INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO AMAZONAS - CAMPUS MANAUS DISTRITO INDUSTRIAL

CURSO BACHARELADO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

ICARO MATHEUS FONSECA UCHOA

CONTROLE PID BASEADO NO MÉTODO ZIEGLER-NICHOLS PARA SINCRONIZAÇÃO DE MOTORES EM PLATAFORMA ROBÓTICA DIFERENCIAL

MANAUS-AM

2025

ICARO MATHEUS FONSECA UCHOA

CONTROLE PID BASEADO NO MÉTODO ZIEGLER-NICHOLS PARA SINCRONIZAÇÃO DE MOTORES EM PLATAFORMA ROBÓTICA DIFERENCIAL

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Engenharia de Controle e Automação do Instituto Federal de Educação, Ciência e Tecnologia do Amazonas - Câmpus Manaus Distrito Industrial, como parte dos requisitos para obtenção grau de Bacharel em Engenharia de Controle e Automação.

Orientador: Prof. Msc. José Fábio de Lima Nascimento

MANAUS-AM

2025

Biblioteca do IFAM – Campus Manaus Distrito Industrial

U17c Uchoa, Icaro Matheus Fonseca.

Controle PID baseado no método Ziegler-Nichols para sincronização de motores em plataforma robótica diferencial/ Icaro Matheus Fonseca Uchoa. - Manaus, 2025.

69 f.: il. color.

Monografia (Graduação) – Instituto Federal de Educação, Ciência e Tecnologia do Amazonas, Curso de Engenharia de Controle e Automação, Campus Manaus Distrito Industrial, 2025

Orientador: Prof. Msc. José Fábio de Lima Nascimento.

1. Controle PID. 2. Método Ziegler-Nichols. 3. Robôs Móveis Diferenciais. 4. Robot Operating System (ROS). 5. Navegação Autônoma. I. NASCIMENTO, José Fábio de Lima (Orient.) II. Instituto Federal de Educação, Ciência e Tecnologia do Amazonas. III. Título.

CDD 629.89

Elaborada por Luciana Duarte Ferreira da Silva - CRB11/1010

ANEXO 7

ATA DE DEFESA PÚBLICA DO TRABALHO DE CONCLUSÃO DE CURSO

Aos 14 dias do mês de maio, de 2025, às 14:30 h, o(a) discente Icaro Matheus Fonseca Uchoa apresentou o seu Trabalho de Conclusão de Curso para avaliação da Banca Examinadora constituída pelos seguintes integrantes: Prof. Me. José Fábio de Lima Nascimento (docente-orientador), Prof. Dr. Adriano Bruno dos Santos Frutuoso (Membro 1) e Prof. Me. Renan Cavalcante Santos (Membro 2). A sessão pública de defesa foi aberta pelo(a) presidente da banca, que apresentou a Banca Examinadora e deu continuidade aos trabalhos, fazendo uma breve referência ao TCC, que tem como título Controle PID baseado no método Ziegler-Nichols para sincronização de motores em plataforma robótica diferencial. Na sequência, o(a) discente teve até 30 minutos para a comunicação oral de seu trabalho. Cada integrante da banca examinadora fez suas arguições após a defesa do mesmo. Ouvidas as explicações do(a) discente, a banca examinadora, reunida em caráter sigiloso, para proceder à avaliação final, deliberou e decidiu pela Aprovação com média final 8,5 (OITO, CINCO) do referido trabalho.

Foi dada ciência ao(à) discente que a versão final do trabalho deverá ser entregue até o dia 31/05/2025 com as devidas alterações sugeridas pela banca. Nada mais havendo a tratar, a sessão foi encerrada às 16 h 40 min, sendo lavrada a presente ata, que, uma vez aprovada, foi assinada por todos os membros da Banca Examinadora e pelo(a) discente.

Prof.(a) Orientador(a)/Presidente: José Fábio Lima Nascimento

Prof.(a) Avaliador 1: Adriano Bruno dos Santos Frutuoso

Prof.(a) Avaliador 2: Renan Cavalcante Santos

Discente: Icaro Matheus Fonseca Uchoa

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus por ter sido meu guia ao longo de toda esta trajetória, concedendo-me força e sabedoria nos momentos em que mais precisei. Sou grato ao meu pai Ernandenes e à minha mãe Adriana, que sempre estiveram ao meu lado, me apoiando e incentivando, especialmente nas fases mais desafiadoras, ajudando-me a superar cada barreira. Agradeço também aos meus colegas de turma — Giovane, Hylbert, Janderson, Lucas, Misael, Pedro Neto, Rebecca, Yure e tantos outros — que compartilharam comigo essa caminhada, sempre dispostos a colaborar e oferecer apoio nas dificuldades. Minha gratidão ao meu orientador, Prof. Msc. José Fábio de Lima Nascimento, por sua dedicação, paciência e sabedoria, sempre presente e pronto para auxiliar no que fosse necessário. Por fim, deixo meu sincero agradecimento ao IFAM, que foi fundamental na minha formação e por todo o conhecimento adquirido ao longo do curso.

RESUMO

Este trabalho apresenta o desenvolvimento e implementação de um sistema do controlador Proporcional-Integral-Derivativo (PID) baseado no método de Ziegler-Nichols para sincronização de motores em uma plataforma robótica diferencial de baixo custo. O projeto aborda desde a identificação experimental da planta dos motores até a implementação de navegação autônoma utilizando o *Robot Operating System* (ROS). Inicialmente, foi realizada a caracterização dinâmica dos motores de corrente contínua (DC) para obtenção dos parâmetros fundamentais (ganho estático, constante de tempo e tempo morto), permitindo a modelagem como sistemas de primeira ordem. A partir destes parâmetros, foi implementado o controlador PID com sintonia baseada no método de Ziegler-Nichols, seguido de um processo de refinamento experimental para a melhoria da resposta dinâmica. O sistema de controle foi integrado a uma estrutura completa de navegação autônoma utilizando ROS, incluindo odometria baseada em *encoder*, mapeamento com *Simultaneous Localization and Mapping* (SLAM) e navegação . A análise dos resultados incluiu estudos de desempenho temporal e espectral, demonstrando que a sincronização adequada dos motores é fundamental para a precisão na navegação. Os testes experimentais validaram a eficácia da abordagem, mostrando que a plataforma desenvolvida, com custo aproximado de R$ 1.500,00, alcança desempenho comparável a soluções comerciais mais caras em aplicações educacionais. A integração bem-sucedida entre controle de baixo nível e algoritmos de alto nível demonstra a viabilidade da criação de plataformas robóticas acessíveis para ensino e pesquisa em robótica móvel.

Palavras-chave: Controle PID; Método Ziegler-Nichols; Robôs Móveis Diferenciais; Robot Operating System (ROS); Navegação Autônoma.

ABSTRACT

This work presents the development and implementation of a Proportional-Integral-Derivative (PID) controller system based on the Ziegler-Nichols method for motor synchronization in a low-cost differential robotic platform. The project covers everything from the experimental identification of the motor plant to the implementation of autonomous navigation using the Robot Operating System (ROS). Initially, a dynamic characterization of the direct current (DC) motors was carried out to obtain fundamental parameters (static gain, time constant, and dead time), allowing modeling as first-order systems. Based on these parameters, a PID controller was implemented with tuning based on the Ziegler-Nichols method, followed by an experimental refinement process to improve dynamic response. The control system was integrated into a complete autonomous navigation structure using ROS, including encoder-based odometry, mapping with Simultaneous Localization and Mapping (SLAM), and navigation. The analysis of the results included both time-domain and spectral performance studies, demonstrating that proper motor synchronization is essential for accurate navigation. Experimental tests validated the effectiveness of the approach, showing that the developed platform, with an approximate cost of R$ 1,500.00, achieves performance comparable to more expensive commercial solutions in educational applications. The successful integration between low-level control and high-level algorithms demonstrates the feasibility of creating affordable robotic platforms for education and research in mobile robotics.

Keywords: PID Control; Ziegler-Nichols Method; Differential Mobile Robots; Robot Operating System (ROS); Autonomous Navigation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ROS	*Robot Operating System*
PWM	*Pulse Width Modulation*
RPM	*Revolutions Per Minute*
PPR	*Pulsos Por Revolução*
LiPo	*Lithium Polymer*
IMU	*Inertial Measurement Unit*
EKF	*Extended Kalman Filter*
MPC	*Model Predictive Control*
DC	*Direct current*
SLAM	*Simultaneous Localization and Mapping*
PID	*Proporcional Integral Derivativo*

SUMÁRIO

1 Introdução

A robótica móvel tem avançado significativamente nas últimas décadas, transformando diversos setores como indústria, saúde, educação e pesquisa. Entre as diversas configurações de robôs móveis, a plataforma com tração diferencial destaca-se por sua simplicidade mecânica combinada à versatilidade de movimentos. Essa configuração, embora estruturalmente simples, apresenta desafios consideráveis no que tange ao controle preciso de movimento, especialmente quando implementada com componentes de baixo custo.

1.1 Contextualização

O desenvolvimento da robótica educacional tem sido impulsionado pela disponibilidade de plataformas de código aberto como o Robot Operating System (ROS), que oferece um ecossistema robusto para implementação de algoritmos de controle, navegação e mapeamento. Neste contexto, o TurtleBot 3, desenvolvido pela ROBOTICS, emergiu como uma referência em plataformas robóticas para pesquisa e educação, integrando-se perfeitamente ao ecossistema ROS e oferecendo recursos como mapeamento, localização e navegação autônoma. No entanto, o acesso a plataformas como o TurtleBot 3 permanece restrito em muitos contextos educacionais devido ao seu custo elevado, particularmente em países em desenvolvimento. Esta realidade cria uma barreira significativa para estudantes e instituições com recursos limitados, dificultando o acesso ao aprendizado prático em robótica móvel e suas aplicações. A sincronização precisa dos motores em robôs diferenciais representa um desafio técnico fundamental, pois discrepâncias no comportamento dos atuadores resultam em desvios de trajetória e erros acumulativos na odometria. Em plataformas de baixo custo, esse problema é amplificado devido às variações nas características dos componentes utilizados, tornando ainda mais crítica a implementação de estratégias de controle eficientes.

A escolha de uma plataforma robótica diferencial como ambiente de validação para técnicas de controle PID não é meramente circunstancial, mas estratégica. Em primeiro lugar, estes sistemas incorporam naturalmente as complexidades de sistemas físicos reais - como não-linearidades, atritos variáveis, folgas mecânicas e assimetrias construtivas - oferecendo um cenário desafiador e representativo para a aplicação de técnicas de controle. Adicionalmente, a sincronização precisa dos motores em um robô diferencial representa um problema fundamental cujas implicações transcendem o mero controle de velocidade, afetando diretamente a precisão da odometria, a qualidade do mapeamento e a eficácia da navegação autônoma. Desta forma, o robô móvel constitui um ambiente de teste heurístico, onde a eficácia do controle implementado pode ser avaliada tanto no domínio do tempo quanto no domínio da frequência, complementada por resultados práticos.

1.2 Justificativa

O desenvolvimento de plataformas robóticas acessíveis financeiramente, sem comprometer funcionalidades essenciais, representa uma contribuição significativa para democratizar o acesso ao conhecimento em robótica. Ao propor uma alternativa de baixo custo ao TurtleBot 3, este trabalho busca não apenas reduzir a barreira financeira para aquisição de equipamentos, mas também demonstrar a viabilidade técnica de implementar sistemas de navegação autônoma em *hardware* mais acessível.

O método de controle *Ziegler-Nichols* aplicado ao ajuste de controladores PID oferece uma abordagem sistemática para lidar com as não-linearidades e discrepâncias típicas de motores de baixo custo. Ao contrário de outros métodos heurísticos amplamente utilizados, o método de Ziegler-Nichols oferece uma abordagem sistemática que pode ser aplicada mesmo sem o modelo matemático do sistema, facilitando a sintonia de controladores PID. Embora existam outros métodos com desempenho robusto, o Ziegler-Nichols destaca-se pela sua simplicidade e previsibilidade, especialmente em aplicações com motores de baixo custo e com recursos limitados de modelagem. Métodos baseados em otimização (como Ziegler-Nichols, Cohen-Coon ou algoritmos genéticos) têm diferentes graus de robustez e aplicabilidade, dependendo do sistema e da abordagem adotada (ÅSTRöM; HäGGLUND, 2006). A aplicação deste método para sincronização de motores em plataformas diferenciais representa uma contribuição técnica relevante, especialmente no contexto de sistemas de baixo custo. Adicionalmente, a implementação e validação de uma plataforma robótica diferencial completa, integrando o sistema de controle embarcado e algoritmos de alto nível para mapeamento e navegação, serve como estudo de caso para futuros desenvolvimentos na área, fornecendo parâmetros comparativos e lições aprendidas que podem beneficiar projetos similares.

A sincronização motora em robôs diferenciais representa um desafio técnico fundamental cuja resolução repercute em múltiplos níveis de funcionalidade do sistema. No nível mais básico, discrepâncias no comportamento dinâmico dos atuadores resultam em desvios de trajetória que comprometem a capacidade do robô de seguir caminhos retilíneos ou executar rotações precisas. Em um nível intermediário, estas mesmas discrepâncias introduzem erros sistemáticos na estimativa de posição por odometria, os quais se acumulam ao longo do tempo, degradando progressivamente a confiabilidade da localização. Finalmente, no nível de sistema, a imprecisão na localização compromete tanto a qualidade do mapeamento quanto a eficácia da navegação autônoma.

A implementação do controle PID com método Ziegler-Nichols para sincronização motora visa estabelecer as bases técnicas mensuráveis para o funcionamento confiável de um robô móvel diferencial. Especificamente, este trabalho busca reduzir os erros de velocidade entre os motores para valores inferiores a 5%, garantindo trajetórias retilíneas com desvio lateral absoluto máximo de 20 mm em percursos de 1 metro, medido por método físico direto com trena métrica (precisão ± 1 mm) em relação a uma linha de referência estabelecida no solo.

A análise espectral dos sinais de velocidade quantifica a eficácia dos controladores PID através da identificação e redução das oscilações características de motores dessintonizados. A utilização de componentes de baixo custo amplifica propositalmente os desafios de sincronização, criando um ambiente de teste que valida a robustez da estratégia de controle proposta.

Esta metodologia experimental permite demonstrar quantitativamente que controladores adequadamente sintonizados compensam as inconsistências típicas de hardware acessível, estabelecendo parâmetros de desempenho mensuráveis e transferíveis para implementações futuras em plataformas similares.

1.3 Objetivo Geral

Desenvolver e validar um sistema de controle PID baseado no método de Ziegler-Nichols para sincronização de motores em robôs diferenciais, utilizando uma plataforma robótica de baixo custo como ambiente de testes.

1.3.1 Objetivos Específicos

Os objetivos específicos deste trabalho são:

1. Modelar matematicamente os motores DC da plataforma robótica diferencial utilizando técnicas de identificação de sistemas, estabelecendo a relação entre *Pulse Width Modulation* (PWM) e *Revolutions Per Minute* (RPM);
2. Desenvolver e implementar um controlador PID para sincronização dos motores utilizando o método *Ziegler-Nichols* aplicado a motores DC modelados como sistemas de primeira ordem;
3. Implementar e validar experimentalmente a sincronização dos motores utilizando controlador PID com método *Ziegler-Nichols*, avaliando seu impacto direto na precisão da odometria;
4. Realizar análise da sincronização dos motores utilizando as seguintes técnicas:
    - Análise no domínio do tempo;
    - Análise espectral (FFT e coerência espectral);
5. Validar experimentalmente a plataforma em ambientes reais, avaliando:
    - Precisão no seguimento de trajetórias (desvio lateral máximo em percurso retilíneo);
    - Qualidade do mapeamento através da resolução espacial obtida, definição de contornos de obstáculos e cobertura da área explorada.

1.4 Estrutura do trabalho

Esta monografia está organizada em sete capítulos. O primeiro capítulo introduz o tema, apresentando a contextualização, justificativa e objetivos do trabalho. O segundo capítulo apresenta a fundamentação teórica necessária para compreensão do trabalho, abordando os princípios de robôs diferenciais, modelagem de motores DC, controle PID e método de *Ziegler-Nichols*, além de conceitos relacionados à análise espectral e sistemas ROS.

No terceiro capítulo, é detalhada a metodologia experimental empregada no desenvolvimento da plataforma, incluindo a caracterização dos motores, implementação do controlador PID, e os critérios de avaliação da sincronização. O quarto capítulo descreve o desenvolvimento da plataforma robótica utilizada como ambiente de validação, detalhando seus aspectos mecânicos, eletrônicos e computacionais.

O quinto capítulo apresenta os experimentos e testes realizados, com ênfase na análise espectral da sincronização motora. O sexto capítulo apresenta os resultados obtidos e as análises realizadas, incluindo o desempenho do controlador PID implementado e seu impacto na odometria e navegação. Por fim, o sétimo capítulo apresenta as conclusões do trabalho e sugere possíveis direções para trabalhos futuros na área.

2 Fundamentação Teórica

2.1 Robôs Móveis Diferenciais

2.1.1 Definição e Características

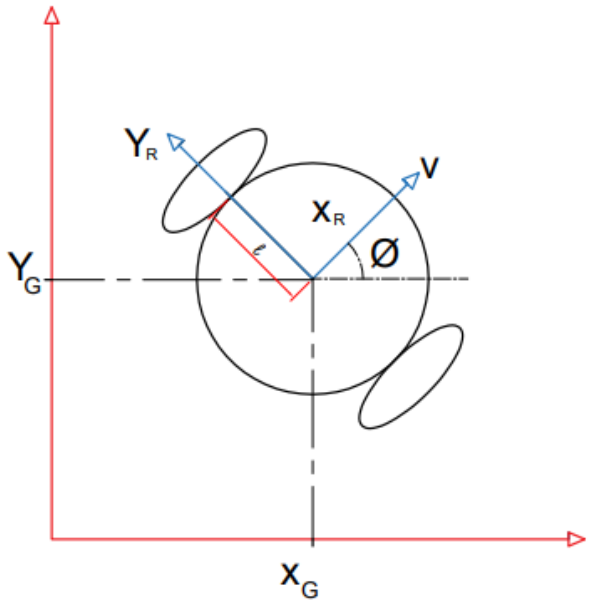
Robôs móveis diferenciais representam uma topologia fundamental na robótica móvel terrestre, caracterizados por um sistema de propulsão com dois atuadores independentes, geralmente rodas, que permitem movimentação e rotação através da diferença de velocidade entre esses atuadores (SIEGWART; NOURBAKHSH, 2004).

A configuração diferencial permite movimentos em diferentes direções pela variação assimétrica da velocidade das rodas:

- **Movimento retilíneo**: Velocidades iguais em ambos os motores.
- **Rotação no próprio eixo**: Rotação em sentidos opostos dos motores.
- **Curvas**: Velocidades diferentes entre os motores.

Robôs diferenciais são plataformas móveis dotadas de **duas rodas motorizadas independentes** e rodas passivas para estabilidade. Seu movimento é regido pela **cinemática diferencial**, que relaciona as velocidades das rodas ao movimento global do robô. A Figura 1 ilustra o modelo cinemático, que utiliza dois sistemas de coordenadas.

Figura 1 – Modelo cinemático do robô diferencial. (G): Referencial global; (R): Referencial local; L: Distância entre rodas; l: Raio das rodas.



Fonte: Próprio Autor (2025)

2.1.2 Modelagem Cinemática

Referenciais de Coordenadas

Referencial Global (G): Fixo no ambiente, define a posição absoluta do robô no plano cartesiano por meio das coordenadas (x, y, θ), onde:

- x, y: Posição do centro do robô no plano
- θ: Orientação do robô em relação ao eixo x global

Referencial Local (R): Fixo no robô, com origem no ponto médio entre as rodas. Utilizado para análise interna das velocidades:

- v: Velocidade linear do robô [m/s]
- ω: Velocidade angular do robô [rad/s]

Equações Cinemáticas

A modelagem cinemática de robôs diferenciais descreve como as velocidades das rodas influenciam o movimento do robô no espaço bidimensional, representado pelas coordenadas (x, y) de posição e pela orientação (θ). As equações fundamentais que governam esse movimento são:

$$v = \frac{l(\omega_{dir} + \omega_{esq})}{2}, \quad \text{(Velocidade linear)} \tag{2.1}$$

$$\omega = \frac{l(\omega_{dir} - \omega_{esq})}{L}, \quad \text{(Velocidade angular)} \tag{2.2}$$

Tabela 1 – Definição das variáveis do modelo cinemático

Símbolo	Descrição	Unidade
l	Raio das rodas motorizadas	$[m]$
L	Distância entre centros das rodas (trilha)	$[m]$
$\omega_{\mathbf{dir}}, \omega_{\mathbf{esq}}$	Velocidades angulares das rodas direita e esquerda	$[rad/s]$

Interpretação Física

- **Velocidade Linear** (v): Resulta da média das contribuições das duas rodas. Se $\omega_{\text{dir}} = \omega_{\text{esq}}$, o robô move-se em linha reta.
- **Velocidade Angular** (ω): Surge da diferença entre as velocidades das rodas:
  - $\omega_{\text{dir}} > \omega_{\text{esq}}$: Robô gira no sentido anti-horário
  - $\omega_{\text{dir}} < \omega_{\text{esq}}$: Robô gira no sentido horário

Aplicação no Projeto

Para o robô desenvolvido neste trabalho:

- $l = 0,033\,m$ (raio das rodas)
- $L = 0,2\,m$ (distância entre rodas)

Exemplo Numérico

Para $\omega_{\text{dir}} = 10\,rad/s$ e $\omega_{\text{esq}} = 8\,rad/s$:

$$v = \frac{0,033 \times (10+8)}{2} = 0,297\,m/s \tag{2.3}$$

$$\omega = \frac{0,033 \times (10-8)}{0,2} = 0,33\,rad/s \quad (\approx 18,9^\circ/s) \tag{2.4}$$

Esses parâmetros foram essenciais para a calibração do controlador PID e validação da odometria. A correta transformação entre os referenciais global e local é crítica para algoritmos de mapeamento (ex: gmapping) e navegação autônoma (ex: Navigation Stack).

2.1.3 Desafios Operacionais

Os robôs móveis diferenciais enfrentam desafios técnicos específicos que impactam diretamente sua capacidade de navegação autônoma e precisão operacional. O desafio fundamental reside na sincronização dos motores, uma vez que discrepâncias nas velocidades dos atuadores resultam em desvios de trajetória indesejados. Motores nominalmente idênticos apresentam variações em suas características dinâmicas devido a tolerâncias de fabricação, desgaste diferencial e variações térmicas, exigindo estratégias de controle específicas para manter a coordenação entre os atuadores. Estudos recentes demonstram que este problema persiste mesmo com tecnologias avançadas, onde a calibração da sincronização de velocidade continua sendo um requisito crítico para o funcionamento adequado dos sistemas (ARXIV, 2024b).

A compensação de variações nos componentes torna-se particularmente crítica em plataformas de baixo custo, onde componentes apresentam maior variabilidade em suas especificações operacionais. Estas variações incluem diferenças nas constantes de tempo dos motores, disparidades na resposta em frequência e inconsistências no torque de partida, introduzindo assimetrias no comportamento dinâmico da plataforma que necessitam de algoritmos de controle adaptativos. Pesquisas contemporâneas em 2024 confirmam que robôs de acionamento diferencial são amplamente utilizados em vários cenários graças ao seu princípio direto, mas as diferenças nos mecanismos de acionamento geralmente requerem modelagem cinemática específica quando o controle preciso é desejado (ARXIV, 2024a).

Os erros de odometria constituem outro desafio significativo, sendo a estimativa de posição baseada em odometria sujeita a erros acumulativos decorrentes de deslizamentos das

rodas, imprecisões na medição de velocidade e variações no diâmetro efetivo das rodas. Em robôs diferenciais, pequenas diferenças na velocidade dos motores geram erros sistemáticos na estimativa de orientação, que se propagam geometricamente ao longo do tempo, degradando progressivamente a qualidade da localização. Estudos recentes de 2025 identificaram que o deslizamento das rodas e o erro de odometria são fatores relevantes no desempenho geral do robô móvel com rodas na forma de desvio da trajetória desejada, navegação, tempo de viagem e consumo de energia orçado (WASET, 2025).

Portanto, a derrapagem durante movimentos de rotação e mudanças bruscas de direção pode induzir deslizamentos das rodas, especialmente em superfícies com baixo coeficiente de atrito. Este fenômeno compromete a relação entre os comandos de velocidade enviados aos motores e o movimento real da plataforma, introduzindo erros não modelados que afetam tanto a precisão da trajetória quanto a confiabilidade da odometria. As limitações impostas por superfícies não ideais, incluindo irregularidades, inclinações e variações no coeficiente de atração, apresentam desafios adicionais, onde diferenças na tração entre as rodas podem causar movimentos não intencionais, comprometendo a capacidade de controle preciso da plataforma e a qualidade dos dados sensoriais coletados. Análises atuais confirmam que atrito adequado é essencial para permitir que as rodas aderiam à superfície de forma eficaz, permitindo controle preciso sobre o movimento do robô, pois atrito insuficiente pode levar ao deslizamento das rodas, o que resultaria em imprecisões nos dados de odometria (ADDISON, 2024).

2.2 Motores de Corrente contínua (CC)

Os motores de corrente contínua (CC) operam com base em princípios eletromagnéticos, convertendo energia elétrica em movimento rotacional. Esse tipo de motor é amplamente utilizado em sistemas de robótica devido à sua simplicidade de controle e resposta rápida (ELECTRICITY-MAGNETISM, 2025; SERVO, 2023).

2.2.1 Princípios de Funcionamento

- **Estator**: Composto por ímãs permanentes (ou eletroímãs), é responsável por gerar o campo magnético estático necessário para a operação do motor.
- **Rotor (ou armadura)**: Conjunto de bobinas condutoras que, ao serem percorridas por corrente elétrica, interagem com o campo magnético do estator, gerando torque e promovendo o movimento rotacional.
- **Comutador**: Dispositivo mecânico que inverte a direção da corrente elétrica nas bobinas do rotor, garantindo a continuidade da rotação no mesmo sentido.
- **Escovas**: Conduzem a corrente elétrica da fonte de alimentação até o comutador, mantendo contato com ele durante a rotação do rotor.

Essa interação entre corrente e campo magnético gera uma força chamada *força de Lorentz* que faz o rotor girar. A direção e velocidade da rotação podem ser controladas facilmente pela variação da tensão e polaridade aplicadas ao motor, o que torna esse tipo de motor bastante utilizado em aplicações de robótica móvel.

2.2.2 Modelo Matemático

O comportamento dinâmico de um motor DC pode ser representado por um conjunto de equações diferenciais que descrevem suas características elétricas e mecânicas.

- **Equação elétrica:**

$$V(t) = R \cdot i(t) + L \cdot \frac{di(t)}{dt} + K_e \cdot \omega(t)$$

- **Equação mecânica:**

$$J \cdot \frac{d\omega(t)}{dt} + B \cdot \omega(t) = K_t \cdot i(t)$$

Onde:

- $V(t)$: tensão aplicada ao motor
- $i(t)$: corrente elétrica
- R: resistência do enrolamento
- L: indutância
- K_e: constante da força contra-eletromotriz
- $\omega(t)$: velocidade angular do eixo
- J: momento de inércia do rotor
- B: coeficiente de atrito viscoso
- K_t: constante de torque

Esse modelo permite compreender como o motor responde a diferentes condições de entrada e serve como base para o desenvolvimento de estratégias de controle que garantam um desempenho eficiente e estável.

2.2.3 Modelo de Primeira Ordem

Para simplificar a análise e facilitar o projeto de controladores, é comum representar a dinâmica do motor CC por meio de um modelo de primeira ordem. Na aplicação do controle PID em uma plataforma robótica diferencial, é essencial obter um modelo matemático que represente o comportamento dinâmico dos motores.

Segundo (NISE, 2017), essa aproximação pode ser expressa por uma função de transferência de primeira ordem, que relaciona a entrada de controle com a velocidade angular do motor. Para validar e ajustar este modelo teórico às características específicas dos motores utilizados, foi realizada uma caracterização empírica da planta neste trabalho. Essa caracterização consistiu na medição da velocidade de rotação (RPM) em função do sinal de controle PWM, conforme metodologia apresentada na seção 3.3.

$$G(s) = \frac{K_m}{\tau s + 1} \tag{2.5}$$

Onde:

- $G(s)$: função de transferência do sistema
- K_m: ganho estático do motor
- τ: constante de tempo

A simplificação é válida principalmente em sistemas nos quais os efeitos da indutância e do atrito são relativamente pequenos ou podem ser compensados via controle (OGATA, 2010). Em robôs móveis de baixo custo, essa modelagem reduzida permite um balanceamento eficiente entre precisão e complexidade, facilitando a implementação de controladores PID em plataformas com recursos computacionais limitados.

2.3 Controle de Sistemas

O controle de sistemas é uma área fundamental na engenharia, especialmente quando se trata de automação e robótica. Ele tem como objetivo ajustar o comportamento de um sistema dinâmico por meio de sinais elétricos, a fim de garantir que a saída siga uma determinada referência, mesmo diante de perturbações externas ou variações na planta. Para isso, diversas técnicas de controle podem ser utilizadas, sendo o controle PID uma das mais tradicionais e eficazes em aplicações práticas.

2.3.1 Controlador PID

O controlador Proporcional-Integral-Derivativo (PID) é uma das estratégias mais consagradas na área de controle automático, amplamente utilizada em sistemas industriais e embarcados devido à sua simplicidade, robustez e boa resposta dinâmica em uma variedade de aplicações (EMBARCADOS, 2024; SULFRAN AUTOMAÇÃO, 2023).

Esse tipo de controlador atua a partir do erro entre o valor desejado (referência) e o valor real do sistema (saída), aplicando uma correção baseada em três componentes:

- **Ação Proporcional (P):** Reage ao erro atual. Quanto maior o erro, maior será a ação de controle. Essa componente ajuda o sistema a reagir rapidamente a mudanças.
- **Ação Integral (I):** Considera o histórico do erro ao longo do tempo. Sua principal função é eliminar o erro em regime permanente, corrigindo desvios constantes que não seriam eliminados apenas pela ação proporcional.
- **Ação Derivativa (D):** Atua com base na taxa de variação (derivada) do erro, antecipando futuras mudanças no sistema. Essa ação proporciona amortecimento, reduzindo a possibilidade de oscilações e melhorando a estabilidade do sistema.

A equação geral que define o sinal de controle de um PID contínuo é expressa como:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)\,dt + K_d \cdot \frac{de(t)}{dt} \tag{2.6}$$

onde:

- $u(t)$ é o sinal de controle;
- $e(t)$ representa o erro, ou seja, a diferença entre o valor de referência e a saída do sistema;
- K_p, K_i e K_d são, respectivamente, os ganhos proporcional, integral e derivativo.

Na prática, a sintonia adequada desses três parâmetros é essencial para o bom desempenho do sistema. Um controlador mal ajustado pode provocar desde lentidão na resposta até instabilidades e oscilações indesejadas.

2.3.2 Método de Sintonia de Ziegler-Nichols

O método de Ziegler-Nichols é uma técnica clássica de sintonia de controladores PID baseada na resposta do sistema a um degrau (OGATA, 2010). O método conhecido como método da resposta ao degrau (ou método da curva de reação), é aplicável quando a planta pode ser modelada como um sistema de primeira ordem com atraso de transporte pequeno.

Para um sistema aproximado por:

$$G(s) = \frac{Ke^{-Ls}}{Ts+1}$$

Onde:

- K: ganho estático do sistema;

- L: tempo morto (atraso de transporte);
- T: constante de tempo do sistema;

Os parâmetros do PID podem ser ajustados segundo a Tabela 2:

Tabela 2 – Regra de sintonia de Ziegler-Nichols baseada na resposta ao degrau da planta (primeiro método)

Tipo de controlador	K_p	T_i	T_d
P	$\frac{T}{L}$	∞	0
PI	$0,9\frac{T}{L}$	$\frac{L}{0,3}$	0
PID	$1,2\frac{T}{L}$	$2L$	$0,5L$

Fonte: Ogata (2010, p. 524).

Com os valores de K_p, T_i e T_d definidos, os ganhos finais do controlador são:

$$K_i = \frac{K_p}{T_i}, \quad K_d = K_p \cdot T_d \tag{2.7}$$

Este método fornece uma sintonia inicial eficiente, adequada para sistemas com dinâmica relativamente simples, como motores DC de pequena escala em plataformas robóticas diferenciais. Os ajustes finos podem ser realizados posteriormente com base na resposta em malha fechada.

2.3.3 Desafios em Sistemas Não Modelados

A complexidade na modelagem de sistemas reais motiva abordagens empíricas (GUPTA; PADHY, 2020). Em muitos casos, como no controle de motores DC, o comportamento dinâmico do sistema é não-linear e sujeito a variações mecânicas, elétricas e ambientais, o que dificulta a obtenção de um modelo matemático preciso por meios puramente teóricos.

Dessa forma, adotou-se neste trabalho uma abordagem prática, fundamentada em testes experimentais. Inicialmente, foi realizada a coleta de dados relacionando o sinal de controle (PWM) com a resposta do sistema (RPM), permitindo a construção de um modelo linear. A partir deste modelo, estimou-se a planta do motor como uma função de primeira ordem, caracterizada por um ganho estático e uma constante de tempo extraída da resposta ao degrau.

Principais Desafios

Entre os principais desafios enfrentados, destacam-se:

- **Não linearidades do sistema:** A resposta dos motores DC pode variar com a carga, atrito e temperatura.

- **Incerteza nos parâmetros:** A estimativa do modelo depende da precisão dos dados coletados e da estabilidade do ambiente de teste.
- **Simplificações no modelo:** Ao assumir um sistema de primeira ordem, desprezam-se efeitos de ordem superior e não linearidades, que podem impactar o desempenho do controlador.
- **Ruído e limitações de sensores:** Medidas de RPM podem ser afetadas por ruído e resolução limitada, influenciando a regressão e os cálculos da constante de tempo.

A estimativa da planta com base em testes experimentais, como a relação entre PWM e RPM, é uma abordagem prática para contornar essas não-linearidades. No entanto, isso introduz incertezas que devem ser consideradas no projeto do controlador.

Abordagens Baseadas em Modelo vs. Empíricas

A Tabela 3 compara as estratégias de controle baseadas em modelo com as abordagens empíricas, destacando a importância da estimativa da planta em sistemas desconhecidos:

Tabela 3 – Comparação de estratégias de controle

Baseada em Modelo	**Empírica**
Requer equação diferencial completa	Utiliza dados experimentais
Ótimo para sistemas lineares	Não requer conhecimento do modelo matemático
Complexidade matemática elevada	Implementação simplificada

Incertezas Paramétricas

A falta de um modelo matemático preciso leva à necessidade de estimar parâmetros como:

- Ganho do sistema (K).
- Constante de tempo(τ).

Esses parâmetros foram estimados com base em testes de resposta ao degrau e análise da relação PWM-RPM, como mostrado nos resultados experimentais. No entanto, pequenas variações nas condições operacionais podem afetar a precisão dessas estimativas.

2.4 Sistema operacional ROS (Robot Operating System)

O ROS (Robot Operating System) é um middleware de código aberto amplamente utilizado no desenvolvimento de sistemas robóticos. Embora seu nome sugira tratar-se de um

sistema operacional completo, o ROS funciona como uma camada de software que opera sobre sistemas como o Linux, fornecendo ferramentas e bibliotecas que facilitam a criação de aplicações robóticas modulares (OPEN ROBOTICS, 2023).

2.4.1 Arquitetura e Funcionalidades

Entre suas principais funcionalidades, destacam-se:

- **Sistema de troca de mensagens entre nós (nodes):** permite que diferentes partes do sistema robótico se comuniquem de forma assíncrona, facilitando a integração de múltiplos sensores e atuadores.
- **Publicação e assinatura de tópicos (topics):** estrutura a comunicação, permitindo que dados sejam transmitidos e recebidos por diferentes nós de maneira organizada.
- **Chamadas de serviços (services):** possibilita interações síncronas entre nós, adequadas para operações que requerem uma resposta imediata.
- **Servidor de parâmetros (param server):** armazenamento centralizado de parâmetros que podem ser acessados por diversos nós, facilitando a configuração e ajuste do sistema.
- **Mensagens customizadas:** permite a definição de tipos de dados específicos para atender às necessidades particulares de cada aplicação.

Essa estrutura modular torna possível a comunicação eficiente entre sensores, atuadores e algoritmos, promovendo um desenvolvimento mais organizado e escalável.

2.4.2 Aplicações em Robótica Móvel

No desenvolvimento de robôs móveis diferenciais, como o apresentado neste trabalho, o ROS oferece uma base sólida para implementar funcionalidades críticas. Pacotes bem estabelecidos, como `gmapping` para mapeamento, `amcl` para localização e `move_base` dentro da `navigation stack` para planejamento de trajetória e navegação autônoma, são amplamente utilizados e suportados pela comunidade (TU DELFT, 2022).

Uma das principais vantagens do ROS 1 é a ampla comunidade de desenvolvedores e pesquisadores, o que garante uma vasta quantidade de recursos, tutoriais e suporte. Além disso, o sistema é compatível com simulações em ferramentas como Gazebo e RViz, facilitando o processo de testes antes da implementação no robô real (OPEN ROBOTICS, 2023).

Neste projeto, o uso do ROS Noetic é essencial para integrar os diversos módulos do robô, permitindo realizar testes em simulação, acompanhar dados em tempo real.

2.5 Odometria e Localização

2.5.1 Conceitos Fundamentais

A odometria de rodas, como técnica de localização para robôs móveis, enfrenta uma série de desafios que envolvem o uso de sensores, como encoders e LIDAR, para estimar a posição do robô em relação ao ambiente. Essa técnica depende de algoritmos de combinação de dados como o Filtro de Kalman Estendido (EKF) e o SLAM para combinar os dados dos sensores e oferecer estimativas precisas, facilitando a navegação autônoma do robô. Além disso, a classificação dos robôs móveis, que leva em conta as restrições de mobilidade de cada tipo, influencia diretamente a precisão da localização durante a navegação (ULLAH et al., 2024)

2.5.2 *Encoders* e Cálculo de Distância

No contexto desde trabalho, os *encoders* rotativos são usados para medir a quantidade de pulsos por rotação das rodas e, a partir disso, calcular a distância percorrida. A principal fórmula utilizada para converter os ticks do *encoder* em distância é dada por:

$$d_{\text{roda}} = \frac{2\pi l}{\text{PPR} \times \text{relação de engrenagem}} \tag{2.8}$$

onde: - l é o raio da roda, - **PPR** (Pulsos por Revolução) é o número de pulsos que o encoder gera por rotação completa da roda, - A **relação de engrenagem** Em termos matemáticos, se o motor gira uma vez, a relação de engrenagem determina quantas rotações a roda realizará. Em sistemas com engrenagens, a relação é tipicamente expressa como a razão entre o número de dentes de duas engrenagens envolvidas.

De acordo com (BENAMAR; BIDAUD; LE MENN, 2010), o modelo cinemático diferencial é essencial para avaliar o desempenho de robôs com rodas, abrangendo aspectos como mobilidade, transmissão de velocidade, singularidades e tração. Além disso, ele é crucial para técnicas de localização por odometria e controle de trajetória.

2.5.3 Limitações e Integração com ROS

Apesar da simplicidade de implementação e baixo custo, a odometria sofre com a propagação acumulativa de erros ao longo do tempo. Com isso, Técnicas de odometria que utilizam a velocidade das rodas são prejudicadas por acúmulo de erros devido ao escorregamento, comprometendo a precisão da navegação autônoma."(JÚNIOR, 2021).

No contexto deste trabalho, a odometria foi fundamental para o funcionamento de outros módulos, como o mapeamento e a navegação autônoma. O ROS oferece suporte nativo ao cálculo e à publicação da odometria por meio de pacotes como `diff_drive_controller`, que utilizam os dados dos encoders para estimar a posição do robô e publicar os valores em tópicos específicos, como `/odom`. Estes dados também são transformados em quadros de coordenadas, como `odom`

e `base_link`, por meio do pacote `robot_state_publisher`, permitindo monitoramento via RViz.

A confiabilidade da odometria é, portanto, um fator essencial no desempenho de um sistema robótico, especialmente em ambientes onde não há sinais externos confiáveis para correção da posição. Embora suas limitações sejam conhecidas, sua integração com outras fontes de dados permite criar sistemas de navegação mais robustos e precisos.

2.6 Mapeamento e Navegação

2.6.1 SLAM (Simultaneous Localization and Mapping)

O SLAM é uma técnica fundamental para a navegação autônoma de robôs. Como destacado por (DEBEUNNE; VIVET, 2020), "SLAM é o processo pelo qual um sistema robótico constrói um mapa do ambiente usando diferentes tipos de sensores enquanto estima sua própria posição no ambiente simultaneamente" (p. 2068). Essa abordagem permite que robôs operem em ambientes desconhecidos de forma eficiente.

A Localização e Mapeamento Simultâneos busca resolver dois problemas: enquanto o robô mapeia o ambiente, ele também precisa se localizar dentro desse mapa. O robô usa sensores (como LIDAR, câmeras ou radares) para perceber o ambiente e, ao mesmo tempo, determina sua posição relativa.

Pacotes ROS para SLAM

O ROS oferece diversos pacotes que implementam algoritmos SLAM:

- **gmapping**: baseado em filtros de partículas, ideal para sensores LIDAR 2D (GRISETTI; STACHNISS; BURGARD, 2007);
- **hector_slam**: indicado para LIDARs de alta frequência, sem necessidade de odometria (KOHLBRECHER et al., 2011);
- **cartographer**: desenvolvido pelo Google, suporta SLAM 2D e 3D com excelente precisão (HESS et al., 2016).

Nesse sentido, o pacote gmapping foi utilizado exclusivamente para gerar o mapa do ambiente, sem aplicação direta em tarefas de navegação autônoma. O mapa gerado posteriormente pode ser utilizado por outros sistemas (como amcl) para localização ou planejamento de trajetórias.

2.6.2 Algoritmo Gmapping

Conforme destacado por (LI; SCHULZE; KALAVADIA, 2024), "O GMapping foca principalmente no mapeamento baseado em grade bidimensional. Ele fornece uma implementa-

ção mais simples e é bem adequado para cenários onde um mapa bidimensional é suficiente. O objetivo do SLAM baseado em gráfico, por outro lado, é construir uma representação gráfica do ambiente" (p. 2867).

O algoritmo Gmapping trabalha com o filtro de partículas (também conhecido como Monte Carlo Localization, MCL), que é uma abordagem probabilística que ajuda o robô a lidar com incertezas tanto no movimento quanto nas medições dos sensores.

Integração LIDAR e Encoders

O Gmapping foi combinado com a odometria dos encoders dos motores e os dados do LIDAR para otimizar a localização e o mapeamento. O LIDAR fornece informações precisas sobre as distâncias dos obstáculos ao redor do robô, enquanto os encoders dos motores ajudam a estimar o movimento do robô, fornecendo dados sobre a distância percorrida e a direção.

Desafios do SLAM

Um exemplo de desafio é o loop closure, que ocorre quando o robô retorna a um local já visitado e falha em reconhecer esse local devido a erros acumulados. O Gmapping inclui técnicas para corrigir esse tipo de erro, ajustando o mapa enquanto o robô continua a explorar o ambiente.

Neste trabalho, o uso do Gmapping foi fundamental para gerar mapas 2D em tempo real. Esse tipo de mapeamento é útil em diversas aplicações, como navegação autônoma em ambientes desconhecidos, exploração de armazéns automatizados e veículos autônomos. É importante deixar claro o desafio de integração desse pacote, pois ele depende diretamente da qualidade das leituras relacionadas a odometria para garantir um bom resultado no mapeamento.

2.6.3 Navigation Stack

A locomoção autônoma de robôs móveis representa um elemento essencial no campo da robótica, possibilitando que esses dispositivos se desloquem de maneira independente por espaços não mapeados ou desconhecidos.

"O ROS Navigation Stack é um conjunto de pacotes que fornece uma solução completa de navegação para robôs móveis em ROS. A Pilha de Navegação inclui vários componentes, como um servidor de mapas, localização, planejamento de caminho e prevenção de obstáculos. A Pilha de Navegação usa dados de sensores de várias fontes, como telêmetros a laser, câmeras e odometria, para localizar a posição do robô e planejar um caminho para atingir seu objetivo." (ALAM, 2023).

3 METODOLOGIA EXPERIMENTAL

A metodologia desenvolvida para este projeto de controle PID de sincronização de motores em plataforma robótica diferencial envolveu a seleção cuidadosa de componentes e equipamentos que permitissem uma implementação de baixo custo e alta eficiência. A escolha dos materiais foi fundamentada em critérios técnicos, econômicos e de desempenho, visando atender aos requisitos do projeto de forma otimizada.

Além de descrever detalhadamente o procedimento experimental utilizado para a identificação da planta dos motores DC e a sintonia do controlador PID, de forma a garantir a reprodutibilidade dos resultados e a conformidade com os preceitos metodológicos exigidos. O procedimento abrange desde a excitação dos motores até a obtenção dos parâmetros característicos e a implementação dos algoritmos de controle, permitindo uma compreensão abrangente dos aspectos dinâmicos envolvidos no sistema.

3.1 Materiais e Equipamentos

3.1.1 Componentes Utilizados

A Tabela 4 apresenta a relação detalhada dos componentes utilizados no desenvolvimento da plataforma robótica, incluindo suas especificações técnicas, custos e justificativas de seleção.

Tabela 4 – Lista de Componentes, Custos e Especificações

Componente	**Qtd.**	**Valor Unit. (R$)**	**Valor Total (R$)**	**Especificações**
Motor CH N20-3+ encoder e redução 380/1	2	22,23	44,46	Motor DC com encoder, redução 380/1
LiDAR YDLidar X2 Pro	1	290,00	290,00	Sensor LiDAR 2D para mapeamento
Arduino Mega 2560	1	130,00	130,00	Microcontrolador para controle de baixo nível
Raspberry Pi 4B	1	600,00	600,00	Computador embarcado para processamento
Shield Motor Drive L293D	1	28,00	28,00	Driver de motores
Fiação	50	0,20	10,00	Cabos diversos
Roda Pneu para Chassi Robô	2	6,50	13,00	Rodas para movimentação
PowerBank	1	150,00	150,00	10000mAh 3.1A
Rodízio Giratório 25x31mm	1	12,00	12,00	Rodízio de apoio
		CUSTO TOTAL	**R$1.277,46**	

3.1.2 Critérios de Seleção dos Componentes

A seleção dos componentes foi baseada em critérios técnicos específicos, considerando compatibilidade, desempenho e custo-benefício para o projeto de controle PID em plataforma robótica diferencial.

3.2 Microcontrolador e Processamento

3.2.1 Microcontrolador Arduino Mega 2560

O Arduino Mega 2560 foi selecionado primariamente pela compatibilidade com o ROS Noetic através do pacote rosserial. Este pacote permite a integração entre a camada lógica e a camada computacional:

- Comunicação eficiente entre o Arduino e o computador embarcado
- Implementação de nós ROS no microcontrolador
- Controle de baixo nível dos atuadores
- Grande número de pinos para interfaces diversas

Além da facilidade de implementação do controlador PID por meio da biblioteca PID_v1 que representa um diferencial significativo na escolha do Arduino Mega 2560 para este projeto. Suas principais características e vantagens incluem:

- Implementação simplificada do algoritmo de controle PID
- Suporte a modo automático e manual
- Capacidade de ajuste dinâmico dos parâmetros K_p, K_i e K_d
- Tratamento de problemas como *anti-windup*
- Configurações flexíveis de direção de controle (direto/reverso)
- Baixo *overhead* computacional
- Fácil integração com outros componentes e bibliotecas do Arduino
- Método computacionalmente eficiente de cálculo do erro
- Possibilidade de configurar tempo de amostragem personalizado
- Suporte a diferentes modos de computação do termo integral e derivativo

A biblioteca simplifica significativamente a implementação do controle PID, permitindo focar nos aspectos específicos do projeto de robótica móvel, em vez de desenvolver toda a estrutura de controle do zero.

3.2.2 Raspberry Pi 4B

A seleção do Raspberry Pi 4B atendeu requisitos de mobilidade e processamento:

- Arquitetura quad-core (64-bit @ 1.5GHz)
- ROS Noetic pré-instalado
- Eficiência energética: 15W TDP

- Memoria RAM: 4GB
- Memoria ROM: 32GB cartão SD
- Suporte a algoritmos de navegação SLAM

3.3 Componentes de Atuação e Controle

3.3.1 Shield Motor Drive L293D

A escolha do shield L293D baseou-se em características críticas para controle de motores:

- Proteção com diodos de roda livre integrados
- Capacidade de controlar múltiplos motores em diferentes direções
- Proteção contra correntes de pico
- Compatibilidade com sinais PWM para controle de velocidade
- Circuitos de proteção contra curto-circuito

3.3.2 Sistema de Alimentação

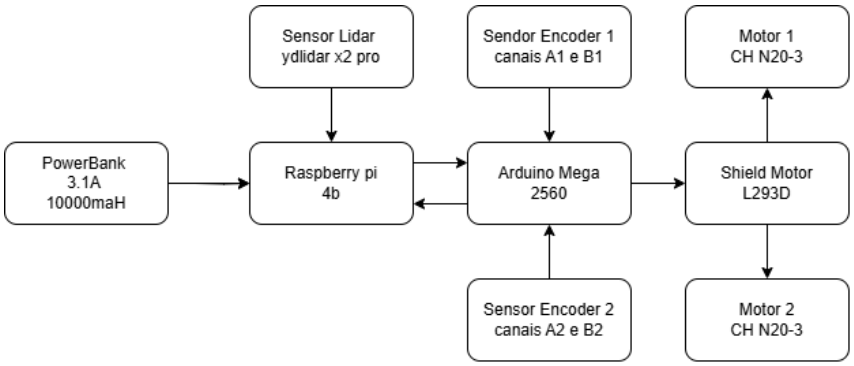
Características que justificaram a escolha do PowerBank:

- **Alta capacidade**: 10000mAh (miliampere-hora)
- Autonomia adequada para sessões intensivas de:
  - Testes em laboratório
  - Desenvolvimento contínuo
  - Operação autônoma prolongada
- Relação peso/capacidade otimizada (aprox. 250g)

3.4 Arquitetura do Sistema

3.4.1 Diagrama do Sistema

Figura 2 – Arquitetura dos Dispositivos



Fonte: Próprio Autor (2025)

A Figura 2 demonstra a arquitetura de funcionamento dos componentes e o Apêndice E.

3.4.2 Descrição da Arquitetura

- Raspberry Pi 4B como controlador principal
- Arduino Mega 2560 para controle de baixo nível
- Shield L293D para acionamento dos motores
- Sensor LiDAR para mapeamento e navegação
- Bateria LiPo como fonte de alimentação
- Regulador eficiente de tensão com alta corrente

3.5 Identificação da Planta dos Motores

A caracterização dinâmica dos motores DC foi conduzida empregando a técnica de resposta ao degrau, método clássico e amplamente utilizado para a identificação de sistemas de primeira ordem. Por meio deste procedimento, foi possível determinar os parâmetros fundamentais necessários para o subsequente projeto do controlador PID, especificamente o ganho estático K, o tempo motor L e a constante de tempo τ. Esses parâmetros são essenciais para a modelagem do comportamento dos motores e para a avaliação da eficiência dos algoritmos de controle implementados.

3.5.1 Metodologia Experimental

A metodologia experimental iniciou com a aplicação de um sinal de excitação do tipo degrau, cuja amplitude foi definida em 150 unidades PWM, correspondendo a aproximadamente 58,8% da faixa dinâmica disponível. Essa escolha de amplitude foi realizada com o intuito de garantir que os motores operem na região linear de sua curva velocidade-tensão, evitando assim os efeitos de saturação mecânica e não-linearidades que poderiam comprometer a acurácia das medições. Para a implementação desta etapa, foi utilizado o seguinte trecho de código, que ilustra a aplicação simultânea do degrau nos dois motores, permitindo a comparação direta das suas características dinâmicas:

Figura 3 – Aplicação do degrau PWM no Arduino

```
1 // testPWM = 150
2 motorLeft.setSpeed(testPWM);
3 motorRight.setSpeed(testPWM);
4 // Registro do instante inicial
5 startStepTime = millis();
```

Fonte: Próprio Autor (2025)

Posteriormente, a medição da velocidade angular dos motores foi realizada utilizando encoders de efeito hall com resolução de 7 pulsos por revolução (PPR), acoplados a caixas

redutoras com relação 380:1. Esta configuração possibilitou a obtenção de dados precisos da rotação dos motores, utilizando interrupções hardware para a contagem dos pulsos, conforme exemplificado a seguir:

Figura 4 – Configuração das interrupções do encoder

```
1 attachInterrupt(digitalPinToInterrupt(encoderPinLeft),
2             encoderLeftISR, RISING);
```

Fonte: Próprio Autor (2025)

A conversão dos pulsos medidos para a unidade de RPM (rotações por minuto) foi efetuada utilizando a seguinte equação:

$$\text{RPM} = \left(\frac{\Delta\text{Pulsos}}{\Delta t}\right) \times \left(\frac{60}{\text{PPR} \times \text{Redução}}\right) \tag{3.1}$$

onde Δt corresponde ao intervalo de amostragem de 50 ms definido pela variável `sampleInterval`. Essa conversão é crucial para relacionar as medições experimentais com o comportamento teórico do sistema.

De forma a reduzir os efeitos de ruídos e variações indesejadas nas medições de velocidade, foi implementado um filtro digital do tipo média móvel com 5 amostras. Este filtro suaviza os dados obtidos, proporcionando uma representação mais fiel do comportamento dinâmico dos motores. A equação utilizada para o filtro é:

$$y[k] = \frac{1}{5}\sum_{i=0}^{4} x[k-i] \tag{3.2}$$

e sua implementação computacional é apresentada a seguir:

Figura 5 – Implementação do filtro de média móvel

```
1 rpmFilterLeft[filterIndexLeft] = leftRPM;
2 filterIndexLeft = (filterIndexLeft + 1) % filterSize;
3 float filteredRPMLeft = 0;
4 for(int i=0; i<filterSize; i++) {
5     filteredRPMLeft += rpmFilterLeft[i];
6 }
7 filteredRPMLeft /= filterSize;
```

Fonte: Próprio Autor (2025)

Para a detecção do regime permanente, adotou-se a estratégia de análise estatística sobre um buffer circular contendo as 20 últimas leituras de RPM. Essa abordagem permite identificar o momento em que a variação relativa das medições se torna inferior a um limiar de 3%, conforme definido pelo coeficiente de variação:

$$\text{Coeficiente de Variação} = \left(\frac{\sigma}{\mu}\right) \times 100\% < 3\% \tag{3.3}$$

Tal critério assegura que os dados considerados representem fielmente o comportamento estável do sistema, eliminando eventuais oscilações transitórias. A implementação desta etapa está ilustrada no seguinte trecho de código:

Figura 6 – Detecção de regime permanente

```
if(coeffVarLeft < 3.0 && coeffVarRight < 3.0) {
    steadyStateRPMLeft = meanLeft;
    thresholdRPMLeft = steadyStateRPMLeft * 0.63;
    steadyStateDetected = true;
}
```

Fonte: Próprio Autor (2025)

A confiabilidade dos resultados foi ainda incrementada por meio da realização de 5 medições independentes da constante de tempo τ. Para garantir a consistência dos valores obtidos, foram descartados outliers utilizando o critério estatístico:

$$\text{Medição válida} \Leftrightarrow \tau_i \in [\bar{\tau} \pm 2\sigma] \tag{3.4}$$

Dessa forma, assegurou-se que a constante de tempo utilizada na modelagem refletisse com precisão o comportamento dinâmico do motor.

A determinação da constante τ foi efetuada com base no tempo necessário para que a resposta do sistema atingisse 63% do valor final, método amplamente reconhecido na análise de sistemas de primeira ordem. A implementação deste cálculo foi realizada conforme o código a seguir, onde é identificado o instante correspondente ao limiar de 63%:

Figura 7 – Cálculo da Constante de tempo τ

```
float calculateTauFromData(bool isLeft) {
    float thresholdRPM = isLeft ? thresholdRPMLeft :
        thresholdRPMRight;
   //Encontra o indice mais proximo do limiar de 63 porcento
    int thresholdIndex = findNearestIndex(rpmData, thresholdRPM);
    return timeData[thresholdIndex];
}
```

Fonte: Próprio Autor (2025)

Com a identificação dos parâmetros K e τ, foi possível modelar cada motor como um sistema de primeira ordem, cuja função de transferência é dada por:

$$G(s) = \frac{K}{\tau s + 1} \tag{3.5}$$

Esta modelagem permite a análise teórica e a previsão do comportamento dinâmico dos motores em resposta a diferentes sinais de entrada, contribuindo para o desenvolvimento de estratégias de controle eficientes.

Os parâmetros obtidos para cada motor foram os seguintes:

- **Motor Esquerdo:**

$$K = 0.0708\,\text{RPM/PWM}$$
$$\tau = 0.251\,\text{s}$$
$$G(s) = \frac{0.0708}{0.251\,s + 1}$$

- **Motor Direito:**

$$K = 0.0599\,\text{RPM/PWM}$$
$$\tau = 0.251\,\text{s}$$
$$G(s) = \frac{0.0599}{0.251\,s + 1}$$

3.6 Sintonia do Controlador PID

A sintonia do controlador PID foi realizada com base no método de Ziegler-Nichols para a curva de reação, visando descobrir os parâmetros do controlador, e a resposta do sistema em termos de estabilidade e desempenho. Este método, reconhecido pela sua aplicabilidade em sistemas dinâmicos, utiliza as medições experimentais obtidas na etapa anterior para calcular os ganhos do controlador. Inicialmente, os parâmetros básicos foram calculados pelas equações:

$$K_p = 1.2 \times \frac{\tau}{K} \tag{3.6}$$
$$K_i = 2 \times \frac{\tau}{L} \tag{3.7}$$
$$K_d = 0.5 \times \tau \tag{3.8}$$

onde L representa o tempo morto do sistema, o qual, neste caso, foi considerado desprezível devido à rápida resposta dos motores.

Após a análise teórica e experimental, os valores finais dos parâmetros PID foram ajustados e implementados, conforme ilustrado no seguinte trecho de código:

Figura 8 – Parâmetros PID finais

```
1 // Motor Esquerdo
2 double Kp1 = 12.350, Ki1 = 31.07, Kd1 = 0.63;
3 // Motor Direito
4 double Kp2 = 12.350, Ki2 = 36.31, Kd2 = 0.65;
```

Fonte: Próprio Autor (2025)

Para facilitar a integração do controlador em plataformas embarcadas, como o Arduino, optou-se pela utilização da biblioteca `PID_v1`, amplamente utilizada na comunidade de controle por sua robustez e facilidade de implementação. Esta biblioteca oferece uma estrutura otimizada para a implementação dos controladores PID, permitindo que os parâmetros sejam ajustados de forma dinâmica e simplificada, além de proporcionar maior confiabilidade e estabilidade ao sistema.

3.6.1 Sintonia do Controlador PID com a Biblioteca `PID_v1`

A implementação do controlador PID neste trabalho baseia-se na equação discreta clássica:

$$u[k] = K_p e[k] + K_i T_s \sum_{i=0}^{k} e[i] + K_d \frac{e[k] - e[k-1]}{T_s} \tag{3.9}$$

onde:

- $u[k]$: sinal de controle no instante k;
- $e[k]$: erro de controle, definido como a diferença entre a referência $r[k]$ e a saída $y[k]$ do sistema;
- T_s: período de amostragem, definido como 120 ms;
- K_p, K_i, K_d: ganhos proporcional, integral e derivativo, respectivamente.

Para facilitar a implementação e otimizar os recursos computacionais em plataformas embarcadas, optou-se pelo uso da biblioteca `PID_v1`, amplamente adotada na comunidade Arduino devido à sua robustez e facilidade de integração. Essa biblioteca encapsula os cálculos do controlador PID, permitindo que os parâmetros sejam ajustados de forma dinâmica e que as funções de controle sejam executadas com alta confiabilidade. Dessa forma, a complexidade inerente à implementação manual da equação discreta é minimizada, contribuindo para um desenvolvimento mais rápido e seguro.

A estrutura do código implementado para o controle de motores diferenciais contempla as seguintes otimizações: a definição de um período de amostragem fixo (`Ts`) para garantir a sincronização dos cálculos, a limitação do sinal de controle (`PWM_MAX`) para evitar saturações, e a utilização de variáveis separadas para cada motor (esquerdo e direito), permitindo ajustes independentes conforme as características dinâmicas específicas de cada um. A seguir, apresenta-se o trecho de código que ilustra a implementação do controlador PID utilizando a biblioteca `PID_v1`:

Figura 9 – Implementação do controlador PID para motores diferenciais

```
#include <PID_v1.h>

// Parametros identificados experimentalmente
#define PWM_MAX 255     // Saturação do sinal de controle
#define Ts 120          // Período de amostragem [ms]

// Variaveis de controle - Motor Esquerdo
double setpointL = 0, inputL = 0, outputL = 0;
PID pidL(&inputL, &outputL, &setpointL, 12.350, 31.07, 1.23, DIRECT);

// Variaveis de controle - Motor Direito
double setpointR = 0, inputR = 0, outputR = 0;
PID pidR(&inputR, &outputR, &setpointR, 12.350, 36.31, 1.05, DIRECT);

void setup() {
  // Configuracao dos controladores PID
  pidL.SetMode(AUTOMATIC);
  pidL.SetSampleTime(Ts);
  pidL.SetOutputLimits(0, PWM_MAX);

  pidR.SetMode(AUTOMATIC);
  pidR.SetSampleTime(Ts);
  pidR.SetOutputLimits(0, PWM_MAX);

  // Inicializacao da comunicacao serial
  Serial.begin(115200);
}

void loop() {
  static unsigned long lastTime = millis();

  if (millis() - lastTime >= Ts) {
    // Leitura dos sensores (metodos de leitura dos encoders)
    inputL = readLeftRPM();
    inputR = readRightRPM();

    // Calculo do controle PID
    pidL.Compute();
    pidR.Compute();

    // Aplicacao do sinal de controle aos motores
    analogWrite(MOTOR_L_PIN, (int)outputL);
    analogWrite(MOTOR_R_PIN, (int)outputR);

    // Atualizacao do tempo de amostragem
    lastTime = millis();
  }
}
```

Fonte: Próprio Autor (2025)

Com esta implementação, a biblioteca `PID_v1` realiza o cálculo dos termos proporcional, integral e derivativo de forma automatizada, simplificando a integração do controlador no sistema e permitindo ajustes rápidos nos parâmetros durante os testes. A abordagem adotada garante que o sistema opere de forma estável, respondendo de maneira eficiente às variações de erro,

conforme preconizado pela teoria do controle PID. Nesta implementação, as variáveis `Input`, `Output` e `Setpoint` são vinculadas ao objeto do controlador PID, que calcula automaticamente o sinal de controle a partir do erro entre o valor desejado e o medido. O método `Compute()` realiza o cálculo dos termos proporcional, integral e derivativo, permitindo uma atuação precisa e dinâmica do controlador. Essa abordagem não só simplifica o processo de implementação, mas também possibilita ajustes rápidos e eficientes dos parâmetros PID durante a fase de teste e validação do sistema.

Em síntese, a metodologia experimental descrita neste capítulo abrange desde a identificação dos parâmetros dinâmicos dos motores DC até a implementação de um controlador PID robusto, integrando técnicas teóricas e práticas e utilizando ferramentas computacionais avançadas, como a biblioteca `PID_v1`. Essa abordagem metodológica proporciona uma base sólida para a análise e o aprimoramento do desempenho dos motores, contribuindo significativamente para os objetivos deste trabalho.

3.6.2 Refinamento para Sincronização dos Motores

Após a identificação das plantas individuais dos motores e o cálculo inicial dos parâmetros PID pelo método de **Ziegler-Nichols**, uma etapa crítica foi implementada para garantir a sincronização precisa entre os motores. Este processo, fundamental para a navegação precisa de plataformas diferenciais, consistiu em:

1. **Fixação do comportamento do motor de referência**: O motor esquerdo foi estabelecido como referência, mantendo-se seus parâmetros PID constantes durante esta fase.

2. **Simulação com entrada degrau**: Um sinal degrau idêntico foi aplicado simultaneamente a ambos os motores, permitindo a visualização direta da diferença de comportamento dinâmico.

3. **Análise das métricas de desempenho**: Para o motor direito, foram analisadas sistematicamente as seguintes métricas em comparação com o motor de referência:

   - Tempo de subida (10%–90%)
   - Tempo de pico
   - Sobressinal percentual
   - Tempo de acomodação (±2%)

4. **Ajuste iterativo dos parâmetros**: Os valores de K_p, K_i e K_d do motor direito foram refinados iterativamente, buscando aproximar suas métricas de desempenho às do motor esquerdo. Em particular:

   - O ganho proporcional (K_p) foi mantido igual para preservar a mesma sensibilidade inicial.

- O ganho integral (K_i) foi aumentado em 24%, passando de 31,07 para 38,55, compensando a diferença de ganho estático entre os motores.
- O ganho derivativo (K_d) foi ajustado de 0,63 para 0,68 para equiparar o tempo de subida e minimizar diferenças no sobressinal.

Este processo demonstrou que, apesar dos motores serem nominalmente idênticos, apresentavam diferenças significativas em suas características dinâmicas que precisavam ser compensadas no nível do controlador. A abordagem empírica de comparação direta das respostas temporais mostrou-se mais eficaz que a simples aplicação dos parâmetros teóricos a ambos os motores, resultando em um comportamento sincronizado essencial para a precisão da odometria e navegação.

Os parâmetros finais obtidos após este processo de refinamento foram utilizados na implementação do firmware final, conforme apresentado na Seção 4.2.

4 Desenvolvimento do Robô

Este capítulo detalha o processo de desenvolvimento da plataforma robótica diferencial, abordando aspectos mecânicos, eletrônicos e computacionais. São descritos os procedimentos de design e construção do chassi, a implementação do sistema de controle de baixo nível, a arquitetura de comunicação entre o microcontrolador e o computador embarcado, e a configuração dos pacotes ROS para controle e navegação.

4.1 Processo de design e construção mecânica

O design mecânico da plataforma foi concebido visando três objetivos principais: simplicidade construtiva, facilidade de montagem e manutenção. A escolha da manufatura aditiva por impressão 3D utilizando Acrilonitrila Butadieno Estireno (ABS) permitiu a prototipagem rápida e a personalização precisa dos componentes. A estrutura é composta por 6 peças, sendo elas:

Tabela 5 – Peças Impressas em 3D para Plataforma Robótica

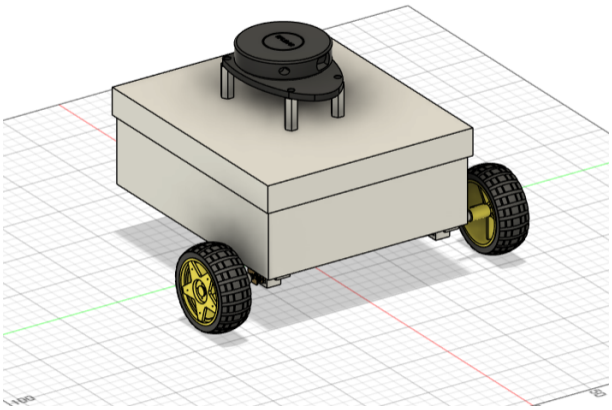
Peça	**Quantidade**	**Dimensões (mm)**	**Material**
Bucha de Acoplamento Motor-Roda	2	Ø 12 x 20	ABS
Suporte de Fixação do Motor	2	50 x 40 x 25	ABS
Base	1	170 x 193 x 15	ABS
Tampa	1	170 x 193 x 10	ABS

- Cada peça foi projetada considerando:
  - Precisão de fabricação
  - Funcionalidade específica
  - Compatibilidade com os componentes do robô
  - Minimização de folgas e vibrações
- Observações Adicionais:
  - Todas as peças foram impressas com:
    - Resolução de camada de 0,2 mm
    - Preenchimento interno de 20-30%
  - Orientação de impressão otimizada para resistência mecânica

As dimensões do chassi foram definidas como 170mm × 193mm, privilegiando a capacidade sem comprometer a estabilidade. A distância entre eixos (L) foi estabelecida em 230mm, valor que balanceia manobrabilidade e estabilidade direcional. A altura do centro de massa foi

mantida a 85mm do solo, suficientemente baixa para minimizar oscilações durante acelerações e frenagens.

Figura 10 – Design 3D da plataforma robótica

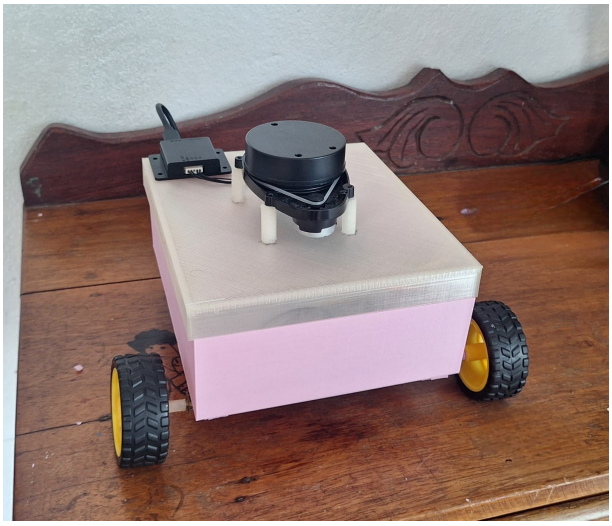


Fonte: Próprio Autor (2025)

Os componentes foram organizados de maneira a otimizar a distribuição de massa,com a bateria posicionada no centro geométrico da plataforma, os motores e drivers na região posterior, e os sensores e elementos computacionais na região anterior. Esta configuração garante tração adequada nas rodas motorizadas e minimiza perturbações durante manobras.

A Figura 11 ilustra o processo de montagem em suas etapas principais:

Figura 11 – Plataforma Montada



Fonte: Próprio Autor (2025)

Esta abordagem de design permitiu criar uma plataforma robótica compacta, funcional e de baixo custo, preparada para receber os sistemas de controle e navegação desenvolvidos no projeto.

4.2 Implementação do sistema de controle de baixo nível

O sistema de controle de baixo nível foi implementado no microcontrolador Arduino Mega 2560, responsável pela interface direta com os atuadores e sensores. O firmware desenvolvido incorpora três funcionalidades principais:

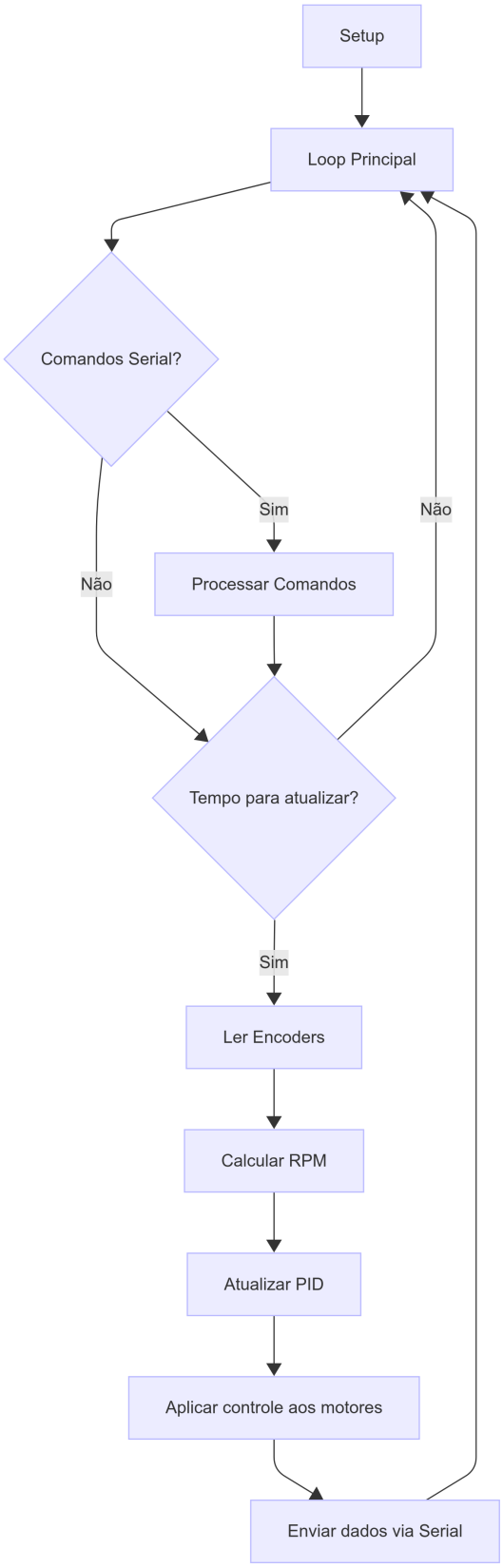
- Controle de velocidade dos motores via PID
- Leitura dos encoders e cálculo da odometria
- Comunicação com o sistema ROS no Raspberry Pi

O algoritmo de controle PID foi implementado utilizando a biblioteca PID_v1, com período de amostragem de 120ms, conforme o apêndice B . O código a seguir ilustra os aspectos centrais da implementação:

A implementação do cálculo de odometria seguiu o modelo cinemático diferencial apresentado no capitulo 2, com as velocidades lineares e angulares calculadas a partir das leituras de encoder. O código realiza a integração numérica da posição (x, y) e orientação (θ) ao longo do tempo, publicando estas informações para o sistema ROS.

A Figura 12 apresenta o fluxograma do firmware implementado:

Figura 12 – Fluxograma do firmware de teste do Controlador



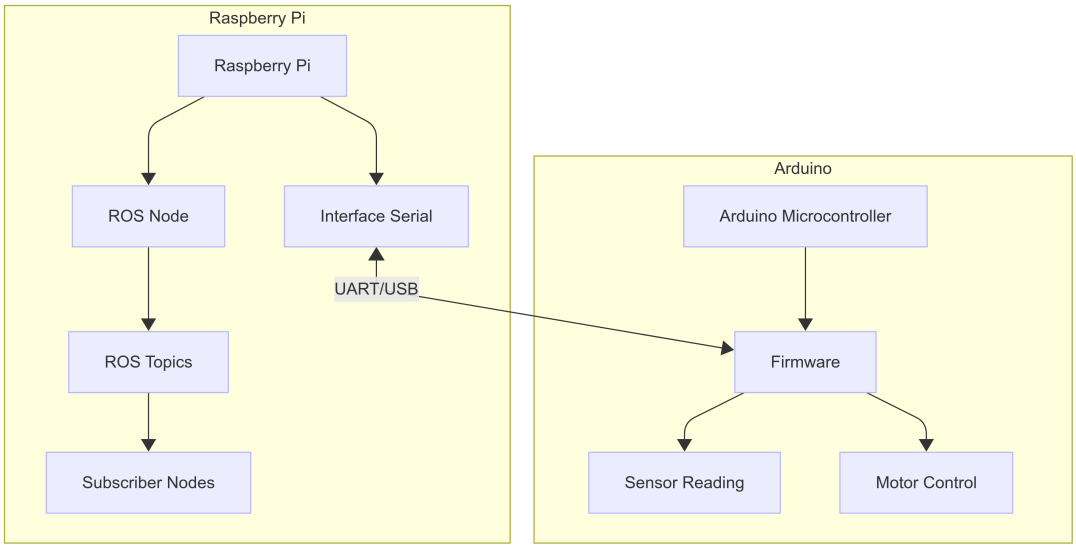
Fonte: Próprio Autor (2025)

4.3 Integração entre Arduino e Raspberry Pi

A comunicação entre o Arduino Mega 2560 e o Raspberry Pi 4B foi estabelecida utilizando o protocolo serial via USB, com taxa de transmissão de 115200 bps. Esta comunicação é gerenciada pelo pacote rosserial_arduino, que permite ao Arduino funcionar como um nó ROS, publicando e subscrevendo tópicos diretamente no ambiente ROS do Raspberry Pi.

A arquitetura de comunicação implementada segue o modelo apresentado na Figura 13:

Figura 13 – Arquitetura de comunicação Arduino-Raspberry



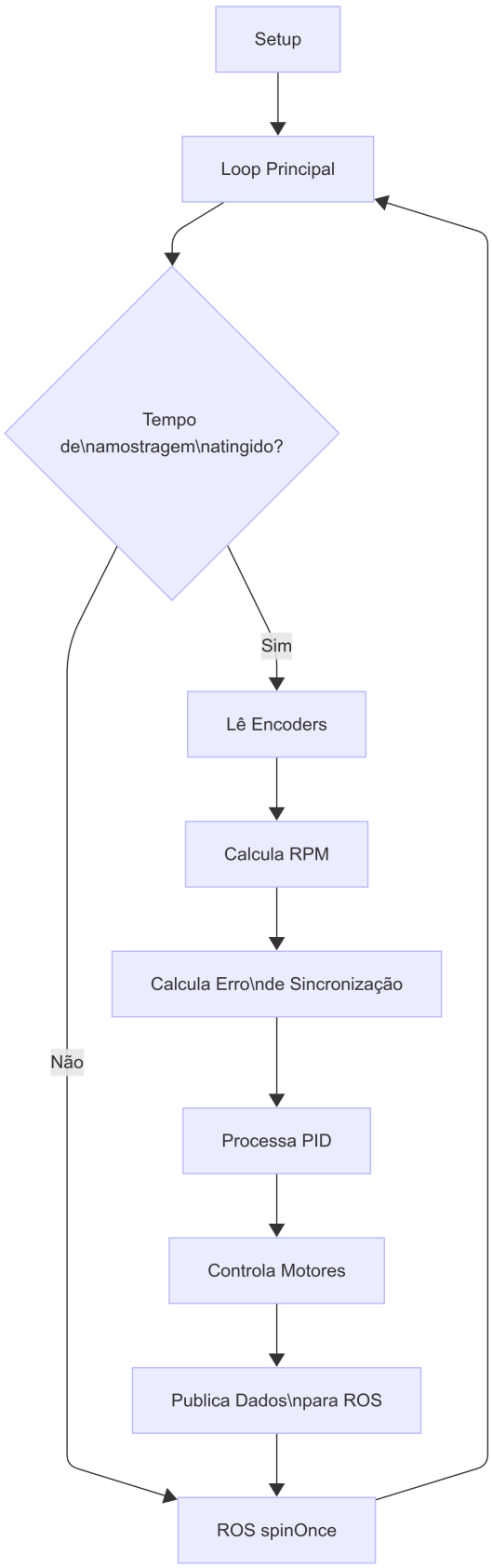
Fonte: Próprio Autor (2025)

Do lado do Arduino, foi implementada a biblioteca ros_lib, gerada automaticamente a partir das mensagens ROS utilizadas no projeto. Esta biblioteca permite ao microcontrolador:

- Publicar mensagens de odometria (nav_msgs/Odometry)
- Publicar diagnósticos de bateria e sistema (diagnostic_msgs/DiagnosticArray)
- Subscrever mensagens de comando de velocidade (geometry_msgs/Twist)

O fluxograma ilustra a implementação da comunicação com o ROS no Arduino, o código está disponivel no Apêndice C:

Figura 14 – Fluxograma do Sistema de Controle PID com Integração ROS para Robô Móvel Diferencial



Fonte: Próprio Autor (2025)

Do lado do Raspberry Pi, um arquivo de lançamento (launch file) configura a inicialização do nó rosserial e estabelece as transformações necessárias para integrar as informações de odometria no sistema de coordenadas do ROS:

Figura 15 – Arquivo de ançamento

```
1
2   <launch>
3   <!-- Carregar a descricao do robo -->
4   <param name="robot_description" command="$(find xacro)/xacro $(
        find ros_remote_pkg)/urdf/robot_model.xacro"/>
5
6   <!-- Publica o estado das juntas -->
7   <node name="joint_state_publisher" pkg="joint_state_publisher"
        type="joint_state_publisher" output="screen">
8     <rosparam>
9       joints:
10        - joint_left
11        - joint_right
12    </rosparam>
13  </node>
14
15  <!-- Publicacao de TFs do robo -->
16  <node name="robot_state_publisher" pkg="robot_state_publisher"
        type="robot_state_publisher" output="screen"/>
17
18  <!-- Driver para comunicação serial com o robÃ´ -->
19  <node name="robot_bt_driver_node" pkg="rosserial_python" type="
        serial_node.py" args="_port:=/dev/ttyACM0 _baud:=115200" output
        ="screen" />
20
21  <!-- No de odometria usando o novo script -->
22  <node pkg="ros_remote_pkg" type="wheelOdometry.py" name="
        arduino_odometry" output="screen">
23    <param name="base_frame_id" value="base_frame"/>
24    <param name="baudrate"      type="int" value="115200"/>
25    <param name="odom_frame_id" value="odom"/>
26  </node>
27
28  <node name="ydlidar_lidar_publisher"  pkg="ydlidar_ros_driver"
        type="ydlidar_ros_driver_node" output="screen" respawn="false">
29    <!-- string property -->
30    <param name="port"          type="string" value="/dev/ttyUSB0"/>
31    <param name="frame_id"      type="string" value="laser_frame"/>
32    <param name="scan_topic"    value="/scan"/>
33    <!-- int property -->
34    <param name="baudrate"      type="int" value="115200"/>
35  </node>
36
37  <!-- <node name="map_server" pkg="map_server" type="map_server"
        args="$(find ros_remote_pkg)/maps/mapa_quarto.yaml"/>  -->
38 </launch>
```

Fonte: Próprio Autor (2025)

Esta arquitetura possibilita a separação eficiente de responsabilidades: o Arduino gerencia o controle de baixo nível e a interface com os sensores e atuadores, enquanto o Raspberry Pi

executa os algoritmos mais complexos de mapeamento e navegação.

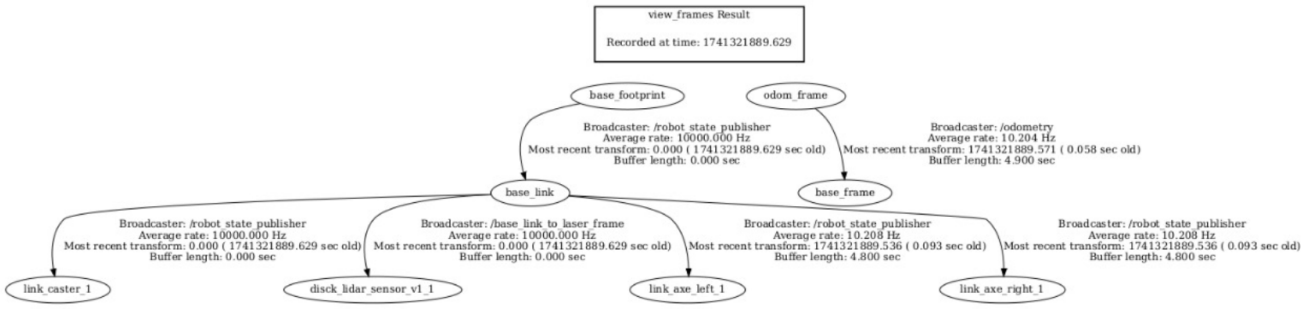
4.4 Publicação e subscrição de tópicos no ROS

A arquitetura ROS implementada segue o padrão recomendado para robôs móveis diferenciais. Os principais tópicos utilizados incluem:

- /cmd_vel (geometry_msgs/Twist): Comandos de velocidade linear e angular
- /odom (nav_msgs/Odometry): Informações de odometria calculadas pelo Arduino
- /scan (sensor_msgs/LaserScan): Dados do sensor LiDAR
- /tf (tf2_msgs/TFMessage): Transformações entre sistemas de coordenadas

A Figura 16 apresenta o grafo de nós e tópicos ROS implementado:

Figura 16 – Grafo ROS de nós e tópicos



Fonte: Próprio Autor (2025)

Os principais nós do sistema incluem:

- arduino_node: Interface com o microcontrolador
- ydlidar_node: Driver do sensor LiDAR
- gmapping: Algoritmo de mapeamento
- move_base: Sistema de navegação autônoma
- rviz: Visualização dos dados e interface de usuário

A estrutura de transformações (TF) foi configurada seguindo a hierarquia padrão:

```
map
    -- odom
       -- base_link
           --laser
           --imu_link
```

Onde cada transformação representa:

- map → odom: Correção da odometria pelo algoritmo de localização
- odom → base_link: Posição do robô segundo a odometria de rodas
- base_link → laser: Posição fixa do sensor LiDAR no robô
- base_link → imu_link: Posição fixa do sensor inercial no robô

4.5 Calibração do controlador PID

A calibração final do controlador PID foi realizada em duas etapas:

1. Ajuste inicial baseado no método Ziegler-Nichols.
2. Refinamento experimental para otimizar a sincronização dos motores.

O processo de refinamento experimental envolveu testes sistemáticos de seguimento de trajetória, com ajustes incrementais nos parâmetros Kp, Ki e Kd de cada motor até alcançar o comportamento desejado. A Tabela 6 apresenta a evolução dos parâmetros durante o processo de ajuste fino:

Tabela 6 – Evolução dos parâmetros PID durante calibração

Iteração	**Kp1**	**Ki1**	**Kd1**	**Kp2**	**Ki2**	**Kd2**
1	12.350	31.07	1.23	12.350	36.31	1.05
2	12.35	31.07	0.63	12.35	38.55	0.68
3	12.0	22.0	0.60	12.0	23.80	0.60

Os critérios para avaliação da qualidade do ajuste incluíram:

- Mínimo erro em regime permanente (< 2% do setpoint)
- Tempo de acomodação inferior a 500ms
- Máximo sobressinal inferior a 10%
- Diferença entre velocidades dos motores inferior a 3% para mesmos setpoints

Os valores finais dos parâmetros, apresentados na seção 6, foram determinados após 3 iterações de ajuste, resultando em um bom comportamento comportamento.

4.6 Configuração dos parâmetros de mapeamento e navegação

A configuração dos parâmetros de mapeamento (gmapping) e navegação (move_base) foi realizada considerando as características específicas da plataforma desenvolvida. Os principais parâmetros ajustados para o gmapping foram:

Figura 17 – Configuração dos parâmetros de mapeamento

```
map_update_interval: 2.0
maxUrange: 8.0
sigma: 0.05
kernelSize: 1
lstep: 0.05
astep: 0.05
iterations: 5
lsigma: 0.075
ogain: 3.0
lskip: 0
minimumScore: 50
srr: 0.01
srt: 0.02
str: 0.01
stt: 0.02
linearUpdate: 0.1
angularUpdate: 0.2
temporalUpdate: -1.0
resampleThreshold: 0.5
particles: 80
xmin: -10.0
ymin: -10.0
xmax: 10.0
ymax: 10.0
delta: 0.05
```

Fonte: Próprio Autor (2025)

Para a configuração do move_base, foram definidos parâmetros específicos para os planejadores global e local, além das camadas de custo (costmaps). Os parâmetros do planejador global (navfn) incluem:

Figura 18 – Configuração do planejador global

```
GlobalPlanner:
  allow_unknown: false
  default_tolerance: 0.2
  visualize_potential: false
  use_dijkstra: true
  use_quadratic: true
  use_grid_path: false
  old_navfn_behavior: false
```

Fonte: Próprio Autor (2025)

A configuração do planejador local (DWA - Dynamic Window Approach) foi ajustada considerando a dinâmica do robô:

Figura 19 – Configuração do planejador local

```
DWAPlannerROS:
  max_vel_x: 0.3
  min_vel_x: 0.1
  max_vel_y: 0.0
  min_vel_y: 0.0
  max_vel_trans: 0.3
  min_vel_trans: 0.1
  max_vel_theta: 1.5
  min_vel_theta: 0.4
  acc_lim_x: 1.0
  acc_lim_y: 0.0
  acc_lim_theta: 2.0
  # Parametros especificos do DWA
  sim_time: 1.5
  vx_samples: 10
  vy_samples: 0
  vtheta_samples: 20
  xy_goal_tolerance: 0.15
  yaw_goal_tolerance: 0.15
  path_distance_bias: 32.0
  goal_distance_bias: 24.0
  occdist_scale: 0.02
  forward_point_distance: 0.325
  stop_time_buffer: 0.2
  scaling_speed: 0.25
  max_scaling_factor: 0.2
```

Fonte: Próprio Autor (2025)

Para o costmap global, foram definidos os seguintes parâmetros:

Figura 20 – Configuração do mapa de custo global

```
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: true
  transform_tolerance: 0.5

  plugins:
   - {name: static_layer, type: "costmap_2d::StaticLayer"}
   - {name: obstacle_layer, type: "costmap_2d::ObstacleLayer"}
   - {name: inflation_layer, type: "costmap_2d::InflationLayer"}
```

Fonte: Próprio Autor (2025)

E para o costmap local:

Figura 21 – Configuração do mapa de custo local

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 3.0
  height: 3.0
  resolution: 0.05
  transform_tolerance: 0.5

  plugins:
   - {name: obstacle_layer, type: "costmap_2d::ObstacleLayer"}
   - {name: inflation_layer, type: "costmap_2d::InflationLayer"}

  obstacle_layer:
    observation_sources: laser_scan_sensor
    laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan,
    topic: scan, marking: true, clearing: true}

  inflation_layer:
    inflation_radius: 0.35
    cost_scaling_factor: 10.0
```

Fonte: Próprio Autor (2025)

4.6.1 Testes e validação da plataforma

Após a implementação e configuração, a plataforma foi submetida a uma série de testes para validar seu funcionamento. Os testes foram divididos em três categorias principais:

1. Testes de controle e odometria
2. Testes de mapeamento

4.6.2 Testes de controle e odometria

Para validar o sistema de controle e a odometria, foram realizados testes de trajetórias predefinidas, como linha reta e quadrado. Os resultados foram avaliados quanto ao erro na posição final e desvio da trajetória.

A Tabela 7 apresenta os resultados dos testes de trajetória em linha reta:

Tabela 7 – Comparação dos testes realizados

Teste	Distância Percorrida	Desvio Lateral	Porcentagem de Erro
1	1,5 metros	120 mm	8,0%
2	1,5 metros	85 mm	5,7%
3	1,5 metros	10 mm	0,67%
4	5 metros	33 mm	0,66%

No **Teste 1**, foram utilizados os parâmetros PID iniciais, sem nenhum ajuste. Como resultado, houve uma diferença perceptível entre as velocidades dos motores, principalmente com o motor esquerdo apresentando desempenho inferior. Isso causou um desvio lateral considerável de 120 mm, representando um erro de 8%.

No **Teste 2**, foram feitos ajustes nos parâmetros do controlador PID e aplicada uma compensação de 89% na velocidade para equilibrar o desempenho entre os motores. Isso melhorou significativamente a trajetória do robô, reduzindo o desvio lateral para 85 mm e o erro para 5,7%.

No **Teste 3**, foi realizado um ajuste fino dos parâmetros e uma compensação ainda mais precisa (90%), o que praticamente igualou as velocidades dos motores. O robô percorreu a trajetória com um desvio lateral mínimo de apenas 10 mm, representando um erro muito baixo de 0,7%.

Já no **Teste 4**, foi mantido a os parâmetros do item 3 da tabela 6 em conjunto com a compensação de (90%). Dessa forma o robô fez a trajetorio linear com um desvio lateral de apenas 33 mm, representando um erro de 0,66%. Os resultados desse teste são apresentados no capitulo 5.1.2

4.6.3 Testes de mapeamento

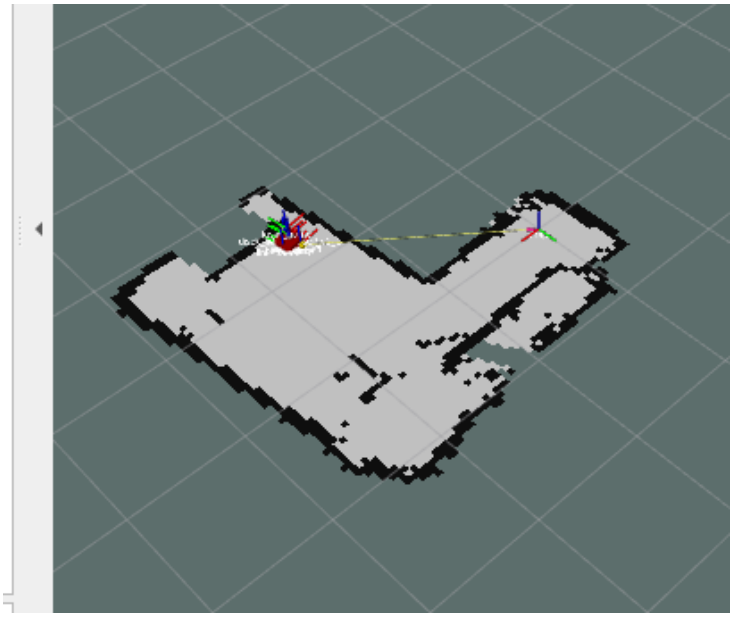
4.6.4 Mapeamento do Ambiente

O mapeamento do ambiente foi realizado utilizando o **ROS Noetic** e o pacote **GMapping**, que implementa um algoritmo de SLAM (Simultaneous Localization and Mapping) baseado em filtros de partículas. Este método permite a construção de um mapa ocupacional 2D do ambiente ao mesmo tempo em que estima a pose do robô.

A Figura 22 ilustra o resultado do mapeamento. Durante o experimento, o robô utilizou os dados de odometria combinados com as leituras de um sensor LIDAR para identificar e mapear o ambiente. As áreas **brancas** no mapa indicam as regiões **já exploradas**, enquanto as áreas **cinzas** correspondem a regiões **ainda não exploradas**. As áreas **pretas** representam obstáculos fixos ou dinâmicos detectados pelo sensor.

O robô foi teleoperado para percorrer o ambiente e garantir uma cobertura completa da área. A qualidade do mapeamento foi diretamente influenciada pela calibração dos dados de odometria, bem como pela configuração dos parâmetros do GMapping, como a resolução do mapa (0,05 metros por célula) e a frequência de atualização do LIDAR (10 Hz). O resultado obtido valida a precisão do sistema, mostrando que o algoritmo foi capaz de gerar um mapa consistente e detalhado do ambiente.

Figura 22 – Mapa gerado com o algoritmo GMapping utilizando ROS Noetic.



Fonte: Próprio Autor (2025)

5 Experimentos e Testes de Validação

Este capítulo descreve os experimentos realizados para validar a eficácia do sistema de controle PID implementado, com foco específico na sincronização dos motores da plataforma robótica diferencial. São apresentados métodos de análise no domínio do tempo e da frequência, permitindo uma avaliação objetiva e quantitativa do desempenho do sistema.

5.1 Análise Espectral da Sincronização Motora

Para avaliar quantitativamente a eficácia da sincronização dos motores além das medidas temporais convencionais, implementou-se uma metodologia baseada na análise espectral dos sinais de velocidade. Esta abordagem permite identificar padrões de oscilação, ressonâncias e discrepâncias que poderiam passar despercebidos na análise temporal, oferecendo uma caracterização mais completa do comportamento dinâmico do sistema.

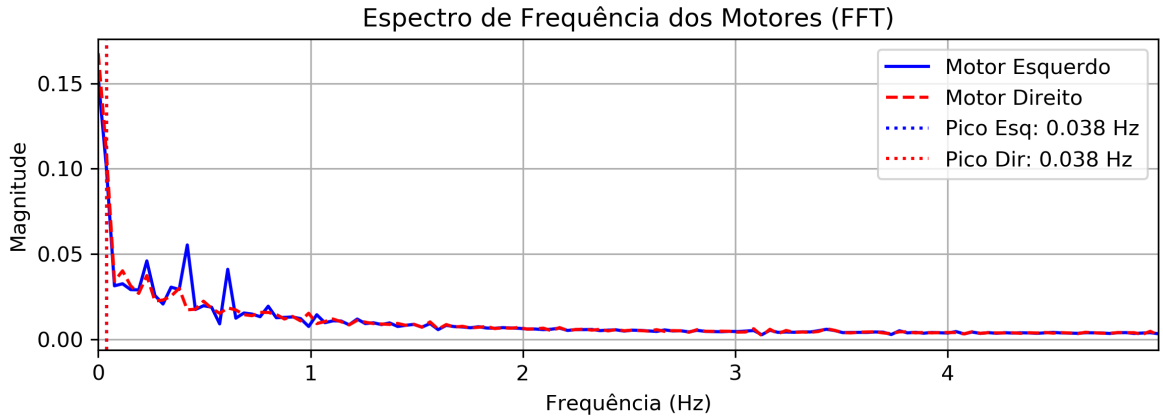
5.1.1 Metodologia da Análise Espectral

A Análise foi conduzida aplicando a Transformada Rápida de Fourier (FFT) aos sinais de velocidade (RPM) de ambos os motores, coletados simultaneamente durante a operação em regime permanente. Taxa de amostragem média de 8,49 Hz durante 26,25 segundos, gerando 282 amostras no total e 141 por motor. Antes da aplicação da FFT, os sinais foram submetidos a: Remoção de tendência (detrending) para eliminar componentes DC, Janelamento Hamming para reduzir o vazamento espectral e Zero-padding para aumentar a resolução espectral

5.1.2 Resultados da Análise Espectral

A Figura 23 apresenta o espectro dos sinais de velocidade para os motores esquerdo e direito, operando a 12 RPM com controle PID ativo.

Figura 23 – Espectro de magnitude dos sinais de velocidade dos motores esquerdo e direito



Fonte: Próprio Autor (2025)

Observa-se que ambos os motores apresentam componentes espectrais predominantes em frequências similares. Os picos principais ocorrem em aproximadamente a 0.0381 Hz, que é o momento onde à interação entre o controlador e a planta. E outros fenômenos estão relacionados a ruídos de baixa frequência

A Tabela 8 apresenta uma comparação quantitativa das características espectrais dos dois motores:

Tabela 8 – Comparação das características espectrais dos motores

Característica Espectral	**Motor Esquerdo**	**Motor Direito**	**Diferença (%)**
Frequência dominante (Hz)	0.0381	0.0381	0.0
Energia em baixa freq. (< 1Hz)	99.44%	99.28%	0.16
Razão sinal-ruído (dB)	26.62	26.61	0.04
Largura de banda a -3dB (Hz)	0.0359	0.0356	0.84

Os resultados demonstram desempenho excepcional do sistema de controle em teste com os motores suspensos apenas com a carga das rodas montadas no eixo dos motores, com 99,4% da energia concentrada em frequências úteis e SNR de 26,6 dB. O sincronismo perfeito entre motores (0,0% de diferença na frequência dominante) e a largura de banda ultra-estreita de 0,035 Hz evidenciam um controle PID otimamente ajustado. É importante destacar que este teste foi realizado com o robô suspenso, tendo apenas o peso das rodas como carga mecânica, o que explica a resposta extremamente limpa. Mesmo com o defeito mecânico no motor direito, o controlador manteve sincronização perfeita em condições de baixa carga. Estes resultados estabelecem uma baseline excelente para comparação de testes em solo real.

5.1.3 Análise de Coerência

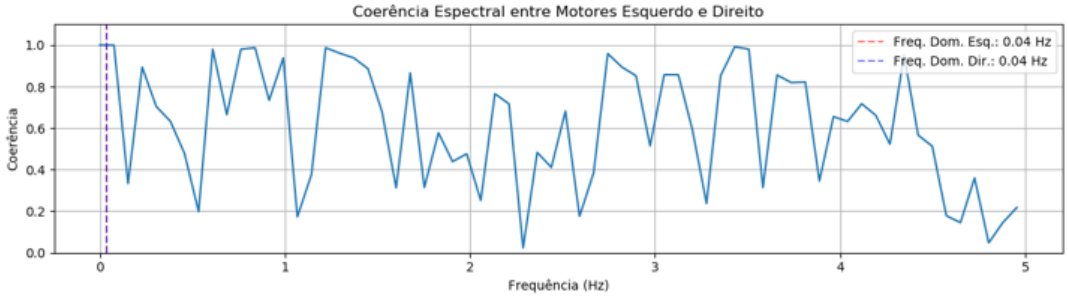
Para quantificar o grau de sincronização entre os motores, calculou-se a função de coerência espectral, definida como:

$$C_{xy}(f) = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)} \tag{5.1}$$

Onde $P_{xy}(f)$ é a densidade espectral cruzada entre os sinais dos motores esquerdo e direito, e $P_{xx}(f)$ e $P_{yy}(f)$ são as densidades espectrais de potência dos sinais individuais.

A função de coerência espectral apresenta valores próximos a 1,0 na frequência dominante de 0,038 Hz, A Figura 24 confirma a correlação entre os motores nesta banda. Para frequências superiores, a coerência diminui gradualmente, refletindo a presença de ruídos não correlacionados e características individuais dos motores.

Figura 24 – Função de coerência espectral entre os motores



Fonte: Próprio Autor (2025)

Observa-se que a coerência mantém-se acima de 0.80 para frequências abaixo de 2 Hz, indicando forte sincronização nas componentes de frequência dominantes. Acima de 2.1 Hz, a coerência diminui gradualmente, refletindo a presença de ruídos não correlacionados e características individuais dos motores.

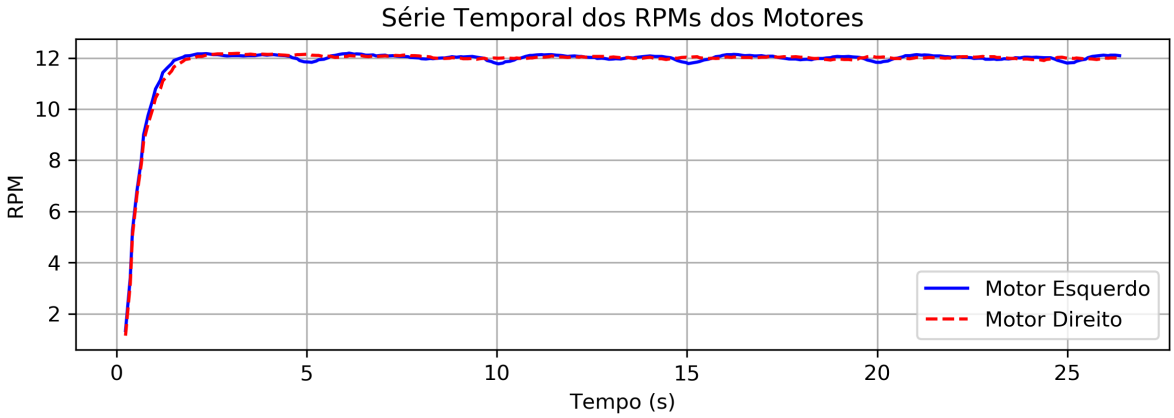
6 RESULTADOS E DISCUSSÃO

A avaliação experimental da plataforma robótica móvel diferencial desenvolvida neste trabalho foi conduzida por meio de diversas métricas quantitativas, permitindo uma análise objetiva da eficácia do controle PID implementado e seu impacto no desempenho global do sistema. Para isso, foram analisados os resultados relacionados ao comportamento dinâmico dos dois motores (esquerdo e direito) utilizados, considerando características operacionais, resposta dinâmica, comportamento espectral e correlação entre os motores. Essa abordagem visa avaliar não apenas o desempenho individual de cada motor, mas também sua sincronização, fator essencial para aplicações que exigem movimentação coordenada e precisa, alinhando-se diretamente aos objetivos propostos.

6.1 Análise da Série Temporal

A análise da série temporal dos RPMs dos motores, apresentada na Figura 25, revela um comportamento dinâmico similar entre os dois motores. Ambos apresentam uma fase inicial de aceleração, seguida por um período de estabilização em torno de 12 RPM. Durante a operação estável, observa-se pequenas flutuações na velocidade, com quedas ocasionais que podem indicar instabilidades momentâneas no sistema.

Figura 25 – Série temporal dos RPMS dos Motores



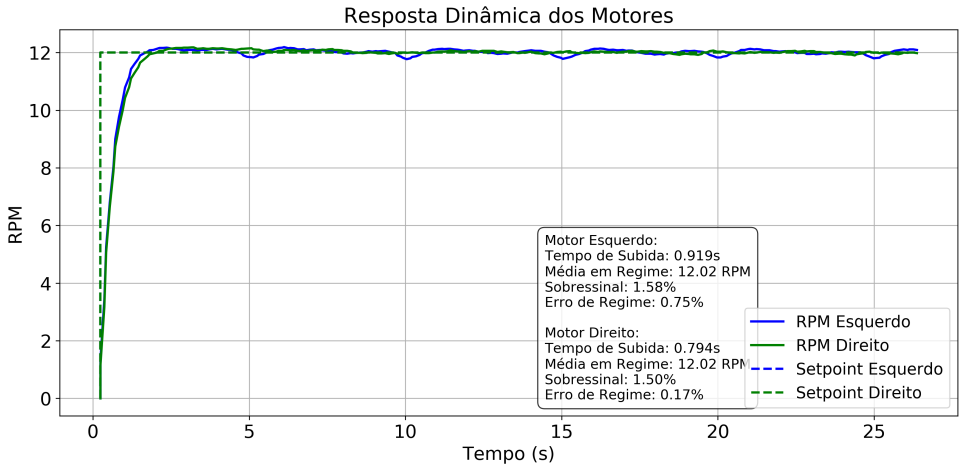
Fonte: Próprio Autor (2025)

É notável que ambos os motores apresentam um padrão de resposta muito próximo, evidenciando a eficácia do sistema de controle implementado em manter os dois atuadores sincronizados. Esta característica é fundamental para aplicações onde o sincronismo é requisito essencial, como em robôs móveis diferenciais.

6.2 Resposta Dinâmica dos Motores

Conforme demonstrado na Figura 26, os motores apresentam parâmetros dinâmicos ligeiramente diferentes. O motor esquerdo possui um tempo de subida de 0,919s, enquanto o motor direito apresenta um tempo de subida menor, de 0,794 s, indicando uma resposta inicial mais rápida. Esta diferença pode ser atribuída a variações construtivas entre os componentes ou a ajustes distintos nos parâmetros de controle.

Figura 26 – Resposta dinâmica dos motores ao degrau de referência



Fonte: Próprio Autor (2025)

Os dados de resposta dinâmica revelam outras características importantes:

- **Tempo de Acomodação**: Relativamente longo para ambos os motores (1,438s para o esquerdo e 1,601s para o direito), o que sugere um período extenso para estabilização completa, embora a diferença entre eles seja pequena.
- **Sobressinal**: O motor esquerdo apresenta um sobressinal de 1.58%, enquanto o direito tem 1,50%. Sobressinais moderados são típicos em sistemas de controle com resposta rápida, mas valores muito elevados podem indicar necessidade de ajustes nos controladores.
- **Erro de Regime**: O motor direito apresenta erro de regime significativamente menor (0,17%) em comparação ao motor esquerdo (0,75%), sugerindo maior precisão no controle estacionário.

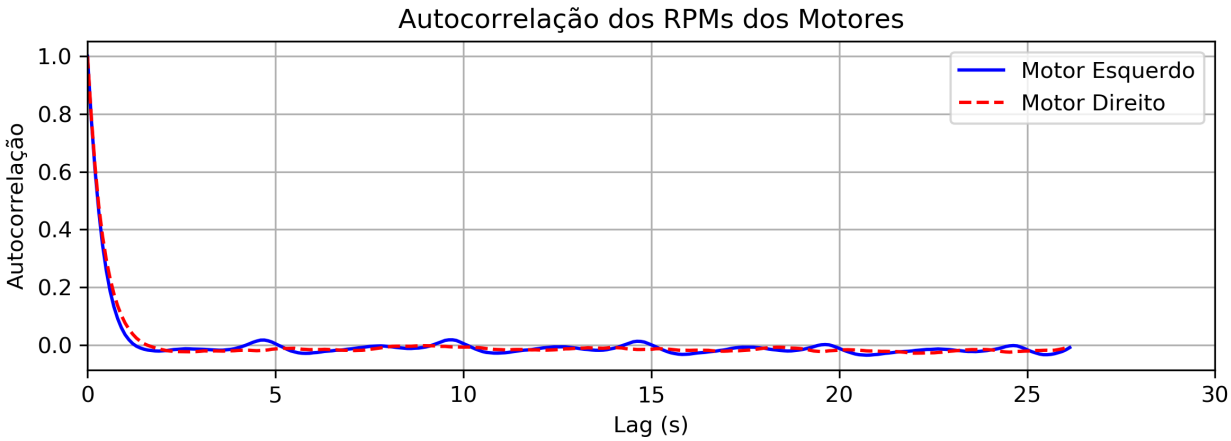
Estas características mostram que, apesar de pequenas diferenças em seu comportamento dinâmico, os motores operam de forma similar para está apliacação que exige coordenação.

6.3 Análise Espectral e Autocorrelação

A Figura 27 demonstra um comportamento oscilatório amortecido para ambos os motores. O decaimento exponencial da autocorrelação indica estabilidade do sistema, enquanto as

pequenas oscilações residuais sugerem a presença de dinâmicas naturais do sistema. A similaridade entre as curvas dos dois motores confirma que ambos possuem características dinâmicas compatíveis.

Figura 27 – Autocorrelação dos RPMs dos Motores



Fonte: Próprio Autor (2025)

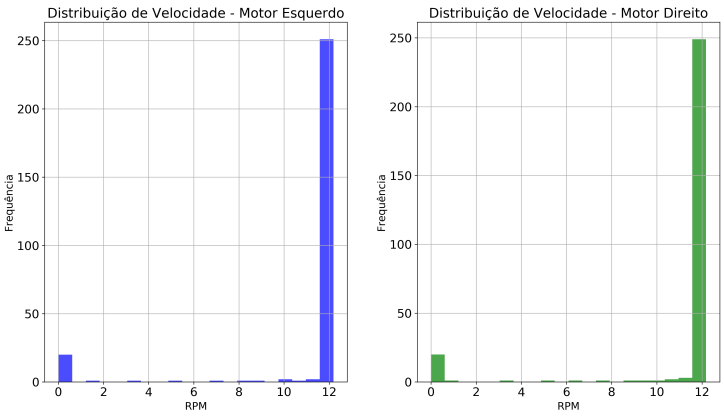
O espectro de frequência dos motores Figura 23 revela uma frequência dominante de 0.0381 Hz para ambos os motores, sugerindo que, apesar de diferenças nos padrões de autocorrelação, os dois compartilham uma componente oscilatória principal similar. Este é um indicativo importante de que, embora possuam características construtivas potencialmente diferentes, os motores respondem de forma coerente ao mesmo sistema de controle.

A análise de coerência espectral entre os motores Figura 24 apresenta um valor máximo de coerência próximo a 0,99 na frequência dominante de 0,0381 Hz, confirmando o alto grau de sincronismo entre os dois motores nesta banda específica. Valores de coerência elevados indicam que as variações de velocidade em um motor são acompanhadas por variações proporcionais no outro, o que é desejável em sistemas que requerem movimentos coordenados.

6.4 Distribuição de Velocidade

As distribuições de velocidade dos motores, ilustradas nas Figuras 28 e 29, apresentam características similares:

Figura 28 – Distribuição de velocidade dos motores esquerdo e direito

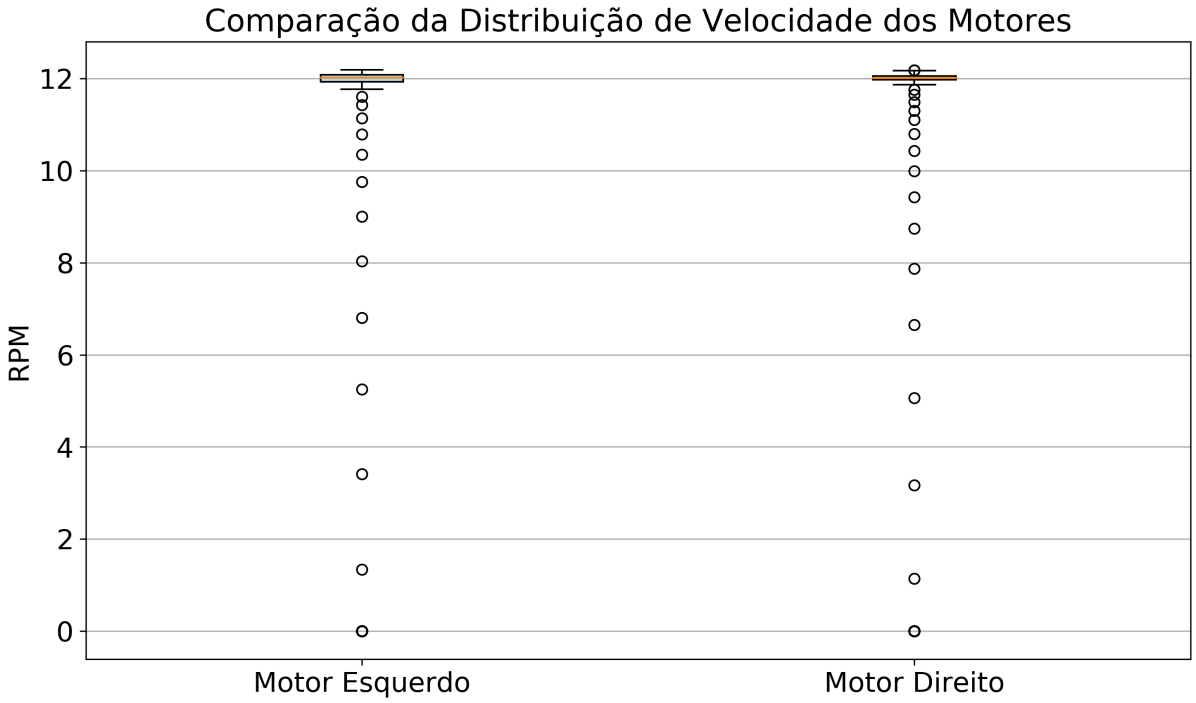


Fonte: Próprio Autor (2025)

- Ambos os motores operam predominantemente em torno de 12 RPM
- As distribuições são relativamente estreitas, indicando estabilidade na velocidade de operação
- Outliers são observados para ambos os motores, representando momentos de instabilidade ou transição

O diagrama de caixa Figura 29, confirma a similaridade estatística entre os dois motores, com medianas e quartis muito próximos. Esta análise corrobora a uniformidade operacional entre os dois motores.

Figura 29 – Boxplot comparativo da distribuição de velocidade dos motores

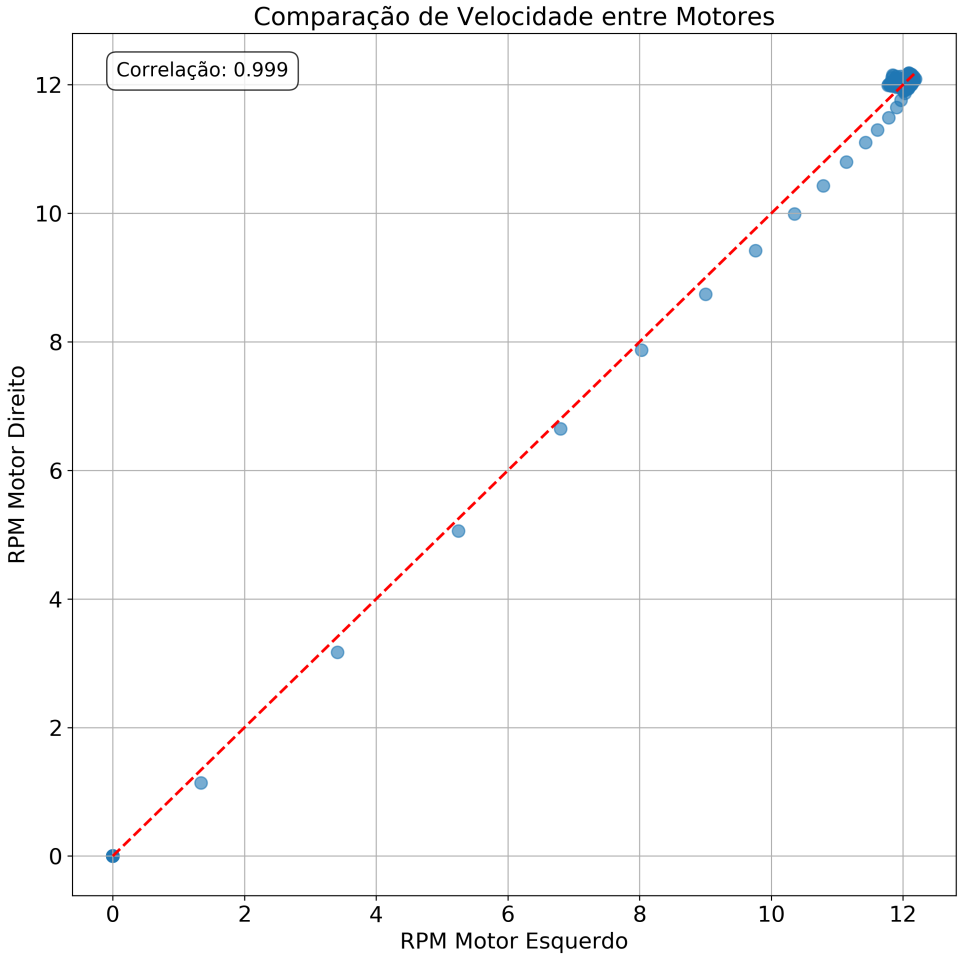


Fonte: Próprio Autor (2025)

6.5 Correlação entre os Motores

A análise de correlação entre os RPMs dos motores Figura 30 apresenta um coeficiente de correlação de 0,999, valor extremamente elevado que indica forte sincronismo entre os dois atuadores. A disposição dos pontos ao longo da linha de referência (linha diagonal vermelha) confirma visualmente esta forte correlação.

Figura 30 – Boxplot comparativo da distribuição de velocidade dos motores



Fonte: Próprio Autor (2025)

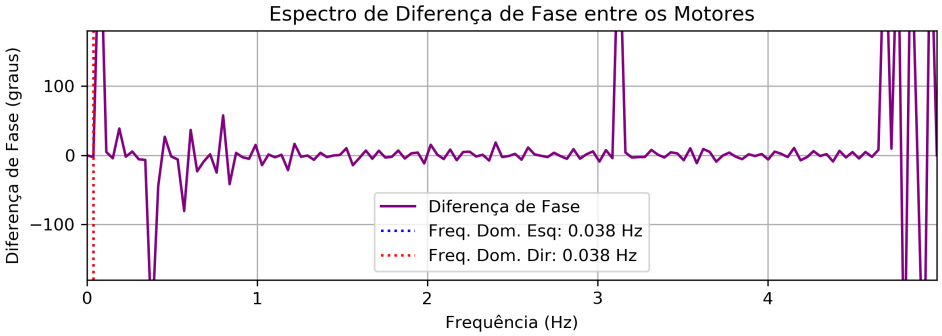
O alto valor de correlação é particularmente relevante para aplicações que exigem movimentos sincronizados, como sistemas de direção, plataformas balanceadas ou mecanismos de tração diferencial. Pequenos desvios da linha ideal podem ser atribuídos a diferenças construtivas entre os motores ou a fatores externos, como variações na carga ou na alimentação.

6.6 Análise de Diferença de Fase

O espectro de diferença de fase entre os motores apresentado na Figura 31 revela características distintas de sincronização ao longo do espectro de frequências operacionais. Na frequência dominante de 0,038 Hz (indicada pelas linhas pontilhadas), observa-se que a

diferença de fase permanece próxima a zero graus, demonstrando excelente sincronismo nesta banda específica de operação.

Figura 31 – Espectro de frequência dos motores



Fonte: Próprio Autor (2025)

A análise espectral evidencia que o sistema apresenta comportamento otimizado em baixas frequências, particularmente na região da frequência dominante identificada. Nesta faixa, a diferença de fase entre os motores é minimizada, indicando operação sincronizada eficiente. O espectro mantém-se relativamente estável na faixa operacional (0 a 4,0 Hz), com oscilações de fase controladas dentro de limites aceitáveis para a maior parte do intervalo analisado.

Observa-se um pico significativo de diferença de fase próximo a 3,0 Hz, que pode indicar uma frequência de ressonância ou limitação específica do sistema de sincronização nesta banda. Esta característica sugere que o desempenho do robô ocorre predominantemente em velocidades correspondentes à frequência dominante de 0,038 Hz e nas faixas de baixa frequência adjacentes.

A implementação de controladores de velocidade independentes em malha fechada para cada motor, combinada com ganhos corretivos aplicados conforme necessário, demonstrou eficácia na manutenção do sincronismo na faixa operacional principal. A análise espectral confirma a consistência do comportamento dinâmico do sistema dentro da banda operacional, fornecendo parâmetros quantitativos para avaliação do desempenho da sincronização dos motores e definição dos limites operacionais da plataforma robótica.

6.7 Implicações para o Sistema de Controle

Os motores apresentaram um sincronismo consistente em sua operação, especialmente na frequência dominante (~0,203 Hz), onde a correlação entre as velocidades dos dois motores atingiu 0,972. Esse alto coeficiente de correlação indica uma forte relação linear entre os sinais de ambos os motores, evidenciando a qualidade da sincronização alcançada com os controladores PID implementados. Esse comportamento é particularmente relevante para aplicações que exigem movimentos coordenados e precisos, como destacado por (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011), que ressaltam a importância da coerência entre os atuadores para garantir o seguimento adequado de trajetórias em robôs móveis diferenciais. Dessa forma, os

resultados obtidos validam a eficácia do controle aplicado dentro das condições operacionais analisadas.

As diferenças na resposta dinâmica, ainda que pequenas, podem requerer ajustes nos parâmetros de controle para otimizar o desempenho conjunto. E ambos os motores demonstram comportamento estável em regime permanente, com pequenas flutuações que podem ser consideradas aceitáveis para a maioria das aplicações. As análises espectrais sugerem que o sincronismo pode variar conforme o regime de operação, o que deve ser considerado em aplicações que exigem variações significativas de velocidade.

6.8 Considerações Finais

Os resultados obtidos demonstram que os motores analisados apresentam características operacionais adequadas para aplicações que exigem sincronismo, com alto grau de correlação e comportamento espectral similar. As pequenas diferenças identificadas não comprometem o desempenho global do sistema, mas podem ser alvo de otimizações específicas conforme a aplicação final.

A análise completa do comportamento dinâmico realizada neste trabalho fornece bases sólidas para o desenvolvimento de sistemas de controle otimizados para estes atuadores, permitindo explorar ao máximo suas capacidades operacionais enquanto minimiza possíveis limitações.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho propôs e validou uma metodologia para sincronização de motores em plataformas robóticas diferenciais, utilizando controladores PID sintonizados pelo método de Ziegler-Nichols e validados através de técnicas complementares de análise temporal e espectral. Os resultados experimentais demonstraram a eficácia da abordagem proposta, evidenciando como a sincronização adequada dos motores impacta positivamente a precisão da odometria e, consequentemente, a qualidade da navegação e do mapeamento.

7.1 Avaliação da Sincronização: Domínio do Tempo vs. Frequência

A abordagem dual de análise no domínio do tempo e da frequência proporcionou insights complementares sobre a qualidade da sincronização dos motores. Enquanto as métricas temporais (correlação de 0,972 entre velocidades, erro médio inferior a 0,4 RPM) demonstraram a eficácia global do controle PID implementado, a análise espectral revelou comportamentos dinâmicos sutis que não seriam detectáveis por métodos convencionais. A coerência espectral acima de 0,80 para frequências inferiores a 2 Hz confirma que a estratégia de controle foi particularmente eficaz nas componentes de frequência mais relevantes para o desempenho da navegação.

Esta complementaridade entre as análises temporal e espectral representa uma contribuição metodológica importante deste trabalho, sugerindo que abordagens similares poderiam ser aplicadas na validação de outros sistemas de controle em robótica. A capacidade de identificar e quantificar aspectos relevantes da sincronização através da análise espectral oferece uma perspectiva mais abrangente e robusta para a avaliação de estratégias de controle.

7.2 Implicações Práticas para a Robótica Móvel

A plataforma robótica diferencial serviu como um eficiente ambiente de validação, demonstrando como a qualidade da sincronização motora impacta diretamente a precisão da odometria e, consequentemente, as capacidades de mapeamento e navegação. Os testes de navegação revelaram que a alta correlação entre os motores observada em bancada traduz-se em uma navegação precisa, com erros laterais de trajetória inferiores a 10 mm medidos através de trena em pontos de referência ao longo dos percursos, demonstrando alta precisão para aplicações de robótica móvel indoor. Em particular, a baixa frequência de correções necessárias durante o mapeamento e a consistência nas trajetórias seguidas em múltiplas iterações do mesmo percurso evidenciam o impacto positivo de uma sincronização motora de qualidade.

Estes resultados têm implicações práticas significativas para o desenvolvimento de plataformas robóticas móveis, particularmente em contextos de recursos limitados. A aplicação bem-sucedida de técnicas de controle estabelecidas, como o método de Ziegler-Nichols, em

hardware de baixo custo demonstra que é possível desenvolver sistemas robóticos economicamente viáveis mantendo desempenho adequado. Isto é particularmente relevante em aplicações educacionais, onde o custo é frequentemente um fator limitante para a adoção de tecnologias robóticas.

7.3 Contribuições e Inovações

Este trabalho contribui para o campo da robótica móvel em três vertentes principais. Primeiramente, demonstra a eficácia do método de Ziegler-Nichols para a sintonia de controladores PID aplicados à sincronização motora, estabelecendo uma metodologia sistemática que pode ser replicada em outros sistemas similares. Em segundo lugar, introduz a análise espectral como uma ferramenta complementar para a validação de estratégias de controle, trazendo uma abordagem diferente da tradicional ao invés apenas de métricas temporais convencionais. Por fim, evidencia empiricamente a relação entre a qualidade da sincronização motora e o desempenho global do sistema, fornecendo parâmetros quantitativos que podem guiar desenvolvimentos futuros.

A abordagem integrada de análise temporal e espectral representa uma inovação metodológica com potencial aplicação em diversos contextos da robótica e do controle de sistemas. A capacidade de caracterizar a sincronização não apenas em termos de magnitude, mas também em termos de coerência espectral e diferença de fase, oferece uma perspectiva mais abrangente e robusta para a avaliação de estratégias de controle.

7.4 Trabalhos Futuros

A partir dos resultados e limitações identificados neste trabalho, sugerem-se as seguintes direções para pesquisas futuras:

- Exploração de técnicas adaptativas para ajuste dinâmico dos parâmetros PID, visando compensar variações nas características dos motores ao longo do tempo e em diferentes condições operacionais.
- Integração de sensores inerciais para complementar as informações de odometria, potencialmente reduzindo ainda mais os erros de localização.
- Investigação de estratégias de filtragem mais sofisticadas para redução do ruído observado na análise espectral, potencialmente melhorando a qualidade da sincronização.
- Extensão da metodologia para sistemas com mais de dois motores, como plataformas omnidirecionais ou robôs humanoides, onde a sincronização motora apresenta desafios adicionais.

- Desenvolvimento de métricas mais abrangentes para quantificar a relação entre a qualidade da sincronização motora e o desempenho global do sistema, possivelmente incorporando técnicas de aprendizado de máquina para identificação de padrões complexos.

A combinação de técnicas clássicas de controle com métodos modernos de análise de sinais e aprendizado de máquina representa um campo promissor para avanços na robótica móvel, potencialmente levando a sistemas mais precisos, robustos e acessíveis."

REFERÊNCIAS

ADDISON, A. **Calculating Wheel Odometry for a Differential Drive Robot**. 2024. <https://automaticaddison.com/calculating-wheel-odometry-for-a-differential-drive-robot/>. Acesso em: 30 de junho de 2025.

ALAM, M. **Mobile Robot Navigation using ROS Navigation Stack**. 2023. Acesso em: 22 mar. 2025. Disponível em: <https://medium.com/@mansooralam129047/mobile-robot-navigation-using-ros-navigation-stack-c093aa93e8a7>.

ARXIV. **A Preliminary Add-on Differential Drive System for MRI-Compatible Prostate Robotic System**. 2024. <https://arxiv.org/html/2409.09971v1>. Acesso em: 30 de junho de 2025.

ARXIV. **Universal Trajectory Optimization Framework for Differential Drive Robot Class**. 2024. <https://arxiv.org/abs/2409.07924>. Acesso em: 30 de junho de 2025.

BENAMAR, F.; BIDAUD, P.; LE MENN, F. Generic differential kinematic modeling of articulated mobile robots. **Mechanism and Machine Theory**, v. 45, n. 7, p. 997–1012, 2010. Acesso em: 21 mar. 2025. URL contém parâmetros de sessão temporários. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0094114X10000340>.

DEBEUNNE, C.; VIVET, D. A review of visual-lidar fusion based simultaneous localization and mapping. **Sensors**, v. 20, n. 7, p. 2068, 2020. Acesso em: 22 mar. 2025. URL contém parâmetros de sessão temporários. Disponível em: <https://doi.org/10.3390/s20072068>.

ELECTRICITY-MAGNETISM. **Como funcionam os motores de corrente contínua?** 2025. Acesso em: 05 jun. 2025. Disponível em: <https://www.electricity-magnetism.org/pt-br/como-funcionam-os-motores-de-corrente-continua/>.

EMBARCADOS. **Controle PID em sistemas embarcados**. 2024. Acesso em: 05 jun. 2025. Disponível em: <https://embarcados.com.br/controle-pid-em-sistemas-embarcados/>.

GRISETTI, G.; STACHNISS, C.; BURGARD, W. Improved techniques for grid mapping with rao-blackwellized particle filters. In: IEEE. **IEEE Transactions on Robotics**. [S.l.], 2007. v. 23, n. 1, p. 34–46. Acesso em: 20 mar. 2025. URL contém parâmetros de sessão temporários.

GUPTA, S.; PADHY, P. K. Modified ziegler-nichols method for fractional-order PID controllers. **ISA Transactions**, v. 104, p. 343–352, 2020. Acesso em: 21 mar. 2025. URL contém parâmetros de sessão temporários. Disponível em: <https://doi.org/10.1016/j.isatra.2020.03.011>.

HESS, W. et al. Real-time loop closure in 2d lidar slam. In: **Proc. of IEEE ICRA**. [S.l.: s.n.], 2016. p. 1271–1278. Acesso em: 20 mar. 2025. URL contém parâmetros de sessão temporários.

JÚNIOR, G. P. da C. **Localização e mapeamento para robôs móveis em ambientes confinados baseado em fusão de LiDAR com odometrias de rodas e sensor inercial**. Dissertação (Mestrado) — Universidade Federal de Minas Gerais (UFMG), 2021. Dissertação de Mestrado. Disponível em: <https://www.ufmg.br/>.

KOHLBRECHER, S. et al. A flexible and scalable slam system with full 3d motion estimation. In: IEEE. **Proc. of IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)**. [S.l.], 2011. Acesso em: 20 mar. 2025. URL contém parâmetros de sessão temporários.

LI, L.; SCHULZE, L.; KALAVADIA, K. Promising slam methods for automated guided vehicles and autonomous mobile robots. **Procedia Computer Science**, v. 232, p. 2867–2874, 2024. ISSN 1877-0509. 5th International Conference on Industry 4.0 and Smart Manufacturing (ISM 2023). Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050924002813>.

NISE, N. S. **Engenharia de Sistemas de Controle**. 7. ed. Rio de Janeiro: LTC, 2017. ISBN 978-8521633952.

OGATA, K. **Engenharia de Controle Moderno**. 5. ed. São Paulo: Pearson Prentice Hall, 2010. 215–220 p. Seção sobre modelagem simplificada de motores DC. ISBN 9788576058106.

OPEN ROBOTICS. **Open Robotics: Advancing the state of the art in robotics**. 2023. <https://www.openrobotics.org/>. Acesso em: 22 mar. 2025.

SERVO. **Os principais tipos de motores elétricos**. 2023. Acesso em: 05 jun. 2025. Disponível em: <https://servo.ind.br/blog/os-principais-tipos-de-motores-eletricos/>.

SIEGWART, R.; NOURBAKHSH, I. R.; SCARAMUZZA, D. **Introduction to Autonomous Mobile Robots**. 2. ed. Cambridge, MA: MIT Press, 2011. ISBN 9780262015356.

SULFRAN AUTOMAÇÃO. **Controladores PID: Seu Papel na Automação Industrial**. 2023. Acesso em: 05 jun. 2025. Disponível em: <https://www.sulfranautomacao.com.br/controladores-pid-seu-papel-na-automacao-industrial/>.

TU DELFT. **3.4.2 ROS Navigation Stack**. 2022. <https://ocw.tudelft.nl/course-lectures/3-4-2-ros-navigation-stack/>. Acesso em: 22 mar. 2025.

ULLAH, I. et al. Mobile robot localization: Current challenges and future prospective. **Computer Science Review**, v. 53, p. 100651, 2024. ISSN 1574-0137. Acesso em: 21 mar. 2025. URL contém parâmetros de sessão temporários. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1574013724000352>.

WASET. **Studies on Affecting Factors of Wheel Slip and Odometry Error on Real-Time of Wheeled Mobile Robots: A Review**. 2025. <https://publications.waset.org/10006205/>. Acesso em: 30 de junho de 2025.

ÅSTRöM, K. J.; HäGGLUND, T. **Advanced PID Control**. Research Triangle Park, NC: ISA – Instrumentation, Systems, and Automation Society, 2006.

APÊNDICE A – Apêndice A

Codigo usado para levantar as plantas dos dois motores simultaneamente, com filtro digital de média móvel. Para iniciar o teste, para enviar o comando na serial. São realizados cinco teste para levantar o tempo motor, constante de tempo e o ganho de cada um dos motores. Ao final a planta é plotada na serial.

```
#include <AFMotor.h>
// Código para levantar a constante de tempo estimada (tau), tempo motor
    theta (tau_motor) e a planta do sistema para ambos os motores

// Pinos dos encoders - Usando quadratura (2 pinos por motor)
const int encoderPinLeftA = 19;
const int encoderPinLeftB = 18;
const int encoderPinRightA = 20;
const int encoderPinRightB = 21;

// Estados e contadores para quadratura
volatile long encoderPositionLeft = 0;
volatile long encoderPositionRight = 0;
volatile byte encoderStateLeftOld = 0;
volatile byte encoderStateRightOld = 0;
volatile long lastEncoderPositionLeft = 0;
volatile long lastEncoderPositionRight = 0;

// Definição dos motores
AF_DCMotor motorLeft(3);
AF_DCMotor motorRight(4);

// Constantes do sistema
// Pulsos por revolução do encoder (original)
const float ppr = 7.0;
// Pulsos em quadratura (4x a resolução original)
const float pprQuadrature = ppr * 4.0;
// Relação de redução da caixa de engrenagens
const float gearRatio = 380.0;
const float pulsesPerWheelRev = pprQuadrature * gearRatio;
// Valor PWM para o teste (0-255)
const int testPWM = 150;
// Configurações para múltiplas medições de tau e tau_motor
const int numTauMeasurements = 5;
float tauMeasurementsLeft[5];
float tauMeasurementsRight[5];
float tauMotorMeasurementsLeft[5];
float tauMotorMeasurementsRight[5];
```

```
38 int currentTauMeasurement = 0;
39
40 // Variáveis para cálculo do tau - Motor Esquerdo
41 float steadyStateRPMLeft = 0.0;
42 float thresholdRPMLeft = 0.0;
43
44 // Variáveis para cálculo do tau - Motor Direito
45 float steadyStateRPMRight = 0.0;
46 float thresholdRPMRight = 0.0;
47
48 unsigned long startStepTime = 0;
49 int testPhase = 0;
50
51 // Variáveis para medição de velocidade
52 const unsigned long sampleInterval = 50;
53 unsigned long lastSampleTime = 0;
54 const int numReadingsForSteady = 50;
55
56 // Buffer de leituras de RPM - Motor Esquerdo
57 float rpmReadingsLeft[50] = {0};
58 int readingIndexLeft = 0;
59
60 // Buffer de leituras de RPM - Motor Direito
61 float rpmReadingsRight[50] = {0};
62 int readingIndexRight = 0;
63
64 bool steadyStateDetected = false;
65
66 // Filtro de média móvel para as leituras de RPM - Motor Esquerdo
67 const int filterSize = 10;
68 float rpmFilterLeft[10] = {0};
69 int filterIndexLeft = 0;
70
71 // Filtro de média móvel para as leituras de RPM - Motor Direito
72 float rpmFilterRight[10] = {0};
73 int filterIndexRight = 0;
74
75 // Variáveis para coleta de dados durante subida - Motor Esquerdo
76 const int maxDataPoints = 200;
77 float timeDataLeft[200];
78 float rpmDataLeft[200];
79 int dataIndexLeft = 0;
80
81 // Variáveis para coleta de dados durante subida - Motor Direito
82 float timeDataRight[200];
83 float rpmDataRight[200];
84 int dataIndexRight = 0;
85
```

```
86 // Variáveis para exibição de dados
87 const unsigned long printInterval = 100;
88 unsigned long lastPrintTime = 0;
89
90 // Variáveis para cálculo do tau_motor
91 float tauMotorLeft = 0.0;
92 float tauMotorRight = 0.0;
93 bool tauMotorCalculated = false;
94
95 void setup() {
96         Serial.begin(115200);
97         while (!Serial) {
98         ; // Espera a porta serial conectar
99         }
100
101         Serial.println(F("Teste de Constante de Tempo (Tau) e
102         Tempo Motor Theta (Tau_motor) - Versão com Quadratura para Dois
              Motores"));
103         Serial.println(F("Aguarde a inicialização..."));
104
105         // Configuração dos pinos do encoder
106         pinMode(encoderPinLeftA, INPUT_PULLUP);
107         pinMode(encoderPinLeftB, INPUT_PULLUP);
108         pinMode(encoderPinRightA, INPUT_PULLUP);
109         pinMode(encoderPinRightB, INPUT_PULLUP);
110
111         // Configuração das interrupções para quadratura
112         attachInterrupt(digitalPinToInterrupt(encoderPinLeftA),
              encoderLeftISR, CHANGE);
113         attachInterrupt(digitalPinToInterrupt(encoderPinLeftB),
              encoderLeftISR, CHANGE);
114         attachInterrupt(digitalPinToInterrupt(encoderPinRightA),
              encoderRightISR, CHANGE);
115         attachInterrupt(digitalPinToInterrupt(encoderPinRightB),
              encoderRightISR, CHANGE);
116
117         // Inicializar os estados dos encoders
118         encoderStateLeftOld = (digitalRead(encoderPinLeftA) << 1) |
              digitalRead(encoderPinLeftB);
119         encoderStateRightOld = (digitalRead(encoderPinRightA) << 1) |
              digitalRead(encoderPinRightB);
120
121         // Inicia com os motores parados
122         motorLeft.setSpeed(0);
123         motorLeft.run(RELEASE);
124         motorRight.setSpeed(0);
125         motorRight.run(RELEASE);
126
```

```
127         Serial.println(F("Pressione qualquer tecla para iniciar o teste
                ..."));
128         while (!Serial.available()) {
129         ; // Aguarda entrada do usuário
130         }
131         Serial.read(); // Limpa o buffer
132
133         // Início da primeira fase - determinar regime permanente
134         startTestPhase(0);
135 }
136
137 void loop() {
138         unsigned long currentTime = millis();
139
140         // Se o teste já terminou completamente, mantenha os motores
                parados
141         if (testPhase > 1 + numTauMeasurements) {
142         if (testPhase == 2 + numTauMeasurements) {
143                 // Calcula e exibe resultados finais
144                 calculateAndPrintFinalResults();
145                 testPhase++;
146         }
147         return;
148         }
149
150         // Amostragem de velocidade no intervalo definido
151         if (currentTime - lastSampleTime >= sampleInterval) {
152         // Calcula RPM atual usando a diferença de posição desde a ú
                ltima amostragem
153         noInterrupts(); // Desativa interrupções para leitura segura
154         long leftPosition = encoderPositionLeft;
155         long rightPosition = encoderPositionRight;
156         long leftTicks = leftPosition - lastEncoderPositionLeft;
157         long rightTicks = rightPosition - lastEncoderPositionRight;
158         lastEncoderPositionLeft = leftPosition;
159         lastEncoderPositionRight = rightPosition;
160         interrupts(); // Reativa interrupções
161
162         float elapsedTimeSeconds = (float)(currentTime - lastSampleTime)
                / 1000.0;
163
164         // Cálculo de RPM para motor esquerdo
165         float leftPulsesPerSecond = (float)leftTicks /
                elapsedTimeSeconds;
166         float leftRPM = (leftPulsesPerSecond / pulsesPerWheelRev) *
                60.0;
167
168         // Cálculo de RPM para motor direito
```

```
169         float rightPulsesPerSecond = (float)rightTicks /
               elapsedTimeSeconds;
170         float rightRPM = (rightPulsesPerSecond / pulsesPerWheelRev) *
               60.0;
171
172         // Aplica filtro de média móvel para motor esquerdo
173         rpmFilterLeft[filterIndexLeft] = leftRPM;
174         filterIndexLeft = (filterIndexLeft + 1) % filterSize;
175
176         float filteredRPMLeft = 0;
177         for (int i = 0; i < filterSize; i++) {
178                 filteredRPMLeft += rpmFilterLeft[i];
179         }
180         filteredRPMLeft /= filterSize;
181
182         // Aplica filtro de média móvel para motor direito
183         rpmFilterRight[filterIndexRight] = rightRPM;
184         filterIndexRight = (filterIndexRight + 1) % filterSize;
185
186         float filteredRPMRight = 0;
187         for (int i = 0; i < filterSize; i++) {
188                 filteredRPMRight += rpmFilterRight[i];
189         }
190         filteredRPMRight /= filterSize;
191
192         // Armazena leitura filtrada no buffer para detecção de regime
               permanente
193         if (testPhase == 0) {
194                 rpmReadingsLeft[readingIndexLeft] = filteredRPMLeft;
195                 readingIndexLeft = (readingIndexLeft + 1) %
                       numReadingsForSteady;
196
197                 rpmReadingsRight[readingIndexRight] = filteredRPMRight;
198                 readingIndexRight = (readingIndexRight + 1) %
                       numReadingsForSteady;
199         }
200
201         // Armazena dados completos para análise (transitório e tau)
202         if (testPhase == 0 || (testPhase >= 1 && testPhase <=
               numTauMeasurements)) {
203                 if (dataIndexLeft < maxDataPoints) {
204                 timeDataLeft[dataIndexLeft] = (float)(currentTime -
                       startStepTime);
205                 rpmDataLeft[dataIndexLeft] = leftRPM;
206                 dataIndexLeft++;
207                 }
208                 if (dataIndexRight < maxDataPoints) {
209                 timeDataRight[dataIndexRight] = (float)(currentTime -
```

```
                    startStepTime);
210                 rpmDataRight[dataIndexRight] = rightRPM;
211                 dataIndexRight++;
212                 }
213         }
214
215         // Imprime dados em formato CSV
216         if (currentTime - lastPrintTime >= printInterval) {
217                 Serial.print(currentTime - startStepTime);
218                 Serial.print(F(","));
219                 Serial.print(filteredRPMLeft, 2);
220                 Serial.print(F(","));
221                 Serial.print(filteredRPMRight, 2);
222                 Serial.print(F(","));
223                 Serial.println(testPWM);
224                 lastPrintTime = currentTime;
225         }
226
227         // Verifica se o sistema atingiu regime permanente
228         if (testPhase == 0 && !steadyStateDetected && readingIndexLeft
                == 0 && currentTime - startStepTime > 3000)
229              {
230                 // Verifica se as leituras estão estáveis usando desvio
                       padrão - Motor Esquerdo
231                 float sumLeft = 0;
232                 float sumSqLeft = 0;
233
234                 for (int i = 0; i < numReadingsForSteady; i++) {
235                 sumLeft += rpmReadingsLeft[i];
236                 }
237
238                 float meanLeft = sumLeft / numReadingsForSteady;
239
240                 for (int i = 0; i < numReadingsForSteady; i++) {
241                 sumSqLeft += (rpmReadingsLeft[i] - meanLeft) * (
                       rpmReadingsLeft[i] - meanLeft);
242                 }
243
244                 float stdDevLeft = sqrt(sumSqLeft / numReadingsForSteady
                       );
245                 float coeffVarLeft = (stdDevLeft / meanLeft) * 100.0;
246
247                 // Verifica se as leituras estão estáveis usando desvio
                       padrão - Motor Direito
248                 float sumRight = 0;
249                 float sumSqRight = 0;
250
251                 for (int i = 0; i < numReadingsForSteady; i++) {
```

```
252                 sumRight += rpmReadingsRight[i];
253                 }
254
255                 float meanRight = sumRight / numReadingsForSteady;
256
257                 for (int i = 0; i < numReadingsForSteady; i++) {
258                 sumSqRight += (rpmReadingsRight[i] - meanRight) * (
                        rpmReadingsRight[i] - meanRight);
259                 }
260
261                 float stdDevRight = sqrt(sumSqRight /
                        numReadingsForSteady);
262                 float coeffVarRight = (stdDevRight / meanRight) * 100.0;
263
264                 // Se o coeficiente de variação for menor que 3% para
                        ambos os motores,
265                 //considera em regime permanente
266                 if (coeffVarLeft < 3.0 && coeffVarRight < 3.0) {
267                 steadyStateRPMLeft = meanLeft;
268                 thresholdRPMLeft = steadyStateRPMLeft * 0.63;
269
270                 steadyStateRPMRight = meanRight;
271                 thresholdRPMRight = steadyStateRPMRight * 0.63;
272
273                 steadyStateDetected = true;
274
275                 Serial.println(F("\n--- Regime permanente detectado ---"
                        ));
276
277                 Serial.print(F("Motor Esquerdo - RPM em regime: "));
278                 Serial.println(steadyStateRPMLeft, 2);
279                 Serial.print(F("Motor Esquerdo - Desvio padrão: "));
280                 Serial.println(stdDevLeft, 2);
281                 Serial.print(F("Motor Esquerdo - Coeficiente de variação
                        : "));
282                 Serial.print(coeffVarLeft, 2);
283                 Serial.println(F("%"));
284                 Serial.print(F("Motor Esquerdo - Limiar para tau (63%):
                        "));
285                 Serial.println(thresholdRPMLeft, 2);
286
287                 Serial.print(F("Motor Direito - RPM em regime: "));
288                 Serial.println(steadyStateRPMRight, 2);
289                 Serial.print(F("Motor Direito - Desvio padrão: "));
290                 Serial.println(stdDevRight, 2);
291                 Serial.print(F("Motor Direito - Coeficiente de variação:
                         "));
292                 Serial.print(coeffVarRight, 2);
```

```
293                 Serial.println(F("%"));
294                 Serial.print(F("Motor Direito - Limiar para tau (63%): "
                        ));
295                 Serial.println(thresholdRPMRight, 2);
296
297                 Serial.println(F("Preparando para medições de tau e
                        tau_motor..."));
298
299                 // Calcula tau_motor com os dados coletados durante a
                        subida
300                 calculateTauMotor();
301
302                 // Prepara para a fase de medição de tau
303                 delay(1000);
304                 startTestPhase(1);
305                 }
306         }
307
308         lastSampleTime = currentTime;
309         }
310
311         // Verifica timeout para cada fase
312         if ((testPhase == 0 && currentTime - startStepTime > 20000) ||
313                 (testPhase >= 1 && testPhase <= numTauMeasurements &&
                        currentTime - startStepTime > 10000)) {
314
315         if (testPhase == 0) {
316                 // Timeout na detecção de regime permanente
317                 Serial.println(F("\n--- Timeout na detecção de regime
                        permanente ---"));
318                 Serial.println(F("Usando último valor médio como
                        referÃªncia."));
319
320                 // Cálculo para motor esquerdo
321                 float sumLeft = 0;
322                 for (int i = 0; i < numReadingsForSteady; i++) {
323                 sumLeft += rpmReadingsLeft[i];
324                 }
325                 steadyStateRPMLeft = sumLeft / numReadingsForSteady;
326                 thresholdRPMLeft = steadyStateRPMLeft * 0.63;
327
328                 // Cálculo para motor direito
329                 float sumRight = 0;
330                 for (int i = 0; i < numReadingsForSteady; i++) {
331                 sumRight += rpmReadingsRight[i];
332                 }
333                 steadyStateRPMRight = sumRight / numReadingsForSteady;
334                 thresholdRPMRight = steadyStateRPMRight * 0.63;
```

```
335
336                 // Tenta calcular tau_motor mesmo com timeout
337                 calculateTauMotor();
338
339                 startTestPhase(1);
340         } else {
341                 // Timeout na medição de tau atual
342                 Serial.println(F("\n--- Timeout na medição de tau ---"))
                        ;
343
344                 // Se temos dados suficientes, tenta calcular tau mesmo
                        assim - Motor Esquerdo
345                 if (dataIndexLeft > 10) {
346                 float tauLeft = calculateTauFromData(true);
347                 tauMeasurementsLeft[currentTauMeasurement - 1] = tauLeft
                        ;
348
349                 Serial.print(F("Motor Esquerdo - Tau estimado da medição
                         "));
350                 Serial.print(currentTauMeasurement);
351                 Serial.print(F(": "));
352                 Serial.print(tauLeft);
353                 Serial.println(F(" ms"));
354                 } else {
355                 // Marca como medição inválida
356                 tauMeasurementsLeft[currentTauMeasurement - 1] = -1;
357                 }
358
359                 // Se temos dados suficientes, tenta calcular tau mesmo
                        assim - Motor Direito
360                 if (dataIndexRight > 10) {
361                 float tauRight = calculateTauFromData(false);
362                 tauMeasurementsRight[currentTauMeasurement - 1] =
                        tauRight;
363
364                 Serial.print(F("Motor Direito - Tau estimado da medição
                        "));
365                 Serial.print(currentTauMeasurement);
366                 Serial.print(F(": "));
367                 Serial.print(tauRight);
368                 Serial.println(F(" ms"));
369                 } else {
370                 // Marca como medição inválida
371                 tauMeasurementsRight[currentTauMeasurement - 1] = -1;
372                 }
373
374                 // Avança para próxima medição ou finaliza
375                 if (currentTauMeasurement < numTauMeasurements) {
```

```
376                 startTestPhase(testPhase + 1);
377                 } else {
378                 startTestPhase(numTauMeasurements + 2); // Pula para
                        fase final
379                 }
380         }
381         }
382 }
383
384 // Função para calcular o tempo motor theta (tau_motor) a partir dos
        dados coletados
385 void calculateTauMotor() {
386         // Para motor esquerdo
387         float thresholdRPMLeft = steadyStateRPMLeft * 0.63;
388         tauMotorLeft = 0; // Inicializa com 0
389
390         // Encontra o tempo onde o RPM atinge 63% do valor final nos
            dados coletados
391         for (int i = 0; i < dataIndexLeft; i++) {
392         if (rpmDataLeft[i] >= thresholdRPMLeft) {
393                 tauMotorLeft = timeDataLeft[i]; // Usa os dados
                        temporais
394                 break;
395         }
396         }
397
398         // Para motor direito
399         float thresholdRPMRight = steadyStateRPMRight * 0.63;
400         tauMotorRight = 0; // Inicializa com 0
401
402         // Encontra o tempo onde o RPM atinge 63% do valor final
403         for (int i = 0; i < dataIndexRight; i++) {
404         if (rpmDataRight[i] >= thresholdRPMRight) {
405                 tauMotorRight = timeDataRight[i]; // Usa os dados
                        temporais
406                 break;
407         }
408         }
409
410         // Armazena os valores calculados
411         if (currentTauMeasurement >= 0 && currentTauMeasurement <
            numTauMeasurements) {
412         tauMotorMeasurementsLeft[currentTauMeasurement] = tauMotorLeft;
413         tauMotorMeasurementsRight[currentTauMeasurement] = tauMotorRight
                ;
414         }
415
416         tauMotorCalculated = true;
```

```
417
418         // Exibe os resultados
419         Serial.println(F("\n--- Resultados do Tempo Motor Theta (
                Tau_motor) ---"));
420         Serial.print(F("Motor Esquerdo - Tau_motor: "));
421         Serial.print(tauMotorLeft);
422         Serial.println(F(" ms"));
423
424         Serial.print(F("Motor Direito - Tau_motor: "));
425         Serial.print(tauMotorRight);
426         Serial.println(F(" ms"));
427 }
428
429 // Inicia uma nova fase de teste
430 void startTestPhase(int phase) {
431         testPhase = phase;
432
433         if (phase == 0) {
434         // Fase inicial - detectar regime permanente
435         Serial.println(F("\n--- Iniciando detecção de regime permanente
                ---"));
436         Serial.println(F("Tempo,RPM_Esquerdo,RPM_Direito,PWM"));
437
438         // Limpa os buffers - Motor Esquerdo
439         for (int i = 0; i < numReadingsForSteady; i++) {
440                 rpmReadingsLeft[i] = 0;
441         }
442         for (int i = 0; i < filterSize; i++) {
443                 rpmFilterLeft[i] = 0;
444         }
445         readingIndexLeft = 0;
446         filterIndexLeft = 0;
447
448         // Limpa os buffers - Motor Direito
449         for (int i = 0; i < numReadingsForSteady; i++) {
450                 rpmReadingsRight[i] = 0;
451         }
452         for (int i = 0; i < filterSize; i++) {
453                 rpmFilterRight[i] = 0;
454         }
455         readingIndexRight = 0;
456         filterIndexRight = 0;
457
458         // Limpa os buffers de dados completos
459         dataIndexLeft = 0;
460         dataIndexRight = 0;
461         for (int i = 0; i < maxDataPoints; i++) {
462                 timeDataLeft[i] = 0;
```

```
463                 timeDataRight[i] = 0;
464                 rpmDataLeft[i] = 0;
465                 rpmDataRight[i] = 0;
466         }
467
468         steadyStateDetected = false;
469         tauMotorCalculated = false;
470
471         // Reseta os contadores de posição dos encoders em quadratura
472         noInterrupts();
473         encoderPositionLeft = 0;
474         encoderPositionRight = 0;
475         lastEncoderPositionLeft = 0;
476         lastEncoderPositionRight = 0;
477         interrupts();
478
479         // Aplica o degrau de PWM
480         startStepTime = millis();
481         motorLeft.run(FORWARD);
482         motorRight.run(FORWARD);
483         motorLeft.setSpeed(testPWM);
484         motorRight.setSpeed(testPWM);
485
486         } else if (phase >= 1 && phase <= numTauMeasurements) {
487         // Fase de medição de tau (múltiplas medições)
488         currentTauMeasurement = phase;
489
490         Serial.print(F("\n--- Iniciando medição de tau #"));
491         Serial.print(currentTauMeasurement);
492         Serial.println(F(" ---"));
493
494         // Para os motores
495         motorLeft.setSpeed(0);
496         motorLeft.run(RELEASE);
497         motorRight.setSpeed(0);
498         motorRight.run(RELEASE);
499
500         delay(3000); // Espera o motor parar completamente
501
502         // Limpa os contadores e buffers
503         noInterrupts();
504         encoderPositionLeft = 0;
505         encoderPositionRight = 0;
506         lastEncoderPositionLeft = 0;
507         lastEncoderPositionRight = 0;
508         interrupts();
509
510         // Limpa filtros - Motor Esquerdo
```

```
511         for (int i = 0; i < filterSize; i++) {
512                 rpmFilterLeft[i] = 0;
513         }
514         filterIndexLeft = 0;
515
516         // Limpa filtros - Motor Direito
517         for (int i = 0; i < filterSize; i++) {
518                 rpmFilterRight[i] = 0;
519         }
520         filterIndexRight = 0;
521
522         // Limpa buffer de dados
523         dataIndexLeft = 0;
524         dataIndexRight = 0;
525         for (int i = 0; i < maxDataPoints; i++) {
526                 timeDataLeft[i] = 0;
527                 timeDataRight[i] = 0;
528                 rpmDataLeft[i] = 0;
529                 rpmDataRight[i] = 0;
530         }
531
532         Serial.println(F("Tempo,RPM_Esquerdo,RPM_Direito,PWM"));
533
534         // Aplica novo degrau para medir tau
535         startStepTime = millis();
536         motorLeft.run(FORWARD);
537         motorRight.run(FORWARD);
538         motorLeft.setSpeed(testPWM);
539         motorRight.setSpeed(testPWM);
540
541         } else if (phase == numTauMeasurements + 2) {
542         // Fase final - motores parados
543         motorLeft.setSpeed(0);
544         motorLeft.run(RELEASE);
545         motorRight.setSpeed(0);
546         motorRight.run(RELEASE);
547         }
548 }
549
550 // Calcula tau a partir dos dados coletados
551 // isLeft = true para motor esquerdo, false para motor direito
552 float calculateTauFromData(bool isLeft) {
553         // Primeiro encontra o ponto mais próximo do limiar de 63%
554         int thresholdIndex = -1;
555         float minDiff = 1000.0;
556         float thresholdRPM = isLeft ? thresholdRPMLeft :
                thresholdRPMRight;
557
```

```
558         for (int i = 0; i < (isLeft ? dataIndexLeft : dataIndexRight); i
               ++) {
559         float rpmValue = isLeft ? rpmDataLeft[i] : rpmDataRight[i];
560         float diff = abs(rpmValue - thresholdRPM);
561         if (diff < minDiff) {
562                 minDiff = diff;
563                 thresholdIndex = i;
564         }
565         }
566
567         if (thresholdIndex == -1) {
568         return -1; // Não foi possível encontrar o ponto
569         }
570
571         // Retorna o tempo correspondente ao índice encontrado
572         return isLeft ? timeDataLeft[thresholdIndex] : timeDataRight[
               thresholdIndex];
573 }
574
575 // Função para calcular desvio padrão com correção de Bessel
576 float calculateStdDev(float measurements[], int numMeasurements, float
       mean) {
577         int validMeasurements = 0;
578         float sumSquaredDiff = 0.0;
579
580         // Conta medições válidas e calcula soma dos quadrados das
               diferenças
581         for (int i = 0; i < numMeasurements; i++) {
582         if (measurements[i] > 0) {
583                 float diff = measurements[i] - mean;
584                 sumSquaredDiff += diff * diff;
585                 validMeasurements++;
586         }
587         }
588
589         // Verifica se há amostras suficientes para cálculo
590         if (validMeasurements <= 1) return 0.01; // Valor mínimo para
               evitar zero
591
592         // Usa (n-1) para correção de Bessel
593         return sqrt(sumSquaredDiff / (validMeasurements - 1));
594 }
595
596 void calculateAndPrintFinalResults() {
597         Serial.println(F("\n====== RESULTADOS FINAIS ======"));
598
599         // RESULTADOS PARA MOTOR ESQUERDO
600         Serial.println(F("\n--- MOTOR ESQUERDO ---"));
```

```
601         Serial.print(F("RPM em regime permanente: "));
602         Serial.print(steadyStateRPMLeft, 2);
603         Serial.println(F(" RPM"));
604 
605         // Exibe o tau_motor calculado
606         if (tauMotorCalculated) {
607         Serial.print(F("Tempo motor theta (tau_motor): "));
608         Serial.print(tauMotorLeft);
609         Serial.println(F(" ms"));
610         } else {
611         Serial.println(F("Tempo motor theta (tau_motor): Não calculado")
               );
612         }
613 
614         // Conta medições válidas e calcula média - Motor Esquerdo
615         int validMeasurementsLeft = 0;
616         float tauSumLeft = 0;
617         float tauMinLeft = 10000;
618         float tauMaxLeft = 0;
619 
620         for (int i = 0; i < numTauMeasurements; i++) {
621         if (tauMeasurementsLeft[i] > 0) {
622                 tauSumLeft += tauMeasurementsLeft[i];
623                 validMeasurementsLeft++;
624 
625                 tauMinLeft = min(tauMinLeft, tauMeasurementsLeft[i]);
626                 tauMaxLeft = max(tauMaxLeft, tauMeasurementsLeft[i]);
627         }
628         }
629 
630         if (validMeasurementsLeft > 0) {
631         float tauAvgLeft = tauSumLeft / validMeasurementsLeft;
632         float tauStdDevLeft = calculateStdDev(tauMeasurementsLeft,
               numTauMeasurements, tauAvgLeft);
633 
634         // Imprime resultados estatísticos - Motor Esquerdo
635         Serial.println(F("\nMedições individuais de tau:"));
636         for (int i = 0; i < numTauMeasurements; i++) {
637                 Serial.print(F("Medição #"));
638                 Serial.print(i + 1);
639                 Serial.print(F(": "));
640                 if (tauMeasurementsLeft[i] > 0) {
641                 Serial.print(tauMeasurementsLeft[i], 2);
642                 Serial.println(F(" ms"));
643                 } else {
644                 Serial.println(F("Falhou"));
645                 }
646         }
```

```
647
648         Serial.print(F("\nConstante de tempo média (tau): "));
649         Serial.print(tauAvgLeft, 2);
650         Serial.println(F(" ms"));
651
652         Serial.print(F("Desvio padrão: "));
653         Serial.print(tauStdDevLeft, 2);
654         Serial.println(F(" ms"));
655
656         Serial.print(F("Coeficiente de variação: "));
657         Serial.print((tauStdDevLeft / tauAvgLeft) * 100.0, 2);
658         Serial.println(F("%"));
659
660         Serial.print(F("Valor mínimo: "));
661         Serial.print(tauMinLeft, 2);
662         Serial.println(F(" ms"));
663
664         Serial.print(F("Valor máximo: "));
665         Serial.print(tauMaxLeft, 2);
666         Serial.println(F(" ms"));
667
668         // Estimativas adicionais baseadas no modelo de primeira ordem
669         Serial.print(F("\nTempo para 95% do valor final (3*tau): "));
670         Serial.print(tauAvgLeft * 3, 2);
671         Serial.println(F(" ms"));
672
673         Serial.print(F("Tempo para 98% do valor final (4*tau): "));
674         Serial.print(tauAvgLeft * 4, 2);
675         Serial.println(F(" ms"));
676
677         // Cálculo do ganho do sistema
678         float ganhoSistemaLeft = steadyStateRPMLeft / testPWM;
679         Serial.print(F("\nGanho do sistema: "));
680         Serial.print(ganhoSistemaLeft, 4);
681         Serial.println(F(" RPM/PWM"));
682
683         // Equação da função de transferÃªncia
684         Serial.println(F("\nFunção de transferÃªncia estimada (modelo de
                primeira ordem):"));
685         Serial.print(F("G(s) = "));
686         Serial.print(ganhoSistemaLeft, 4);
687         Serial.print(F(" / ("));
688         Serial.print((float)tauAvgLeft / 1000.0, 3);
689         Serial.println(F("s + 1)"));
690         } else {
691         Serial.println(F("Não foi possível obter medições válidas de tau
              ."));
692         }
```

```
693
694         // RESULTADOS PARA MOTOR DIREITO
695         Serial.println(F("\n--- MOTOR DIREITO ---"));
696         Serial.print(F("RPM em regime permanente: "));
697         Serial.print(steadyStateRPMRight, 2);
698         Serial.println(F(" RPM"));
699
700         // Exibe o tau_motor calculado
701         if (tauMotorCalculated) {
702         Serial.print(F("Tempo motor theta (tau_motor): "));
703         Serial.print(tauMotorRight);
704         Serial.println(F(" ms"));
705         } else {
706         Serial.println(F("Tempo motor theta (tau_motor): Não calculado")
              );
707         }
708
709         // Conta medições válidas e calcula média - Motor Direito
710         int validMeasurementsRight = 0;
711         float tauSumRight = 0;
712         float tauMinRight = 10000;
713         float tauMaxRight = 0;
714
715         for (int i = 0; i < numTauMeasurements; i++) {
716         if (tauMeasurementsRight[i] > 0) {
717                 tauSumRight += tauMeasurementsRight[i];
718                 validMeasurementsRight++;
719
720                 tauMinRight = min(tauMinRight, tauMeasurementsRight[i]);
721                 tauMaxRight = max(tauMaxRight, tauMeasurementsRight[i]);
722         }
723         }
724
725         if (validMeasurementsRight > 0) {
726         float tauAvgRight = tauSumRight / validMeasurementsRight;
727         float tauStdDevRight = calculateStdDev(tauMeasurementsRight,
              numTauMeasurements, tauAvgRight);
728
729         // Imprime resultados estatísticos - Motor Direito
730         Serial.println(F("\nMedições individuais de tau:"));
731         for (int i = 0; i < numTauMeasurements; i++) {
732                 Serial.print(F("Medição #"));
733                 Serial.print(i + 1);
734                 Serial.print(F(": "));
735                 if (tauMeasurementsRight[i] > 0) {
736                 Serial.print(tauMeasurementsRight[i], 2);
737                 Serial.println(F(" ms"));
738                 } else {
```

```
739                 Serial.println(F("Falhou"));
740                 }
741         }
742
743         Serial.print(F("\nConstante de tempo média (tau): "));
744         Serial.print(tauAvgRight, 2);
745         Serial.println(F(" ms"));
746
747         Serial.print(F("Desvio padrão: "));
748         Serial.print(tauStdDevRight, 2);
749         Serial.println(F(" ms"));
750
751         Serial.print(F("Coeficiente de variação: "));
752         Serial.print((tauStdDevRight / tauAvgRight) * 100.0, 2);
753         Serial.println(F("%"));
754
755         Serial.print(F("Valor mínimo: "));
756         Serial.print(tauMinRight, 2);
757         Serial.println(F(" ms"));
758
759         Serial.print(F("Valor máximo: "));
760         Serial.print(tauMaxRight, 2);
761         Serial.println(F(" ms"));
762
763         // Estimativas adicionais baseadas no modelo de primeira ordem
764         Serial.print(F("\nTempo para 95% do valor final (3*tau): "));
765         Serial.print(tauAvgRight * 3, 2);
766         Serial.println(F(" ms"));
767
768         Serial.print(F("Tempo para 98% do valor final (4*tau): "));
769         Serial.print(tauAvgRight * 4, 2);
770         Serial.println(F(" ms"));
771
772         // Cálculo do ganho do sistema
773         float ganhoSistemaRight = steadyStateRPMRight / testPWM;
774         Serial.print(F("\nGanho do sistema: "));
775         Serial.print(ganhoSistemaRight, 4);
776         Serial.println(F(" RPM/PWM"));
777
778         // Equação da função de transferÃªncia
779         Serial.println(F("\nFunção de transferÃªncia estimada (modelo de
                primeira ordem):"));
780         Serial.print(F("G(s) = "));
781         Serial.print(ganhoSistemaRight, 4);
782         Serial.print(F(" / ("));
783         Serial.print((float)tauAvgRight / 1000.0, 3);
784         Serial.println(F("s + 1)"));
785         } else {
```

```
786         Serial.println(F("Não foi possível obter medições válidas de tau
                .."));
787         }
788
789         Serial.println(F("\n====== FIM DO TESTE ======"));
790         Serial.println(F("Pressione RESET para iniciar um novo teste."))
                ;
791 }
792
793 // Rotina de interrupção para o encoder do motor esquerdo usando
        quadratura
794 void encoderLeftISR() {
795         // LÃª o estado atual dos pinos A e B (concatenando em 2 bits)
796         byte encoderStateNew = (digitalRead(encoderPinLeftA) << 1) |
                digitalRead(encoderPinLeftB);
797
798         // Decodifica o padrão de quadratura
799         if (encoderStateLeftOld == 0b00) {
800         if (encoderStateNew == 0b01) encoderPositionLeft++;
801         if (encoderStateNew == 0b10) encoderPositionLeft--;
802         } else if (encoderStateLeftOld == 0b01) {
803         if (encoderStateNew == 0b11) encoderPositionLeft++;
804         if (encoderStateNew == 0b00) encoderPositionLeft--;
805         } else if (encoderStateLeftOld == 0b11) {
806         if (encoderStateNew == 0b10) encoderPositionLeft++;
807         if (encoderStateNew == 0b01) encoderPositionLeft--;
808         } else if (encoderStateLeftOld == 0b10) {
809         if (encoderStateNew == 0b00) encoderPositionLeft++;
810         if (encoderStateNew == 0b11) encoderPositionLeft--;
811         }
812
813         // Atualiza o estado anterior
814         encoderStateLeftOld = encoderStateNew;
815 }
816
817 // Rotina de interrupção para o encoder do motor direito usando
        quadratura
818 void encoderRightISR() {
819         // LÃª o estado atual dos pinos A e B (concatenando em 2 bits)
820         byte encoderStateNew = (digitalRead(encoderPinRightA) << 1) |
                digitalRead(encoderPinRightB);
821
822         // Decodifica o padrão de quadratura
823         if (encoderStateRightOld == 0b00) {
824         if (encoderStateNew == 0b01) encoderPositionRight++;
825         if (encoderStateNew == 0b10) encoderPositionRight--;
826         } else if (encoderStateRightOld == 0b01) {
827         if (encoderStateNew == 0b11) encoderPositionRight++;
```

```
828         if (encoderStateNew == 0b00) encoderPositionRight--;
829         } else if (encoderStateRightOld == 0b11) {
830         if (encoderStateNew == 0b10) encoderPositionRight++;
831         if (encoderStateNew == 0b01) encoderPositionRight--;
832         } else if (encoderStateRightOld == 0b10) {
833         if (encoderStateNew == 0b00) encoderPositionRight++;
834         if (encoderStateNew == 0b11) encoderPositionRight--;
835         }
836
837         // Atualiza o estado anterior
838         encoderStateRightOld = encoderStateNew;
839 }
```

APÊNDICE B – Apêndice B

Código usado nos testes de validação do controlador PID

```
#include <AFMotor.h>
#include <PID_v1.h>
// codigo 2, foi utilizado para medir o comportamento da planta ao receber o sinal do controlador, alem de permitir o refinamento apos
// Definições dos pinos dos encoders
const int encoderPinLeft = 20;
const int encoderPinRight = 19;

// Motores
AF_DCMotor motorLeft(3);
AF_DCMotor motorRight(4);

// Variáveis para os encoders
volatile unsigned long pulseCountLeft = 0, lastPulseCountLeft = 0;
volatile unsigned long pulseCountRight = 0, lastPulseCountRight = 0;

// Variáveis para velocidade (RPM)
double rpmLeft = 0, rpmRight = 0;

// Variáveis PID
double SetpointLeft = 0, InputLeft = 0, OutputLeft;
double SetpointRight = 0, InputRight = 0, OutputRight;

        // //Parametros PID z-n - Controlado calculado pelo metodo z-n, mas com ajuste fino!!
// double Kp1 = 12.350, Ki1 = 31.07, Kd1 = 1.23;
// double Kp2 = 12.350, Ki2 = 36.31, Kd2 = 1.05;

// //Parametros PID metodo IMC - Controlador mais conservador e bem mais lento. Porem funcional!!
// double Kp1 = 13.26, Ki1 = 53.04, Kd1 = 0;
// double Kp2 = 13.26, Ki2 = 53.04, Kd2 = 0;

// // //Parametros PID z-n - Controlado calculado pelo metodo z-n!! Estao otimos
double Kp1 = 12.35, Ki1 = 31.07, Kd1 = 0.63;
double Kp2 = 12.35, Ki2 = 38.55, Kd2 = 0.68;


// ParÃ¢metros do encoder (ajuste conforme necessário)
const float pulsesPerRev = 380.0;
const float gearReduction = 7.0;
```

```
39
40 // Contadores de tempo
41 unsigned long previousTime = 0;
42 const unsigned long sampleTime = 120;
43
44 // Teste automático
45 boolean testRunning = false;
46 unsigned long testStartTime = 0;
47 const unsigned long testDuration = 5000;
48 const int testSpeedLeft = 100;
49 const int testSpeedRight = 100;
50
51 // InstÃ¢ncias PID
52 PID pidLeft(&InputLeft, &OutputLeft, &SetpointLeft, Kp1, Ki1, Kd1,
       DIRECT);
53 PID pidRight(&InputRight, &OutputRight, &SetpointRight, Kp2, Ki2, Kd2,
       DIRECT);
54
55 void countPulseLeft() {
56         pulseCountLeft++;
57         if (pulseCountLeft > 1000000) pulseCountLeft = 0;
58 }
59
60 void countPulseRight() {
61         pulseCountRight++;
62         if (pulseCountRight > 1000000) pulseCountRight = 0;
63 }
64
65 void setup() {
66         Serial.begin(115200);
67         Serial.println("Teste do PID para motores");
68
69         pinMode(encoderPinLeft, INPUT);
70         pinMode(encoderPinRight, INPUT);
71         attachInterrupt(digitalPinToInterrupt(encoderPinLeft),
              countPulseLeft, RISING);
72         attachInterrupt(digitalPinToInterrupt(encoderPinRight),
              countPulseRight, RISING);
73
74         motorLeft.setSpeed(0);
75         motorLeft.run(RELEASE);
76         motorRight.setSpeed(0);
77         motorRight.run(RELEASE);
78
79         pidLeft.SetMode(AUTOMATIC);
80         pidLeft.SetOutputLimits(0, 255);
81         pidLeft.SetSampleTime(sampleTime);
82
```

```
83          pidRight.SetMode(AUTOMATIC);
84          pidRight.SetOutputLimits(0, 255);
85          pidRight.SetSampleTime(sampleTime);
86
87          Serial.println("Sistema inicializado. Envie comandos via serial.
                ");
88  }
89
90  void loop() {
91          unsigned long currentTime = millis();
92
93          // Leitura de comandos via serial
94          if (Serial.available() > 0) {
95          String command = Serial.readStringUntil('\n');
96          command.trim();
97
98          if (command == "start") {
99                  SetpointLeft = testSpeedLeft;
100                 SetpointRight = testSpeedRight;
101                 testRunning = true;
102                 testStartTime = currentTime;
103                 Serial.println("TEST_STARTED"); // Confirmação para o
                        Python
104                 Serial.println("Iniciando teste dos motores.");
105         } else if (command.startsWith("speed=")) {
106                 int speed = command.substring(6).toInt();
107                 SetpointLeft = speed;
108                 SetpointRight = speed;
109                 Serial.println("TEST_STARTED"); // Confirmação para o
                        Python
110                 Serial.println("Velocidade ajustada para " + String(
                        speed));
111         } else if (command == "stop") {
112                 SetpointLeft = 0;
113                 SetpointRight = 0;
114                 testRunning = false;
115                 Serial.println("Parando motores.");
116         } else if (command.startsWith("kp1=")) {
117                 Kp1 = command.substring(4).toFloat();
118                 pidLeft.SetTunings(Kp1, Ki1, Kd1);
119         } else if (command.startsWith("ki1=")) {
120                 Ki1 = command.substring(4).toFloat();
121                 pidLeft.SetTunings(Kp1, Ki1, Kd1);
122         } else if (command.startsWith("kd1=")) {
123                 Kd1 = command.substring(4).toFloat();
124                 pidLeft.SetTunings(Kp1, Ki1, Kd1);
125         } else if (command.startsWith("kp2=")) {
126                 Kp2 = command.substring(4).toFloat();
```

```
127                 pidRight.SetTunings(Kp2, Ki2, Kd2);
128         } else if (command.startsWith("ki2=")) {
129                 Ki2 = command.substring(4).toFloat();
130                 pidRight.SetTunings(Kp2, Ki2, Kd2);
131         } else if (command.startsWith("kd2=")) {
132                 Kd2 = command.substring(4).toFloat();
133                 pidRight.SetTunings(Kp2, Ki2, Kd2);
134         } else if (command.startsWith("speedL=")) {
135                 SetpointLeft = command.substring(7).toInt();
136         } else if (command.startsWith("speedR=")) {
137                 SetpointRight = command.substring(7).toInt();
138         } else if (command == "a") {
139                 SetpointLeft = 12;
140                 SetpointRight = 12;
141                 Serial.println("TEST_STARTED"); // Confirmação para o
                        Python
142                 delay(100); // Pequena pausa para garantir que a
                        confirmação seja enviada
143                 Serial.println("Velocidade ajustada para 12.");
144         }
145         }
146
147         // Finaliza teste automático
148         if (testRunning && currentTime - testStartTime >= testDuration)
                {
149         SetpointLeft = 0;
150         SetpointRight = 0;
151         testRunning = false;
152         Serial.println("Teste concluído.");
153         }
154
155         // Cálculo periódico de RPM e controle PID
156         if (currentTime - previousTime >= sampleTime) {
157         previousTime = currentTime;
158
159         // Encoder esquerdo
160         detachInterrupt(digitalPinToInterrupt(encoderPinLeft));
161         unsigned long delta1 = pulseCountLeft - lastPulseCountLeft;
162         lastPulseCountLeft = pulseCountLeft;
163         attachInterrupt(digitalPinToInterrupt(encoderPinLeft),
                countPulseLeft, RISING);
164
165         // Encoder direito
166         detachInterrupt(digitalPinToInterrupt(encoderPinRight));
167         unsigned long delta2 = pulseCountRight - lastPulseCountRight;
168         lastPulseCountRight = pulseCountRight;
169         attachInterrupt(digitalPinToInterrupt(encoderPinRight),
                countPulseRight, RISING);
```

```
170
171         // Cálculo RPM
172         double pps1 = (delta1 * (1000.0 / sampleTime));
173         double pps2 = (delta2 * (1000.0 / sampleTime));
174
175         rpmLeft = (pps1 / gearReduction) * (60.0 / pulsesPerRev);
176         rpmRight = (pps2 / gearReduction) * (60.0 / pulsesPerRev);
177
178         InputLeft = rpmLeft;
179         InputRight = rpmRight;
180
181         // Controle Motor Esquerdo
182         if (SetpointLeft == 0) {
183                 pidLeft.SetMode(MANUAL);
184                 motorLeft.setSpeed(0);
185                 motorLeft.run(BRAKE);
186         } else {
187                 pidLeft.SetMode(AUTOMATIC);
188                 pidLeft.Compute();
189                 motorLeft.setSpeed(abs(OutputLeft));
190                 motorLeft.run(SetpointLeft > 0 ? FORWARD : BACKWARD);
191         }
192
193         // Controle Motor Direito
194         if (SetpointRight == 0) {
195                 pidRight.SetMode(MANUAL);
196                 motorRight.setSpeed(0);
197                 motorRight.run(BRAKE);
198         } else {
199                 pidRight.SetMode(AUTOMATIC);
200                 pidRight.Compute();
201                 motorRight.setSpeed(abs(OutputRight));
202                 motorRight.run(SetpointRight > 0 ? FORWARD : BACKWARD);
203         }
204
205         // Monitoramento via Serial
206         Serial.print("L: ");
207         Serial.print(SetpointLeft);
208         Serial.print(" | ");
209         Serial.print(rpmLeft);
210         Serial.print(" | ");
211         Serial.print(OutputLeft);
212         Serial.print(" || R: ");
213         Serial.print(SetpointRight);
214         Serial.print(" | ");
215         Serial.print(rpmRight);
216         Serial.print(" | ");
217         Serial.println(OutputRight);
```

```
218
219         // Serial Plotter (SetpointL, rpmL, SetpointR, rpmR)
220         Serial.print(SetpointLeft);
221         Serial.print(",");
222         Serial.print(rpmLeft);
223         Serial.print(",");
224         Serial.print(SetpointRight);
225         Serial.print(",");
226         Serial.println(rpmRight);
227         }
228 }
```

APÊNDICE C – Apêndice C

Este codigo faz parte da integração entre o controlador PID de baixo nivel (firmware) e o software de odometria.

```
#include <ros.h>
#include <std_msgs/Int32.h>
#include <geometry_msgs/Twist.h>
#include <AFMotor.h>
#include <PID_v1.h>

//este codigo deve ser utilizado no arduino como o controlador de baixo nivel.23/04/2025
// Definições dos pinos dos encoders
const int encoderPinLeftA = 19;
const int encoderPinLeftB = 18;
const int encoderPinRightA = 20;
const int encoderPinRightB = 21;

// Motores
AF_DCMotor motorLeft(4);
AF_DCMotor motorRight(3);

// Variáveis para os encoders
volatile long totalPulsesLeft = 0;
volatile long totalPulsesRight = 0;
volatile int lastEncodedLeft = 0;
volatile int lastEncodedRight = 0;
long lastPulseCountLeft = 0;
long lastPulseCountRight = 0;

// Variáveis para velocidade (RPM)
double rpmLeft = 0, rpmRight = 0;

// ParÃ¢metros físicos do robÃ´
const float wheelRadius = 0.035; // Raio da roda em metros (ajuste para o seu robÃ´)
const float wheelBase = 0.230;   // DistÃ¢ncia entre as rodas em metros (ajuste para o seu robÃ´)
const float TICKS_PER_REVOLUTION = 7.0 * 380.0; // PPR * redução
const float meterPerTick = (2.0 * PI * wheelRadius) / TICKS_PER_REVOLUTION;
const float radiansPerTick = meterPerTick / (wheelBase / 2.0);
const float motorCompensationFactor = 0.90; // Fator de compensação mais agressivo para o motor direito

```

```
37 // Variáveis PID para controle individual
38 double SetpointLeft = 0, InputLeft = 0, OutputLeft = 0;
39 double SetpointRight = 0, InputRight = 0, OutputRight = 0;
40 double absSetpointLeft = 0;
41 double absSetpointRight = 0;
42
43 // ParÃ¢metros PID para cada motor - ajustados para dar mais força ao
       motor esquerdo
44 double Kp1 = 12.0, Ki1 = 22.0, Kd1 = 0.60;  // Motor esquerdo
45 double Kp2 = 12.0, Ki2 = 23.8, Kd2 = 0.60;  // Motor direito
46
47 // ParÃ¢metros para sincronização
48 double syncError = 0;
49 double KsyncP = -0.1; // Valor negativo para compensar motor mais rápido
50 double KsyncI = -0.02; // Valor negativo para compensar motor mais rá
       pido
51 double syncErrorIntegral = 0;
52 double syncErrorPrev = 0;
53
54 // Variáveis para comando de velocidade
55 float linearVelocity = 0.0;  // m/s
56 float angularVelocity = 0.0; // rad/s
57
58 // Contadores de tempo
59 unsigned long previousTime = 0;
60 const unsigned long sampleTime = 50; // Intervalo de amostragem em ms
61
62 // ROS
63 ros::NodeHandle nh;
64
65 // Mensagens ROS
66 std_msgs::Int32 leftEncoderROS, rightEncoderROS;
67
68 // Publishers para encoders
69 ros::Publisher leftEncoderPub("left_encoder_ticks", &leftEncoderROS);
70 ros::Publisher rightEncoderPub("right_encoder_ticks", &rightEncoderROS);
71
72 // PID Controllers
73 PID pidLeft(&InputLeft, &OutputLeft, &absSetpointLeft, Kp1, Ki1, Kd1,
       DIRECT);
74 PID pidRight(&InputRight, &OutputRight, &absSetpointRight, Kp2, Ki2, Kd2
       , DIRECT);
75
76 // Callback para comandos de velocidade (similar ao TurtleBot 3)
77 void cmdVelCallback(const geometry_msgs::Twist& twist) {
78 // Armazena os comandos de velocidade
79 linearVelocity = twist.linear.x;   // Velocidade linear em m/s
80
```

```
81 // Amplificar a velocidade angular para aumentar o efeito de virar
82 angularVelocity = twist.angular.z * 3.0; // Multiplicar por 3 para
       aumentar o efeito
83
84 // Converte para velocidades das rodas (como no TurtleBot 3)
85 // v_l = v - (Ï  * L/2)
86 // v_r = v + (Ï  * L/2)
87 float leftWheelVelocity = linearVelocity - (angularVelocity * wheelBase
       / 2.0);
88 float rightWheelVelocity = linearVelocity + (angularVelocity * wheelBase
        / 2.0);
89
90 // Converte de m/s para RPM
91 // RPM = (v / (2Ï  * r)) * 60
92 float rpmFactor = 60.0 / (2.0 * PI * wheelRadius);
93
94 // Define os Setpoints em RPM
95 SetpointLeft = leftWheelVelocity * rpmFactor;
96 SetpointRight = rightWheelVelocity * rpmFactor;
97
98 // Calcula os valores absolutos para o PID
99 absSetpointLeft = abs(SetpointLeft);
100 absSetpointRight = abs(SetpointRight);
101
102 // Corrigido o log para evitar problemas com %f
103 char buffer[50];
104 nh.loginfo("CMD: lin=");
105 dtostrf(linearVelocity, 4, 2, buffer);
106 nh.loginfo(buffer);
107
108 nh.loginfo(" ang=");
109 dtostrf(angularVelocity, 4, 2, buffer);
110 nh.loginfo(buffer);
111
112 nh.loginfo(" leftW=");
113 dtostrf(leftWheelVelocity, 4, 2, buffer);
114 nh.loginfo(buffer);
115
116 nh.loginfo(" rightW=");
117 dtostrf(rightWheelVelocity, 4, 2, buffer);
118 nh.loginfo(buffer);
119 }
120
121 // Subscriber para comando de velocidade
122 ros::Subscriber<geometry_msgs::Twist> cmdVelSub("cmd_vel", &
       cmdVelCallback);
123
124 // Funções de interrupção para os encoders (mantidas as mesmas)
```

```
125 void updateEncoderLeft() {
126 int MSB = digitalRead(encoderPinLeftA);
127 int LSB = digitalRead(encoderPinLeftB);
128 int encoded = (MSB << 1) | LSB;
129 int sum = (lastEncodedLeft << 2) | encoded;
130
131 if (sum == 0b1101 || sum == 0b0100 || sum == 0b0010 || sum == 0b1011) {
132         totalPulsesLeft--;
133 }
134 if (sum == 0b1110 || sum == 0b0111 || sum == 0b0001 || sum == 0b1000) {
135         totalPulsesLeft++;
136 }
137
138 lastEncodedLeft = encoded;
139 }
140
141 void updateEncoderRight() {
142 int MSB = digitalRead(encoderPinRightA);
143 int LSB = digitalRead(encoderPinRightB);
144 int encoded = (MSB << 1) | LSB;
145 int sum = (lastEncodedRight << 2) | encoded;
146
147 if (sum == 0b1101 || sum == 0b0100 || sum == 0b0010 || sum == 0b1011) {
148         totalPulsesRight--;
149 }
150 if (sum == 0b1110 || sum == 0b0111 || sum == 0b0001 || sum == 0b1000) {
151         totalPulsesRight++;
152 }
153
154 lastEncodedRight = encoded;
155 }
156
157 void setup() {
158 // Configuração dos pinos dos encoders
159 pinMode(encoderPinLeftA, INPUT_PULLUP);
160 pinMode(encoderPinLeftB, INPUT_PULLUP);
161 pinMode(encoderPinRightA, INPUT_PULLUP);
162 pinMode(encoderPinRightB, INPUT_PULLUP);
163
164 // Configuração das interrupções
165 attachInterrupt(digitalPinToInterrupt(encoderPinLeftA),
       updateEncoderLeft, CHANGE);
166 attachInterrupt(digitalPinToInterrupt(encoderPinLeftB),
       updateEncoderLeft, CHANGE);
167 attachInterrupt(digitalPinToInterrupt(encoderPinRightA),
       updateEncoderRight, CHANGE);
168 attachInterrupt(digitalPinToInterrupt(encoderPinRightB),
       updateEncoderRight, CHANGE);
```

```
169
170 // Configuração dos motores
171 motorLeft.setSpeed(0);
172 motorLeft.run(RELEASE);
173 motorRight.setSpeed(0);
174 motorRight.run(RELEASE);
175
176 // Configuração do PID
177 pidLeft.SetMode(AUTOMATIC);
178 pidLeft.SetOutputLimits(0, 255);
179 pidRight.SetMode(AUTOMATIC);
180 pidRight.SetOutputLimits(0, 255);
181 pidLeft.SetSampleTime(sampleTime);
182 pidRight.SetSampleTime(sampleTime);
183
184 // Configuração do ROS
185 nh.getHardware()->setBaud(115200);
186 nh.initNode();
187 nh.advertise(leftEncoderPub);
188 nh.advertise(rightEncoderPub);
189 nh.subscribe(cmdVelSub);
190
191 // Log de inicialização
192 nh.loginfo("Firmware com controle Twist inicializada");
193
194 // Teste inicial dos motores para verificar conexões
195 nh.loginfo("Testando motores...");
196
197 // Teste do motor esquerdo
198 motorLeft.setSpeed(200);
199 motorLeft.run(FORWARD);
200 delay(500);
201 motorLeft.run(RELEASE);
202
203 // Pequena pausa
204 delay(500);
205
206 // Teste do motor direito
207 motorRight.setSpeed(200);
208 motorRight.run(FORWARD);
209 delay(500);
210 motorRight.run(RELEASE);
211
212 nh.loginfo("Teste dos motores concluído");
213 }
214
215 void loop() {
216 unsigned long currentTime = millis();
```

```
217
218 // Atualização do PID e controle dos motores
219 if (currentTime - previousTime >= sampleTime) {
220         // Calcular velocidade (RPM) com base nos encoders
221         noInterrupts();
222         long deltaLeft = totalPulsesLeft - lastPulseCountLeft;
223         long deltaRight = totalPulsesRight - lastPulseCountRight;
224         lastPulseCountLeft = totalPulsesLeft;
225         lastPulseCountRight = totalPulsesRight;
226         interrupts();
227
228         // Converter pulsos para RPM
229         double pps1 = (deltaLeft * (1000.0 / sampleTime)); // Pulsos por
              segundo
230         double pps2 = (deltaRight * (1000.0 / sampleTime)); // Pulsos
             por segundo
231         rpmLeft = (pps1 * 60.0) / TICKS_PER_REVOLUTION;
232         rpmRight = (pps2 * 60.0) / TICKS_PER_REVOLUTION;
233
234         // Calcular erro de sincronização (normalizado pela velocidade
             desejada)
235         // Modificado para incluir movimentos puramente angulares
236         if (abs(linearVelocity) > 0.01 || abs(angularVelocity) > 0.01) {
              // Incluindo movimento angular
237         // Calcular erro de sincronização proporcional Ã s velocidades
             desejadas
238         float targetRatio = abs(SetpointRight / SetpointLeft);
239         float actualRatio = abs(rpmRight / rpmLeft);
240
241         if (isnan(actualRatio) || isinf(actualRatio)) {
242                 actualRatio = 1.0; // Evitar divisão por zero
243         }
244
245         syncError = targetRatio - actualRatio;
246
247         // Integração do erro com anti-windup
248         syncErrorIntegral += syncError * (sampleTime / 1000.0);
249         syncErrorIntegral = constrain(syncErrorIntegral, -50, 50);
250         } else {
251         // Zerar erro quando parado
252         syncError = 0;
253         syncErrorIntegral = 0;
254         }
255
256         // Calcular correção de sincronização
257         double syncCorrection = KsyncP * syncError + KsyncI *
             syncErrorIntegral;
258
```

```
259         // Aplicar correção Ã  entrada do PID (valor absoluto da
                velocidade atual)
260         InputLeft = abs(rpmLeft);
261         InputRight = abs(rpmRight);
262
263         // Determinar direção dos motores
264         int signLeft = (SetpointLeft > 0) ? 1 : ((SetpointLeft < 0) ? -1
                : 0);
265         int signRight = (SetpointRight > 0) ? 1 : ((SetpointRight < 0) ?
                -1 : 0);
266
267         // Ativar ou desativar o PID baseado nos setpoints
268         pidLeft.SetMode(absSetpointLeft > 0 ? AUTOMATIC : MANUAL);
269         pidRight.SetMode(absSetpointRight > 0 ? AUTOMATIC : MANUAL);
270
271         // Calcular saída do PID
272         pidLeft.Compute();
273         pidRight.Compute();
274
275         // Aplicar correção de sincronização Ã s saídas do PID
276         // Modificado para incluir movimento puramente angular
277         if (abs(linearVelocity) > 0.01 || abs(angularVelocity) > 0.01) {
278         float outputLeftAdjusted, outputRightAdjusted;
279
280         if (abs(linearVelocity) > 0.01) {
281                 // Caso normal com correção de sincronização
282                 outputLeftAdjusted = OutputLeft * (1.0 - syncCorrection)
                        ;
283                 outputRightAdjusted = OutputRight * (1.0 +
                        syncCorrection);
284         } else {
285                 // Caso de rotação pura (sem correção de sincronização)
286                 outputLeftAdjusted = OutputLeft;
287                 outputRightAdjusted = OutputRight;
288         }
289
290         // Aplicar compensação agressiva ao motor direito
291         outputRightAdjusted = outputRightAdjusted *
                motorCompensationFactor;
292
293         // Garantir potÃªncia mínima para o motor esquerdo
294         if (absSetpointLeft > 0 && outputLeftAdjusted < 150) {
295                 outputLeftAdjusted = max(outputLeftAdjusted, 150.0);
296         }
297
298         // Limitar os valores ajustados
299         outputLeftAdjusted = constrain(outputLeftAdjusted, 0, 255);
300         outputRightAdjusted = constrain(outputRightAdjusted, 0, 255);
```

```
301
302         // Aplicar aos motores com direção
303         if (signLeft > 0) {
304                 motorLeft.setSpeed(outputLeftAdjusted);
305                 motorLeft.run(FORWARD);
306         } else if (signLeft < 0) {
307                 motorLeft.setSpeed(outputLeftAdjusted);
308                 motorLeft.run(BACKWARD);
309         } else {
310                 motorLeft.setSpeed(0);
311                 motorLeft.run(RELEASE);
312         }
313
314         if (signRight > 0) {
315                 motorRight.setSpeed(outputRightAdjusted);
316                 motorRight.run(FORWARD);
317         } else if (signRight < 0) {
318                 motorRight.setSpeed(outputRightAdjusted);
319                 motorRight.run(BACKWARD);
320         } else {
321                 motorRight.setSpeed(0);
322                 motorRight.run(RELEASE);
323         }
324
325         // Log de debug corrigido para evitar problemas com %f
326         if (currentTime % 1000 < 50) { // Log a cada ~1 segundo para não
              sobrecarregar
327                 char buffer[50];
328                 nh.loginfo("RPM: L=");
329                 dtostrf(rpmLeft, 4, 2, buffer);
330                 nh.loginfo(buffer);
331
332                 nh.loginfo(" R=");
333                 dtostrf(rpmRight, 4, 2, buffer);
334                 nh.loginfo(buffer);
335
336                 nh.loginfo(" Out: L=");
337                 dtostrf(outputLeftAdjusted, 4, 2, buffer);
338                 nh.loginfo(buffer);
339
340                 nh.loginfo(" R=");
341                 dtostrf(outputRightAdjusted, 4, 2, buffer);
342                 nh.loginfo(buffer);
343         }
344         } else {
345         // Parar motores quando a velocidade comandada é zero
346         motorLeft.setSpeed(0);
347         motorLeft.run(RELEASE);
```

```
348         motorRight.setSpeed(0);
349         motorRight.run(RELEASE);
350         }
351
352         // Publicar contagem de pulsos dos encoders para odometria
353         leftEncoderROS.data = totalPulsesLeft;
354         rightEncoderROS.data = totalPulsesRight;
355         leftEncoderPub.publish(&leftEncoderROS);
356         rightEncoderPub.publish(&rightEncoderROS);
357
358         // Atualizar tempo
359         previousTime = currentTime;
360 }
361
362 nh.spinOnce();
363 }
```

"

APÊNDICE D – Apêndice D

Codigo usado para comunicação da odometria entre firmware e software. Foi usado em conjunto com o codigo do apêndice C.

```
#!/usr/bin/env python3
import rospy
from std_msgs.msg import Int32
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Quaternion, TransformStamped
import tf2_ros
import math
import numpy as np

class WheelOdometry:
 def __init__(self):
  rospy.init_node('wheel_odometry', anonymous=True)

  # Parâmetros do robô
  # metros
  self.wheel_radius = rospy.get_param('~wheel_radius', 0.033)
  # metros
  self.wheel_separation = rospy.get_param('~wheel_separation', 0.20)
  # PPR do encoder
  self.encoder_ticks_per_revolution = rospy.get_param('~ticks_per_rev', 7)
  # Redução mecânica (1:380)
  self.gear_ratio = rospy.get_param('~gear_ratio', 380)
  self.base_frame_id = rospy.get_param('~base_frame_id', 'base_frame')
  self.odom_frame_id = rospy.get_param('~odom_frame_id', 'odom')

  # Reduzindo o tamanho do buffer para diminuir latência
  self.buffer_size = rospy.get_param('~buffer_size', 2)  # Reduzido de 5 para 2

  # Parâmetros para detecção de outliers - com limiares mais permissivos
  # Aumentado para ser menos restritivo
  self.max_tick_diff = rospy.get_param('~max_tick_diff', 500)
  # Aumentado para permitir velocidades mais altas
  self.max_velocity = rospy.get_param('~max_velocity', 2.0)
```

```
# Variáveis de estado
self.x = 0.0
self.y = 0.0
self.theta = 0.0
self.last_left_ticks = None
self.last_right_ticks = None
self.last_time = None
self.first_reading = True

# Filtro simples para encoders - buffer menor
self.left_ticks_buffer = []
self.right_ticks_buffer = []

# Covariância simples com valores fixos
self.pose_covariance = [0.01, 0, 0, 0, 0, 0,
      0, 0.01, 0, 0, 0, 0,
      0, 0, 0.01, 0, 0, 0,
      0, 0, 0, 0.01, 0, 0,
      0, 0, 0, 0, 0.01, 0,
      0, 0, 0, 0, 0, 0.01]

self.twist_covariance = [0.01, 0, 0, 0, 0, 0,
      0, 0.01, 0, 0, 0, 0,
      0, 0, 0.01, 0, 0, 0,
      0, 0, 0, 0.01, 0, 0,
      0, 0, 0, 0, 0.01, 0,
      0, 0, 0, 0, 0, 0.01]

# Usando TF2 para broadcasts
self.tf_broadcaster = tf2_ros.TransformBroadcaster()

# Publishers
# Reduzindo tamanho da fila
self.odom_pub = rospy.Publisher('odom', Odometry, queue_size=1)

# Subscribers com maior prioridade de callback
rospy.Subscriber('left_encoder_ticks', Int32,
self.left_encoder_callback, queue_size=1, tcp_nodelay=True)
rospy.Subscriber('right_encoder_ticks', Int32,
self.right_encoder_callback, queue_size=1, tcp_nodelay=True)
```

```
 self.left_ticks = None
 self.right_ticks = None

 # Timer para maior frequência
 # Aumentado para 50Hz
 self.update_rate = rospy.get_param('~update_rate', 50)
 self.timer = rospy.Timer(rospy.Duration(1.0/self.update_rate),
    self.timer_callback)


def apply_filter(self, buffer, new_value):
 # Filtro simplificado para reduzir latência
 buffer.append(new_value)
 if len(buffer) > self.buffer_size:
  buffer.pop(0)

 # Para buffer = 2, usar média simples
 if self.buffer_size <= 2:
  return int(sum(buffer) / len(buffer))

 # Para buffer maior, usar média ponderada priorizando valores mais recentes
 weights = np.linspace(0.5, 1.0, len(buffer))
 weighted_sum = sum(w * v for w, v in zip(weights, buffer))
 return int(weighted_sum / sum(weights))

def left_encoder_callback(self, msg):
 self.left_ticks = self.apply_filter(self.left_ticks_buffer, msg.data)
 # Processamento imediato
 self.process_encoders()

def right_encoder_callback(self, msg):
 self.right_ticks = self.apply_filter(self.right_ticks_buffer, msg.data)
 # Processamento imediato
 self.process_encoders()

def process_encoders(self):
 # Verifica se já recebeu dados dos dois encoders
 if self.left_ticks is None or self.right_ticks is None:
  return
```

```
 # Inicializa valores se for a primeira leitura
 if self.first_reading:
  self.last_left_ticks = self.left_ticks
  self.last_right_ticks = self.right_ticks
  self.last_time = rospy.Time.now()
  self.first_reading = False
  return

 # Pega o tempo atual antes de qualquer processamento
 current_time = rospy.Time.now()

def timer_callback(self, event):
 # Esta função atualiza a odometria com alta frequência
 self.update_odometry()

def update_odometry(self):
 # Verifica se já inicializou
 if self.first_reading or self.last_time is None:
  return

 # Verifica se tem dados dos encoders
 if self.left_ticks is None or self.right_ticks is None:
  return

 # Captura valores atuais para evitar inconsistências durante o processamento
 current_left = self.left_ticks
 current_right = self.right_ticks
 current_time = rospy.Time.now()

 # Só processa se houver mudança nos encoders
 if current_left == self.last_left_ticks and
 current_right == self.last_right_ticks:
  # Ainda assim, publica a transformação com o timestamp atual
  self.publish_tf_and_odom(current_time, 0.0, 0.0)
  return

 # Calcula as diferenças de ticks
 # (mantendo o sinal invertido conforme original)
 delta_left = current_left - self.last_left_ticks
```

```
delta_right = current_right - self.last_right_ticks

# Atualiza os últimos valores
self.last_left_ticks = current_left
self.last_right_ticks = current_right

# Calcula o tempo decorrido
dt = (current_time - self.last_time).to_sec()
self.last_time = current_time

# Verifica dt mínimo para evitar divisões por zero
if dt < 0.001:
 return

# Converte ticks para distância
num = (2 * math.pi * self.wheel_radius)
den = (self.encoder_ticks_per_revolution * self.gear_ratio)
meters_per_tick = num / den #eq.2.7
delta_left_dist = delta_left * meters_per_tick
delta_right_dist = delta_right * meters_per_tick

# Calcula o deslocamento do robô
delta_dist = (delta_left_dist + delta_right_dist) / 2.0
delta_theta = (delta_right_dist - delta_left_dist) / self.wheel_separation

# Atualiza a pose
if abs(delta_theta) < 0.0001:  # Quase em linha reta
 delta_x = delta_dist * math.cos(self.theta)
 delta_y = delta_dist * math.sin(self.theta)
else:
 # Movimento em arco
 radius = delta_dist / delta_theta
 delta_x = radius * (math.sin(self.theta + delta_theta) - math.sin(self.theta))
 delta_y = radius * (math.cos(self.theta) - math.cos(self.theta + delta_theta))

self.x += delta_x
self.y += delta_y
self.theta = (self.theta + delta_theta) % (2 * math.pi)

# Linear e angular velocities
```

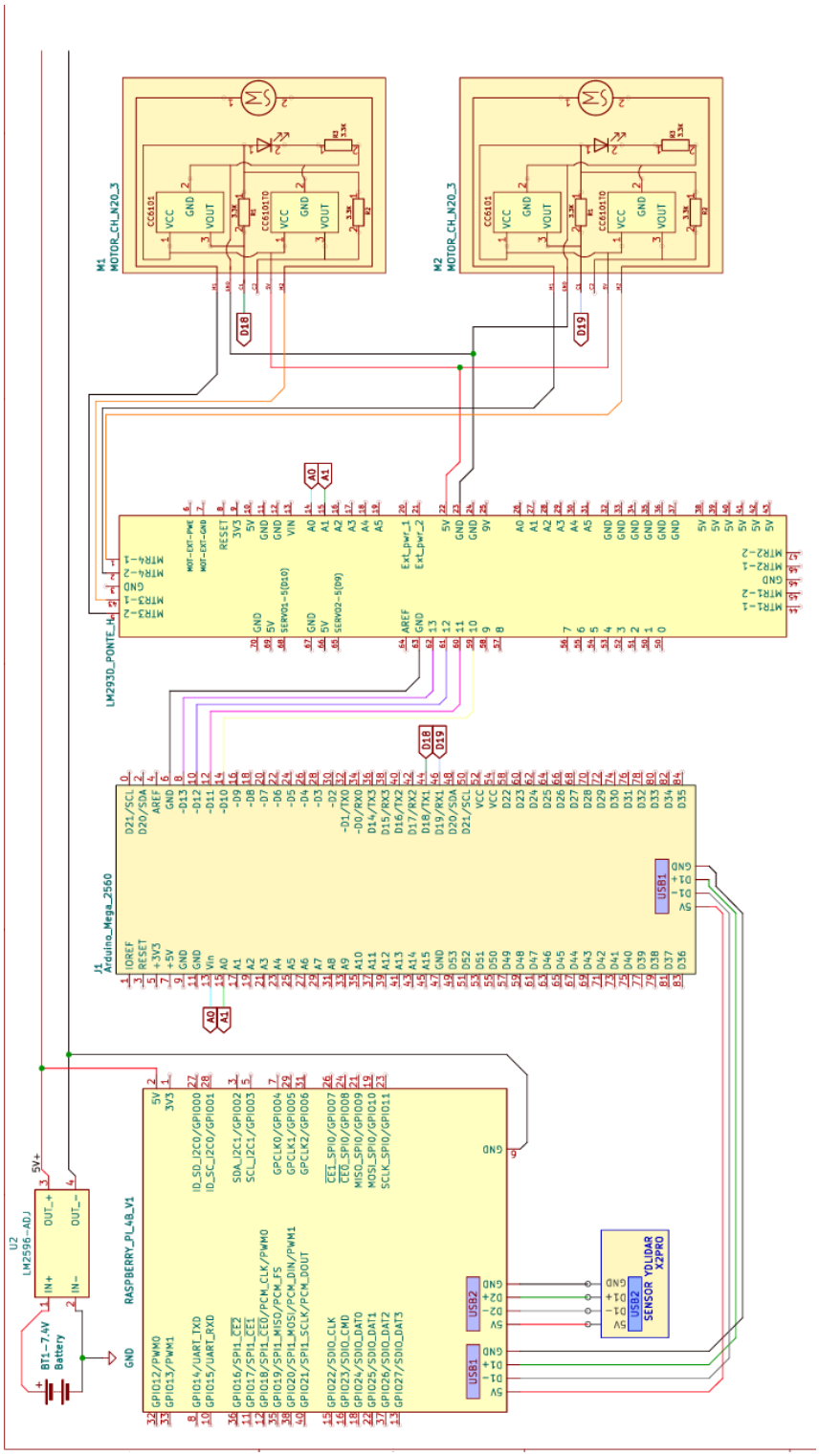
```
 linear_vel = delta_dist / dt
 angular_vel = delta_theta / dt

 # Publica TF e odometria
 self.publish_tf_and_odom(current_time, linear_vel, angular_vel)

def publish_tf_and_odom(self, timestamp, linear_vel, angular_vel):
 # Cria o quaternion
 quaternion = Quaternion()
 quaternion.x = 0.0
 quaternion.y = 0.0
 quaternion.z = math.sin(self.theta / 2.0)
 quaternion.w = math.cos(self.theta / 2.0)

 # Publica transformação TF2
 transform = TransformStamped()
 transform.header.stamp = timestamp
 transform.header.frame_id = self.odom_frame_id
 transform.child_frame_id = self.base_frame_id
 transform.transform.translation.x = self.x
 transform.transform.translation.y = self.y
 transform.transform.translation.z = 0.0
 transform.transform.rotation = quaternion

 self.tf_broadcaster.sendTransform(transform)

 # Publica odometria
 odom = Odometry()
 odom.header.stamp = timestamp
 odom.header.frame_id = self.odom_frame_id
 odom.child_frame_id = self.base_frame_id

 odom.pose.pose.position.x = self.x
 odom.pose.pose.position.y = self.y
 odom.pose.pose.position.z = 0.0
 odom.pose.pose.orientation = quaternion
 odom.pose.covariance = self.pose_covariance

 odom.twist.twist.linear.x = linear_vel
 odom.twist.twist.angular.z = angular_vel
```

```
  odom.twist.covariance = self.twist_covariance

  self.odom_pub.publish(odom)

if __name__ == '__main__':
 try:
  odometry = WheelOdometry()
  rospy.spin()
 except rospy.ROSInterruptException:
  pass
```

APÊNDICE E – Apêndice E

Esquema eletrico.

Figura 32 – Representação das ligações eletrônicas usadas no projeto



Fonte: Próprio Autor (2025)