiv.org/abs/2103.02142>.

PARDO-CASTELLOTE, G. Omg data-distribution service: architectural overview. In: **23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.** [S.l.: s.n.], 2003. p. 200–206.

PODLUBNE, A.; GöHRINGER, D. Fpga-ros: Methodology to augment the robot operating system with fpga designs. In: **2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)**. [S.l.: s.n.], 2019. p. 1–5.

PODLUBNE, A. et al. Model-based generation of hardware/software architectures with hybrid schedulers for robotics systems. **IEEE Transactions on Computers**, v. 73, n. 7, p. 1640–1654, 2024.

_____. Model-based approach for automatic generation of hardware architectures for robotics. **IEEE Access**, v. 9, p. 140921–140937, 2021.

QUIGLEY, M. et al. Ros: an open-source robot operating system. **ICRA workshop on open source software**, 2009.

RODIć, A.; MESTER, G. The modeling and simulation of an autonomous quad-rotor microcopter in a virtual outdoor scenario. **Acta Polytechnica Hungarica**, v. 8, p. 107–122, 01 2011.

SARKAR, S. et al. Real-time jitter measurements under ros2: the inverted pendulum case. 03 2021.

SHAH, S. et al. **AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles**. 2017. Disponível em: <https://arxiv.org/abs/1705.05065>.

TCHÁPEK, K.; MACHAC, V.; JOVANOVIC, A. **A fábrica de robôs**. [S.l.]: Hedra, 2022. (Coleção de Bolso). ISBN 9788577156870.

VALAVANIS, K. P.; VACHTSEVANOS, G. J. (Ed.). **Handbook of Unmanned Aerial Vehicles**. Dordrecht: Springer, 2015. ISBN 978-90-481-9706-4.

WAN, Z. et al. Robotic computing on fpgas: Current progress, research challenges, and opportunities. In: **2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)**. [S.l.: s.n.], 2022. p. 291–295.

_____. **A Survey of FPGA-Based Robotic Computing**. 2021. Disponível em: <https://arxiv.org/abs/2009.06034>.

WIENER, N.; HILL, D.; MITTER, S. **Cybernetics or Control and Communication in the Animal and the Machine, Reissue of the 1961 second edition**. [S.l.]: MIT Press, 2019. ISBN 9780262537841.

WYROBEK, K. **The Origin Story of ROS, the Linux of Robotics**. 2017. Disponível em: <https://spectrum.ieee.org/the-origin-story-of-ros-the-linux-of-robotics>.

WYROBEK, K. A. et al. Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In: **2008 IEEE International Conference on Robotics and Automation**. [S.l.: s.n.], 2008. p. 2165–2170.

YIN, Q. et al. Visual simulation system for quadrotor unmanned aerial vehicles. In: **Proceedings of the 30th Chinese Control Conference**. [S.l.: s.n.], 2011. p. 454–459.

ZHAO, X. et al. Design of a hand-launched solar-powered unmanned aerial vehicle (uav) system for plateau. **Applied Sciences**, v. 10, n. 4, 2020. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/10/4/1300>.

Apêndices

APÊNDICE A

Códigos Fonte

A.1 Código do nó Ping em C++

```cpp
#include <fstream>
#include <memory>
#include <string>
#include <cstdint>

#include "rclcpp/rclcpp.hpp"
#include "rtt_interface/msg/rtt.hpp"
#include "builtin_interfaces/msg/time.hpp"

using rtt_interface::msg::RTT;

static builtin_interfaces::msg::Time time_from_rclcpp(const rclcpp::Time
    & t)
{
  builtin_interfaces::msg::Time out;
  uint64_t ns = t.nanoseconds();
  out.sec = static_cast<int32_t>(ns / 1000000000ULL);
  out.nanosec = static_cast<uint32_t>(ns % 1000000000ULL);
  return out;
}

static int64_t time_to_ns(const builtin_interfaces::msg::Time & t)
{
  return static_cast<int64_t>(t.sec) * 1000000000LL + static_cast<
   int64_t>(t.nanosec);
}
```

```
25
26 class PingNode : public rclcpp::Node
27 {
28 public:
29   PingNode()
30   : Node("rtt_ping_cpp"), seq_(0)
31   {
32     double hz_default = 10.0;
33     this->declare_parameter<double>("hz", hz_default);
34     this->declare_parameter<std::string>("csv_path", "/tmp/rtt_cpp.csv")
    ;
35
36     double hz = this->get_parameter("hz").as_double();
37     csv_path_ = this->get_parameter("csv_path").as_string();
38
39     pub_ = this->create_publisher<RTT>("rtt_ping", 10);
40     sub_ = this->create_subscription<RTT>(
41       "rtt_pong", 10,
42       std::bind(&PingNode::on_reply, this, std::placeholders::_1));
43
44     timer_ = this->create_wall_timer(
45       std::chrono::duration<double>(1.0 / hz),
46       std::bind(&PingNode::tick, this));
47
48     open_csv(csv_path_);
49   }
50
51   ~PingNode()
52   {
53     if (csv_.is_open()) csv_.close();
54   }
55
56 private:
57   void tick()
58   {
59     RTT msg;
60     msg.id = seq_++;
61     msg.t_ping_send = time_from_rclcpp(this->get_clock()->now());
62
63     pub_->publish(msg);
64     RCLCPP_DEBUG(this->get_logger(), "Ping enviado id=%u", msg.id);
65   }
66
67   void on_reply(const RTT::SharedPtr msg)
68   {
69     // marca recep  o no ping
70     builtin_interfaces::msg::Time t_ping_recv = time_from_rclcpp(this->
```

```
    get_clock()->now());
71
72      int64_t t_send_ns = time_to_ns(msg->t_ping_send);
73      int64_t t_recv_ns = time_to_ns(t_ping_recv);
74      int64_t rtt_ns = t_recv_ns - t_send_ns;
75
76      int64_t t_pong_recv_ns = (msg->t_pong_recv.sec != 0 || msg->
    t_pong_recv.nanosec != 0)
77        ? time_to_ns(msg->t_pong_recv) : 0;
78      int64_t t_pong_send_ns = (msg->t_pong_send.sec != 0 || msg->
    t_pong_send.nanosec != 0)
79        ? time_to_ns(msg->t_pong_send) : 0;
80      int64_t pong_proc_ns = (t_pong_recv_ns && t_pong_send_ns) ? (
    t_pong_send_ns - t_pong_recv_ns) : 0;
81
82      RCLCPP_INFO(this->get_logger(),
83                  "id=%u rtt=%.3f ms pong_proc=%.3f ms",
84                  msg->id,
85                  static_cast<double>(rtt_ns) / 1e6,
86                  static_cast<double>(pong_proc_ns) / 1e6);
87
88      if (csv_.is_open()) {
89        csv_ << msg->id << ','
90             << t_send_ns << ','
91             << t_pong_recv_ns << ','
92             << t_pong_send_ns << ','
93             << t_recv_ns << ','
94             << rtt_ns << ','
95             << pong_proc_ns << '\n';
96        csv_.flush();
97      }
98    }
99
100   void open_csv(const std::string & path)
101   {
102     bool new_file = !std::ifstream(path).good();
103     csv_.open(path, std::ios::app);
104     if (new_file && csv_.is_open()) {
105       csv_ << "id,t_ping_send_ns,t_pong_recv_ns,t_pong_send_ns,
    t_ping_recv_ns,rtt_ns,pong_process_ns\n";
106       csv_.flush();
107     }
108   }
109
110   rclcpp::Publisher<RTT>::SharedPtr pub_;
111   rclcpp::Subscription<RTT>::SharedPtr sub_;
112   rclcpp::TimerBase::SharedPtr timer_;
```

```
  uint32_t seq_;
  std::ofstream csv_;
  std::string csv_path_;
};

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<PingNode>());
  rclcpp::shutdown();
  return 0;
}
```

A.2 Código do nó Pong em C++

```
#include <memory>
#include <cstdint>

#include "rclcpp/rclcpp.hpp"
#include "rtt_interface/msg/rtt.hpp"
#include "builtin_interfaces/msg/time.hpp"

using rtt_interface::msg::RTT;

static builtin_interfaces::msg::Time time_from_rclcpp(const rclcpp::Time
    & t)
{
  builtin_interfaces::msg::Time out;
  uint64_t ns = t.nanoseconds();
  out.sec = static_cast<int32_t>(ns / 1000000000ULL);
  out.nanosec = static_cast<uint32_t>(ns % 1000000000ULL);
  return out;
}

class PongNode : public rclcpp::Node
{
public:
  PongNode()
  : Node("rtt_pong_cpp")
  {
    pub_ = this->create_publisher<RTT>("rtt_pong", 10);
    sub_ = this->create_subscription<RTT>(
      "rtt_ping", 10,
      std::bind(&PongNode::on_ping, this, std::placeholders::_1));
  }

private:
```

```
  void on_ping(const RTT::SharedPtr in_msg)
  {
    RTT out = *in_msg;
    out.t_pong_recv = time_from_rclcpp(this->get_clock()->now());

    // (coloque aqui processamento se quiser medir o custo)
    out.t_pong_send = time_from_rclcpp(this->get_clock()->now());

    pub_->publish(out);
    RCLCPP_DEBUG(this->get_logger(), "Pong devolveu id=%u", out.id);
  }

  rclcpp::Publisher<RTT>::SharedPtr pub_;
  rclcpp::Subscription<RTT>::SharedPtr sub_;
};

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<PongNode>());
  rclcpp::shutdown();
  return 0;
}
```

A.3 Código do nó Pong com carga em C++

```
#include <memory>
#include <cstdint>

#include "rclcpp/rclcpp.hpp"
#include "rtt_interface/msg/rtt.hpp"
#include "builtin_interfaces/msg/time.hpp"

using rtt_interface::msg::RTT;

static builtin_interfaces::msg::Time time_from_rclcpp(const rclcpp::Time
    & t)
{
  builtin_interfaces::msg::Time out;
  uint64_t ns = t.nanoseconds();
  out.sec = static_cast<int32_t>(ns / 1000000000ULL);
  out.nanosec = static_cast<uint32_t>(ns % 1000000000ULL);
  return out;
}

class PongNode : public rclcpp::Node
{
```

```
21 public:
22   PongNode()
23   : Node("rtt_pong_cpp")
24   {
25     pub_ = this->create_publisher<RTT>("rtt_pong", 10);
26     sub_ = this->create_subscription<RTT>(
27       "rtt_ping", 10,
28       std::bind(&PongNode::on_ping, this, std::placeholders::_1));
29   }
30
31 private:
32   void on_ping(const RTT::SharedPtr in_msg)
33   {
34     RTT out = *in_msg;
35     out.t_pong_recv = time_from_rclcpp(this->get_clock()->now());
36
37     // (coloque aqui processamento se quiser medir o custo)
38   double mat[3][3] = {
39     {1.0, 2.0, 3.0},
40     {0.0, 1.0, 4.0},
41     {5.0, 6.0, 0.0}
42   };
43   double inv[3][3];
44
45   // 1. Calcular o determinante
46   double det = mat[0][0] * (mat[1][1] * mat[2][2] - mat[1][2] * mat
     [2][1]) -
47         mat[0][1] * (mat[1][0] * mat[2][2] - mat[1][2] * mat[2][0]) +
48         mat[0][2] * (mat[1][0] * mat[2][1] - mat[1][1] * mat[2][0]);
49
50   // 2. Calcular a matriz inversa se o determinante n o for zero
51   if (det != 0.0) {
52     double invdet = 1.0 / det;
53
54     inv[0][0] = (mat[1][1] * mat[2][2] - mat[1][2] * mat[2][1]) * invdet
     ;
55     inv[0][1] = (mat[0][2] * mat[2][1] - mat[0][1] * mat[2][2]) * invdet
     ;
56     inv[0][2] = (mat[0][1] * mat[1][2] - mat[0][2] * mat[1][1]) * invdet
     ;
57
58     inv[1][0] = (mat[1][2] * mat[2][0] - mat[1][0] * mat[2][2]) * invdet
     ;
59     inv[1][1] = (mat[0][0] * mat[2][2] - mat[0][2] * mat[2][0]) * invdet
     ;
60     inv[1][2] = (mat[0][2] * mat[1][0] - mat[0][0] * mat[1][2]) * invdet
     ;
```

```
61
62      inv[2][0] = (mat[1][0] * mat[2][1] - mat[1][1] * mat[2][0]) * invdet
        ;
63      inv[2][1] = (mat[0][1] * mat[2][0] - mat[0][0] * mat[2][1]) * invdet
        ;
64      inv[2][2] = (mat[0][0] * mat[1][1] - mat[0][1] * mat[1][0]) * invdet
        ;
65    }
66
67      out.t_pong_send = time_from_rclcpp(this->get_clock()->now());
68
69      pub_->publish(out);
70      RCLCPP_DEBUG(this->get_logger(), "Pong devolveu id=%u", out.id);
71    }
72
73    rclcpp::Publisher<RTT>::SharedPtr pub_;
74    rclcpp::Subscription<RTT>::SharedPtr sub_;
75  };
76
77  int main(int argc, char ** argv)
78  {
79    rclcpp::init(argc, argv);
80    rclcpp::spin(std::make_shared<PongNode>());
81    rclcpp::shutdown();
82    return 0;
83  }
```

A.4 Código do nó Ping em Python

```
1  #!/usr/bin/env python3
2  import csv
3  import os
4  import rclpy
5  from rclpy.node import Node
6  from rclpy.qos import QoSProfile
7  from rclpy.time import Time
8  from rtt_interface.msg import RTT
9
10 class PingNode(Node):
11     def __init__(self):
12         super().__init__('rtt_ping_py')
13         qos = QoSProfile(depth=10)
14
15         # par metros
16         self.declare_parameter('hz', 10.0)
17         self.declare_parameter('csv_path', '/tmp/rtt_py.csv')
18         self.hz = float(self.get_parameter('hz').value)
```

```
19          self.csv_path = self.get_parameter('csv_path').value
20
21          # pubs/subs
22          self.pub = self.create_publisher(RTT, 'rtt_ping', qos)
23          self.sub = self.create_subscription(RTT, 'rtt_pong', self.
    on_reply, qos)
24
25          # timer
26          self.timer = self.create_timer(1.0/self.hz, self.tick)
27          self.seq = 0
28
29          # CSV
30          self._csv_file = None
31          self._csv_writer = None
32          self._ensure_csv()
33
34      def _ensure_csv(self):
35          new_file = not os.path.exists(self.csv_path)
36          self._csv_file = open(self.csv_path, 'a', newline='')
37          self._csv_writer = csv.writer(self._csv_file)
38          if new_file:
39              self._csv_writer.writerow([
40                  'id',
41                  't_ping_send_ns',
42                  't_pong_recv_ns',
43                  't_pong_send_ns',
44                  't_ping_recv_ns',
45                  'rtt_ns',
46                  'pong_process_ns'
47              ])
48              self._csv_file.flush()
49
50      def tick(self):
51          msg = RTT()
52          msg.id = self.seq
53          msg.t_ping_send = self.get_clock().now().to_msg()
54          # os demais timestamps ser o preenchidos pelo pong e no retorno
55          self.pub.publish(msg)
56          self.seq += 1
57
58      def on_reply(self, msg: RTT):
59          # completar t_ping_recv e calcular m tricas
60          now = self.get_clock().now().to_msg()
61          msg.t_ping_recv = now
62
63          # converter para ns usando o clock do ping (RTT usa apenas
    t_ping_* locais)
```

```
64          def to_ns(t): return int(t.sec) * 1_000_000_000 + int(t.nanosec)
65          t_send = to_ns(msg.t_ping_send)
66          t_recv = to_ns(msg.t_ping_recv)
67          rtt = t_recv - t_send
68
69          # tempo de processamento no pong (se os rel gios forem os
    mesmos/mesma m quina)
70          t_pong_recv = to_ns(msg.t_pong_recv) if msg.t_pong_recv else 0
71          t_pong_send = to_ns(msg.t_pong_send) if msg.t_pong_send else 0
72          pong_proc = t_pong_send - t_pong_recv if t_pong_send and
    t_pong_recv else 0
73
74          self.get_logger().info(f"id={msg.id} rtt={rtt/1e6:.3f} ms
    pong_proc={pong_proc/1e6:.3f} ms")
75
76          self._csv_writer.writerow([
77              msg.id, t_send, t_pong_recv, t_pong_send, t_recv, rtt,
    pong_proc
78          ])
79          self._csv_file.flush()
80
81      def destroy_node(self):
82          if self._csv_file:
83              self._csv_file.close()
84          super().destroy_node()
85
86  def main(args=None):
87      rclpy.init(args=args)
88      node = PingNode()
89      try:
90          rclpy.spin(node)
91      finally:
92          node.destroy_node()
93          rclpy.shutdown()
```

A.5 Código do nó Pong em Python

```
1  #!/usr/bin/env python3
2  import rclpy
3  from rclpy.node import Node
4  from rclpy.qos import QoSProfile
5  from rtt_interface.msg import RTT
6
7  class PongNode(Node):
8      def __init__(self):
9          super().__init__('rtt_pong_py')
10         qos = QoSProfile(depth=10)
```

```
        self.pub = self.create_publisher(RTT, 'rtt_pong', qos)
        self.sub = self.create_subscription(RTT, 'rtt_ping', self.
    on_ping, qos)

    def on_ping(self, msg: RTT):
        # preenche t_pong_recv ao receber
        msg.t_pong_recv = self.get_clock().now().to_msg()
        # (eventual processamento...)
        # preenche t_pong_send imediatamente antes de devolver
        msg.t_pong_send = self.get_clock().now().to_msg()
        self.pub.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    node = PongNode()
    try:
        rclpy.spin(node)
    finally:
        node.destroy_node()
        rclpy.shutdown()
```

A.6 Código do nó Pong com carga em Python

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile
from rtt_interface.msg import RTT

class PongNode(Node):
    def __init__(self):
        super().__init__('rtt_pong_py')
        qos = QoSProfile(depth=10)
        self.pub = self.create_publisher(RTT, 'rtt_pong', qos)
        self.sub = self.create_subscription(RTT, 'rtt_ping', self.
    on_ping, qos)

    def on_ping(self, msg: RTT):
        # preenche t_pong_recv ao receber
        msg.t_pong_recv = self.get_clock().now().to_msg()
        # (eventual processamento...)
        mat = [
            [1.0, 2.0, 3.0],
            [0.0, 1.0, 4.0],
            [5.0, 6.0, 0.0]
        ]

```

```
24          # 1. Calcular o determinante
25          det = (mat[0][0] * (mat[1][1] * mat[2][2] - mat[1][2] * mat
    [2][1]) -
26                 mat[0][1] * (mat[1][0] * mat[2][2] - mat[1][2] * mat[2][0])
    +
27                 mat[0][2] * (mat[1][0] * mat[2][1] - mat[1][1] * mat[2][0]))
28
29          # 2. Calcular a matriz inversa
30          if det != 0.0:
31              invdet = 1.0 / det
32              inv = [
33                  [
34                      (mat[1][1] * mat[2][2] - mat[1][2] * mat[2][1]) *
    invdet,
35                      (mat[0][2] * mat[2][1] - mat[0][1] * mat[2][2]) *
    invdet,
36                      (mat[0][1] * mat[1][2] - mat[0][2] * mat[1][1]) *
    invdet
37                  ],
38                  [
39                      (mat[1][2] * mat[2][0] - mat[1][0] * mat[2][2]) *
    invdet,
40                      (mat[0][0] * mat[2][2] - mat[0][2] * mat[2][0]) *
    invdet,
41                      (mat[0][2] * mat[1][0] - mat[0][0] * mat[1][2]) *
    invdet
42                  ],
43                  [
44                      (mat[1][0] * mat[2][1] - mat[1][1] * mat[2][0]) *
    invdet,
45                      (mat[0][1] * mat[2][0] - mat[0][0] * mat[2][1]) *
    invdet,
46                      (mat[0][0] * mat[1][1] - mat[0][1] * mat[1][0]) *
    invdet
47                  ]
48              ]
49          # preenche t_pong_send imediatamente antes de devolver
50          msg.t_pong_send = self.get_clock().now().to_msg()
51          self.pub.publish(msg)
52
53  def main(args=None):
54      rclpy.init(args=args)
55      node = PongNode()
56      try:
57          rclpy.spin(node)
58      finally:
59          node.destroy_node()
```

```
60          rclpy.shutdown()
```

A.7 Código do nó Ping em Matlab

```
1  %% Criar n   ROS 2
2  node = ros2node("/matlab_ping_win");
3
4
5
6  %% Publisher e Subscriber
7  pingPub = ros2publisher(node,"/rtt_ping", "ml_latency_msgs/RTT" );
8
9  pongSub = ros2subscriber(node, "/rtt_pong", "ml_latency_msgs/RTT");
10
11 pause(1);  % garante descoberta DDS
12
13 rate = ros2rate(node, 10);
14
15 %% Medi   es
16 N = 100000;
17 RTT = zeros(N,1);
18
19 for k = 1:N
20     msg = ros2message("ml_latency_msgs/RTT");
21     msg.id = uint32(k);
22
23     % Timestamp local (Windows)
24     now = ros2time(node, "now");
25     msg.t_ping_send.sec = int32(now.sec);
26     msg.t_ping_send.nanosec = uint32(now.nanosec);
27
28     send(pingPub, msg);
29
30     % Receber pong
31     pongMsg = receive(pongSub, 10);
32
33     % Timestamp local (Windows)
34     now = ros2time(node, "now");
35     pongMsg.t_ping_recv.sec = int32(now.sec);
36     pongMsg.t_ping_recv.nanosec = uint32(now.nanosec);
37
38     % RTT total ( nico  valor confi vel)
39     RTT(k) = seconds(pongMsg.t_pong_recv) - seconds(pongMsg.t_ping_send)
       ;
40
41
42 end
```

```
43
44 %% Resultados
45 fprintf("RTT m dio : %.3f ms\n", mean(RTT)*1e3);
46 fprintf("RTT m x   : %.3f ms\n", max(RTT)*1e3);
47 fprintf("RTT m n   : %.3f ms\n", min(RTT)*1e3);
48 fprintf("Jitter    : %.3f ms\n", std(RTT)*1e3);
```

A.8 Código da planta com controlador integrado em Matlab

```
1 clear; clc; close all;
2
3 %% 1. Par metros F sicos do Sistema
4 m = 1.0;              % Massa da barra (kg)
5 L = 1.0;              % Comprimento total da barra (m)
6 d = L / 2;            % Dist ncia do piv  ao motor (bra o de alavanca)
7 I = (1/12) * m * L^2; % Momento de In rcia (kg*m^2)
8 b = 0.5;              % Coeficiente de atrito viscoso
9
10 %% 2. Par metros dos Motores
11 F_max = 20;          % For a m xima de cada motor (N)
12 F_base = 5.0;        % For a de "hover"/equil brio (N) - motores a
       rodar constante
13
14 %% 3. Par metros da Simula  o
15 dt = 0.01;
16 T_total = 20;
17 time = 0:dt:T_total;
18 N = length(time);
19
20 %% 4. Inicializa  o de Vetores
21 theta = zeros(1, N);       %  ngulo  (rad)
22 omega = zeros(1, N);       % Velocidade Angular (rad/s)
23 alpha = zeros(1, N);       % Acelera  o Angular (rad/s^2)
24 F1_hist = zeros(1, N);     % Hist rico Motor 1 (Esquerda)
25 F2_hist = zeros(1, N);     % Hist rico Motor 2 (Direita)
26 wind_torque = zeros(1, N);% Hist rico Vento
27
28 % Condi  es Iniciais
29 theta(1) = deg2rad(-15);  % Come a inclinado para baixo (-15 graus)
30
31 %% 5. Configura  o do Controlador
32 Kp = 10.0;                 % Ganho Proporcional
33 target_angle = 0;          %  ngulo  alvo (0 rad)
34
35 %% 6. Loop de Simula  o
```

```
for t = 1:N-1

    % --- A. Perturba  o (Vento) ---
    % Rajada forte entre 8s e 10s
    if time(t) >= 8 && time(t) <= 10
        disturb = 3.0; % Torque externo (N.m)
    else
        disturb = 0;
    end
    wind_torque(t) = disturb;

    % --- B. Controlador (C lculo do Torque Desejado) ---
    error = target_angle - theta(t);
    U_torque = Kp * error; % O controlador pede este torque

    % --- C. Aloca  o de Motores (Mixing) ---
    % Torque = (F1 * d) - (F2 * d)
    % U_torque = (F1 - F2) * d  =>  delta_F = U_torque / (2*d)

    delta_F = U_torque / (2 * d);

    % Aplica a varia  o sobre a for a base
    F1 = F_base + delta_F;
    F2 = F_base - delta_F;

    % Satura  o f sica (Motores n o d o for a negativa nem
    infinita)
    F1 = max(0, min(F1, 10));
    F2 = max(0, min(F2, F_max));

    % Guarda para os gr ficos
    F1_hist(t) = F1;
    F2_hist(t) = F2;

    % --- D. Din mica Real (F sica) ---
    % O torque que realmente acontece no sistema baseia-se nas for as
    reais
    % (considerando que os motores podem ter saturado)
    Tau_motores = (F1 * d) - (F2 * d);

    % Somat rio de Momentos: Motores + Vento - Atrito
    Tau_net = Tau_motores + disturb - (b * omega(t));

    accel = Tau_net / I;
    alpha(t) = accel;

    % Integra  o (Euler)
```

```
    omega(t+1) = omega(t) + accel * dt;
    theta(t+1) = theta(t) + omega(t+1) * dt;
end

% Ajuste final para gr ficos (repetir o  ltimo  valor para vetores
   terem mesmo tamanho)
F1_hist(N) = F1_hist(N-1);
F2_hist(N) = F2_hist(N-1);

%% 7. Visualiza  o
figure('Name', 'Simula  o Drone 2 Motores', 'Color', 'w', 'Position',
   [100 100 800 600]);

% Gr fico 1:  ngulo  da Barra
subplot(3,1,1);
plot(time, rad2deg(theta), 'LineWidth', 2);
hold on;
%yline(0, '--g', 'Est vel (0  )');
xline(8, ':k'); xline(10, ':k'); % Marca zona de vento
ylim([-20 40]);
ylabel(' ngulo  (graus)');
title('Posi  o Angular da Barra');
grid on;

% Gr fico 2: For as dos Motores Individualmente
subplot(3,1,2);
plot(time, F1_hist, 'b', 'LineWidth', 1.5); hold on;
plot(time, F2_hist, 'r', 'LineWidth', 1.5);
yline(F_base, ':k', 'Base Idle');
ylabel('For a (N)');
legend('Motor 1 (Esq)', 'Motor 2 (Dir)');
title('Atua  o Individual dos Motores');
grid on;

% Gr fico 3: Perturba  o
subplot(3,1,3);
area(time, wind_torque, 'FaceColor', [0.9 0.9 0.9], 'EdgeColor', 'k');
xlabel('Tempo (s)');
ylabel('Torque Vento (N.m)');
title('Perturba  o Externa');
grid on;
```

A.9 Código da planta em Matlab com controlador em nó ROS

```
clear; clc; close all;
```

```
2
3 %% 1. Configura  o do ROS 2 no Matlab
4 % Certifique-se de que o ambiente ROS 2 est  configurado corretamente
     no terminal/sistema
5 try
6     node = ros2node("/drone_plant_sim", 0);
7 catch
8     error('Falha ao criar n  ROS2. Verifique se o ROS Toolbox est
     instalado e configurado.');
9 end
10
11 % Publisher: Envia o estado ( ngulo ) para o C++
12 angle_pub = ros2publisher(node, "/drone/angle", "std_msgs/Float64");
13 angle_msg = ros2message("std_msgs/Float64");
14
15 % Subscriber: Recebe o comando (torque) do C++
16 torque_sub = ros2subscriber(node, "/drone/cmd_torque", "std_msgs/Float64
     ");
17
18 %% 2. Par metros F sicos (Idem anterior)
19 m = 1.0; L = 1.0; d = L/2;
20 I = (1/12) * m * L^2;
21 b = 0.5;
22 F_max = 20; F_base = 5.0;
23
24 % Controle de taxa de execu  o (para tentar rodar em tempo quase real)
25 rate = ros2rate(node, 500);
26
27 %% 3. Par metros de Tempo
28 dt = rate.DesiredPeriod;
29 T_total = 20;
30 time = 0:dt:T_total;
31 N = length(time);
32
33 % Inicializa  o de Vetores
34 theta = zeros(1, N);
35 omega = zeros(1, N);
36 control_torque_received = zeros(1, N); % Para guardar o que veio do ROS
37 F1_hist = zeros(1, N);    % Hist rico Motor 1 (Esquerda)
38 F2_hist = zeros(1, N);    % Hist rico Motor 2 (Direita)
39 wind_torque = zeros(1, N);% Hist rico Vento
40 timesec = zeros(1, N);
41 timenano = zeros(1, N);
42 tempo = zeros(1, N);
43
44
45 % Condi  o Inicial
```

```
46 theta(1) = deg2rad(-15);
47
48 disp('Simula  o iniciada. Aguardando controlador C++...');
49
50 reset(rate);
51 %% 4. Loop de Simula  o
52 for t = 1:N-1
53
54     % --- A. Enviar Estado para o ROS (Sensoriamento) ---
55     angle_msg.data = theta(t);
56     send(angle_pub, angle_msg);
57     %disp(angle_msg)
58
59     % --- B. Receber Comando do ROS (Controlador Externo) ---
60     % Verifica se h  nova mensagem de torque. Se n o houver, mant m a
       anterior (Zero-Order Hold)
61     % O 'msg' ser  a mensagem mais recente recebida pelo subscriber
62     cmd_msg = torque_sub.LatestMessage;
63
64     if isempty(cmd_msg)
65         u = 0; % Se ainda n  o conectou, torque zero
66     else
67         u = cmd_msg.data;
68     end
69     %timestamp
70     now = ros2time(node, "now");
71     timesec(t) = now.sec;
72     timenano(t) = now.nanosec;
73
74     control_torque_received(t) = u;
75
76     % --- C. Perturba  o (Vento) ---
77     if time(t) >= 8 && time(t) <= 10
78         disturb = 3.0;
79     else
80         disturb = 0;
81     end
82     wind_torque(t) = disturb;
83
84     % --- D. Aloca  o (Mixing) no lado da Planta ---
85     % O C++ mandou o torque U, o Matlab converte para for as f sicas
86     delta_F = u / (2 * d);
87     F1 = max(0, min(F_base + delta_F, F_max));
88     F2 = max(0, min(F_base - delta_F, F_max));
89
90     % Guarda para os gr ficos
91     F1_hist(t) = F1;
```

```
92      F2_hist(t) = F2;
93
94      % Torque f sico real
95      Tau_motores = (F1 * d) - (F2 * d);
96
97      % --- E. Din mica ---
98      Tau_net = Tau_motores + disturb - (b * omega(t));
99      accel = Tau_net / I;
100
101     omega(t+1) = omega(t) + accel * dt;
102     theta(t+1) = theta(t) + omega(t+1) * dt;
103
104     % --- F. Sincroniza  o de Tempo ---
105     % Isso faz o Matlab esperar para tentar manter o tempo real (
        essencial para ROS)
106     waitfor(rate);
107 end
108
109
110 for t = 1:N-1
111
112     tempo(t) = double(timesec(t) - timesec(1)) + double(timenano(t) -
        timenano(1)) * 1e-9;
113
114 end
115
116
117
118
119 % Ajuste final para gr ficos (repetir o  ltimo  valor para vetores
        terem mesmo tamanho)
120 F1_hist(N) = F1_hist(N-1);
121 F2_hist(N) = F2_hist(N-1);
122 fim = length(tempo)-1;
123
124 figure('Name', 'An lise de Tempo', 'Color', 'w', 'Position', [100 100
        800 600])
125
126 walltime = time;
127 timeros = tempo;
128
129 % Gr fico 1: Walltime vs Tempo de Simula  o
130 subplot(2,1,1);
131 plot(walltime, timeros, 'b', 'LineWidth', 2);
132 hold on;
133 ylabel('Tempo ROS (s)');
134 xlabel('Tempo de Simula  o (s)');
```

```
135 grid on;
136
137 % Gr fico 2: Varia  o do tempo de itera  o
138 delta_t = diff(tempo(2:fim));
139 subplot(2,1,2);
140 plot(delta_t, 'b')
141 title('Varia  o do Tempo de Itera  o')
142 grid on;
143
144 %% 5. Gr ficos (P s-processamento)
145 figure('Name', 'Simula  o Drone 2 Motores', 'Color', 'w', 'Position',
        [100 100 800 600]);
146
147 % Gr fico 1:  ngulo  da Barra
148 subplot(3,1,1);
149 plot(tempo(2:fim), rad2deg(theta(2:fim)), 'LineWidth', 2);
150 hold on;
151 %yline(0, '--g', 'Est vel (0 )');
152 xline(8, ':k'); xline(10, ':k'); % Marca zona de vento
153 ylabel(' ngulo  (graus)');
154 title('Posi  o Angular da Barra');
155 grid on;
156
157 % Gr fico 2: For as dos Motores Individualmente
158 subplot(3,1,2);
159 plot(tempo(2:fim), F1_hist(2:fim), 'b', 'LineWidth', 1.5); hold on;
160 plot(tempo(2:fim), F2_hist(2:fim), 'r', 'LineWidth', 1.5);
161 yline(F_base, ':k', 'Base Idle');
162 ylabel('For a (N)');
163 legend('Motor 1 (Esq)', 'Motor 2 (Dir)');
164 title('Atua  o Individual dos Motores');
165 grid on;
166
167 % Gr fico 3: Perturba  o
168 subplot(3,1,3);
169 area(tempo(2:fim), wind_torque(2:fim), 'FaceColor', [0.9 0.9 0.9], '
        EdgeColor', 'k');
170 xlabel('Tempo ROS(s)');
171 ylabel('Torque Vento (N.m)');
172 title('Perturba  o Externa');
173 grid on;
174
175 fprintf("tempo m dio : %.3f ms\n", mean(delta_t)*1e3);
176 fprintf("tempo m x   : %.3f ms\n", max(delta_t)*1e3);
177 fprintf("tempo m n   : %.3f ms\n", min(delta_t)*1e3);
178 fprintf("Jitter    : %.3f ms\n", std(delta_t)*1e3);
179
```

```
% Limpeza do n   ao fechar
clear node publisher subscriber;
```

A.10 Código do nó de controle em C++

```
// drone_controller.cpp
#include <chrono>
#include <memory>
#include <cmath>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/float64.hpp"

using std::placeholders::_1;

class DroneController : public rclcpp::Node
{
public:
  DroneController() : Node("drone_controller_node")
  {
    // 1. Publisher: Envia o torque calculado para o Matlab
    publisher_ = this->create_publisher<std_msgs::msg::Float64>("/drone/cmd_torque", 10);

    // 2. Subscriber: Recebe o  ngulo  atual do Matlab
    subscription_ = this->create_subscription<std_msgs::msg::Float64>(
      "/drone/angle", 10, std::bind(&DroneController::topic_callback, this, _1));

    // Par metros do Controlador
    kp_ = 8.0;          // Ganho Proporcional
    target_angle_ = 0.0; // Setpoint (0 radianos)

    RCLCPP_INFO(this->get_logger(), "Controlador Iniciado. Kp: %.2f", kp_);
  }

private:
  void topic_callback(const std_msgs::msg::Float64::SharedPtr msg)
  {
    // Recebe o  ngulo  atual (em radianos)
    double current_theta = msg->data;

    // --- L gica de Controle (Kp) ---
    double error = target_angle_ - current_theta;
    double control_u = kp_ * error;

```

```
40      // Cria a mensagem de comando
41      auto message = std_msgs::msg::Float64();
42      message.data = control_u;
43
44      // Publica o comando
45      publisher_->publish(message);
46
47      // Log para debug (opcional)
48      // RCLCPP_INFO(this->get_logger(), "Ang: %.3f | Erro: %.3f | Torque:
        %.3f", current_theta, error, control_u);
49    }
50
51    rclcpp::Subscription<std_msgs::msg::Float64>::SharedPtr subscription_;
52    rclcpp::Publisher<std_msgs::msg::Float64>::SharedPtr publisher_;
53    double kp_;
54    double target_angle_;
55  };
56
57  int main(int argc, char * argv[])
58  {
59    rclcpp::init(argc, argv);
60    rclcpp::spin(std::make_shared<DroneController>());
61    rclcpp::shutdown();
62    return 0;
63  }
```