UNIVERSIDADE FEDERAL DE UBERLÂNDIA

FACULDADE DE ENGENHARIA MECÂNICA - FEMEC

IGOR ALVES DE OLIVEIRA

Implementação de um Sistema de Comando de Pulsos para um Robô Delta

UBERLÂNDIA – MG

2026

IGOR ALVES DE OLIVEIRA

Implementação de um Sistema de Comando de Pulsos para um Robô Delta

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Uberlândia, como parte das exigências da graduação na Faculdade de Engenharia Mecânica, para obtenção do grau de Bacharel em Engenharia Mecatrônica

Orientador: Prof. Dr. Werley Rocherter Borges Ferreira

UBERLÂNDIA – MG

2026

IGOR ALVES DE OLIVEIRA

Implementação de um Sistema de Comando de Pulsos para um Robô Delta

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Uberlândia, como parte das exigências da graduação na Faculdade de Engenharia Mecânica, para obtenção do grau de Bacharel em Engenharia Mecatrônica

Banca Examinadora:

Prof. Msc. Werley Rocherter Borges Ferreira

Orientador

Prof. Dr. Roberto de Souza Martins

Membro da banca

Prof. Dr. José Jean-Paul Zanlucchi de Souza Tavares

Membro da banca

RESUMO

Este trabalho apresenta o desenvolvimento e a implementação de uma arquitetura de controle híbrida para um robô Delta de três graus de liberdade, destinada ao planejamento e execução de trajetórias suaves em aplicações do tipo *pick-and-place*. A proposta combina o uso do MATLAB para geração de trajetórias cartesianas, cálculo da cinemática inversa e discretização dos comandos articulares, com a execução embarcada em tempo real realizada por um microcontrolador ESP32-S3. O acionamento dos três eixos é feito por *servodrives* Yaskawa, sendo os comandos organizados em micro-blocos temporizados e transmitidos por UART sobre USB. Para garantir maior robustez na comunicação, foi implementado um protocolo binário com *bytes* de sincronização, CRC-8, fila circular e controle de fluxo por *software*. A geração dos pulsos no ESP32-S3 utiliza o periférico RMT, reduzindo a dependência de temporização por *software* e favorecendo a execução sincronizada entre os eixos. A validação numérica foi realizada por meio de uma trajetória circular no plano XY, com discretização de 4 ms e resolução efetiva de comando de 0,001 °/pulso, definida pela engrenagem eletrônica dos servodrives. Os resultados mostraram solução válida de cinemática inversa para todos os pontos analisados, frequência máxima de 3000 Hz, pulsos por micro-bloco abaixo dos limites estabelecidos e erro final nulo em pulsos. Nos ensaios experimentais, o robô executou trajetórias geométricas, ciclos repetitivos de *pick-and-place* e deslocamentos entre múltiplos pares de posições, demonstrando estabilidade funcional e compatibilidade entre o planejamento e a execução física. Embora tenham sido observadas distorções nas marcações, associadas a limitações mecânicas e geométricas do protótipo, os resultados confirmaram a viabilidade da arquitetura proposta para controle e execução de trajetórias em robôs Delta.

Palavras-chave: Robô Delta; controle embarcado; trajetória quíntica; *pick-and-place*; ESP32-S3.

ABSTRACT

This work presents the development and implementation of a hybrid control architecture for a three-degree-of-freedom Delta robot, aimed at planning and executing smooth trajectories for pick-and-place applications. The proposed system combines MATLAB for Cartesian trajectory generation, inverse kinematics calculation, and discretization of joint commands, with real-time embedded execution performed by an ESP32-S3 microcontroller. The three axes are driven by Yaskawa servodrives, with commands organized into timed micro-blocks and transmitted through UART over USB. To improve communication robustness, a binary protocol was implemented using synchronization bytes, CRC-8 verification, a circular buffer, and software flow control. Pulse generation on the ESP32-S3 is performed using the RMT peripheral, reducing dependence on software timing and improving synchronized execution among the axes. Numerical validation was carried out using a circular trajectory in the XY plane, with a 4 ms discretization period and an effective command resolution of 0.001 °/pulse, defined by the electronic gear configuration of the servodrives. The results showed valid inverse kinematics solutions for all analyzed points, a maximum frequency of 3000 Hz, pulse counts per micro-block below the established limits, and zero final pulse error. In the experimental tests, the robot executed geometric trajectories, repetitive pick-and-place cycles, and movements between multiple pairs of positions, demonstrating functional stability and compatibility between planning and physical execution. Although distortions were observed in the marked trajectories, associated with mechanical and geometric limitations of the prototype, the results confirmed the feasibility of the proposed architecture for trajectory control and execution in Delta robots.

Keywords: Delta robot; embedded control; quintic trajectory; pick-and-place; ESP32-S3.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

3T — *Three Translational Degrees of Freedom* (Três graus de liberdade translacionais).

ABB — *Asea Brown Boveri* (Empresa multinacional de automação e robótica).

CRC — *Cyclic Redundancy Check* (Verificação de redundância cíclica).

CRC-8 — *8-bit Cyclic Redundancy Check* (Verificação de redundância cíclica de 8 *bits*).

DIRBITS — *Direction Bits* (*Bits* de direção do frame de comando).

DTUS — *Delta Time in Microseconds* (Período do micro-bloco em microssegundos).

END — *End* (Indicação de fim ou último *frame*).

EPFL — *École Polytechnique Fédérale de Lausanne* (Escola Politécnica Federal de Lausanne).

ESP-IDF — *Espressif IoT Development Framework* (Ambiente de desenvolvimento IoT da Espressif).

ESP32-S3 — *Espressif ESP32-S3* (Microcontrolador utilizado no sistema embarcado).

GDL — Graus de liberdade (Número de movimentos independentes de um sistema mecânico).

GND — *Ground* (Terra ou referência elétrica comum do circuito).

GPIO — *General Purpose Input/Output* (Entrada/saída de uso geral).

HV — *High Voltage* (Lado de maior tensão do conversor de nível lógico).

IK — *Inverse Kinematics* (Cinemática inversa).

LV — *Low Voltage* (Lado de menor tensão do conversor de nível lógico).

MATLAB — *Matrix Laboratory* (Ambiente computacional utilizado para cálculo, simulação e geração dos comandos).

PC — *Personal Computer* (Computador pessoal).

PULS — *Pulse* (Sinal de pulso utilizado no comando dos *servodrives*).

PULS/SIGN — *Pulse/Sign* (Modo de comando por pulso e sinal de direção).

RMT — *Remote Control Transceiver* (Transceptor de controle remoto; periférico utilizado para geração temporizada de pulsos).

R.U.R. — *Rossum's Universal Robots* (Peça teatral em que o termo "robô" foi popularizado).

SEQ — *Sequence* (Campo de sequência do *frame* binário).

SIGN — *Sign* (Sinal de direção utilizado no comando dos servodrives).

SOF — *Start of Frame* (Início de *frame*).

UART — *Universal Asynchronous Receiver/Transmitter* (Transmissor/receptor assíncrono universal).

USB — *Universal Serial Bus* (Barramento serial universal).

VAC — *Volts Alternating Current* (Tensão em corrente alternada).

XON/XOFF — *Transmit On/Transmit Off* (Controle de fluxo por *software* para iniciar ou pausar a transmissão).

SUMÁRIO

1 INTRODUÇÃO

1.1 Robô Delta e manipulação rápida

O termo *robô* foi introduzido em 1921 pelo dramaturgo Karel Capek, em sua obra *R.U.R* (*Rossum's Universal Robots)*. A obra retrata como os robôs foram fabricados para substituir os trabalhadores humanos, com jornadas laborais sem descanso, que no final da história, se revoltam com seus criadores e exterminam toda a raça humana (FU; GONZALEZ; LEE, 1987).

Os robôs manipuladores industriais atuais começaram a ser desenvolvidos a partir do final da Segunda Guerra Mundial, com o objetivo de manipular materiais radioativos remotamente. Em 1959, Devol e Joseph F. Engelberger desenvolveram o primeiro robô industrial, introduzido pela empresa *Unimation Inc*. Nesse modelo já havia um computador acoplado ao manipulador, possibilitando que a máquina pudesse ser ensinada e adaptada para atender as necessidades do trabalho que deveria ser realizado (FU; GONZALEZ; LEE, 1987).

Segundo Romano (2002), o grande investimento na aplicação de robôs nos processos produtivos nas últimas décadas vem do fato da crescente necessidade de obter sistemas de produção mais automatizados e dinâmicos.

A automatização por mecanismos robóticos tem por objetivo reduzir os custos de produção, através da diminuição da mão de obra humana, aumento da produtividade, aproveitamento da matéria-prima e energia. Além disso, possibilita a melhoria das condições de trabalho do homem, melhoria da qualidade do produto e a realização de operações antes impossíveis para serem realizadas manualmente, como montagens e movimentos complexos e rápidos (BOUTEILLE *et al*., 1997).

Nesse contexto, Clavel (1991) observou, em uma indústria alimentícia, que a embalagem manual de produtos era uma atividade monótona, penosa e pouco higiênica. Como as soluções de mercado da época eram mal adaptadas para tarefas de alta velocidade e baixa carga, Clavel propôs uma estrutura inovadora de arquitetura paralela com quatro graus de liberdade, denominada robô Delta (CLAVEL, 1991). Nessa estrutura, os atuadores ficam fixos junto à base superior e as "pernas" ligam a base até a plataforma móvel. Essa configuração reduz drasticamente a massa das partes móveis, diminuindo a inércia da arquitetura e permitindo atingir as elevadas acelerações exigidas em operações de *pick-and-place*.

1.2 Problema de pesquisa

Embora o robô Delta possibilite elevadas acelerações e tempos de ciclo reduzidos, a sua estrutura leve torna o sistema sensível a vibrações e excitações dinâmicas durante movimentos rápidos. Em aplicações de *pick-and-place*, variações abruptas de aceleração podem provocar oscilações, aumentar esforços mecânicos e comprometer a repetibilidade do ponto de deposição, exigindo trajetórias com comportamento suave ao longo do tempo.

A suavidade do movimento está associada à continuidade de posição, velocidade e aceleração ao longo da trajetória, reduzindo excitações associadas à derivada temporal da aceleração (*jerk*). Nesse contexto, o uso de polinômios de quinto grau é adequado para movimentos ponto a ponto, pois permite impor condições de contorno e garantir continuidade de posição, velocidade e aceleração (BIAGIOTTI; MELCHIORRI, 2008).

Além do planejamento, a execução em altas frequências impõe requisitos de temporização determinística na geração dos sinais de comando. Em *servodrives* com interface PULS/SIGN (modo de comando por pulso e sinal de direção), pequenas variações no período dos pulsos (*jitter*) e a perda de sincronismo entre eixos podem degradar o desempenho. Assim, propõe-se uma arquitetura híbrida em que o MATLAB realiza o planejamento (trajetória quíntica, cinemática inversa e conversão para passos), enquanto o ESP32-S3 atua como executor em tempo real, recebendo pacotes via UART (*Universal Asynchronous Receiver/Transmitter*) sobre USB (*Universal Serial Bus*) e gerando os pulsos PULS/SIGN de forma estável e sincronizada.

1.3 Objetivos

1.3.1 Objetivo geral

Desenvolver e implementar uma arquitetura de comando para o robô Delta do Laboratório de Automação e Robótica, na qual o MATLAB realiza o planejamento da trajetória por polinômios de quinto grau, aplica a cinemática inversa e gera blocos de execução, enquanto o ESP32-S3 recebe os pacotes via UART sobre USB e executa a geração determinística de pulsos PULS/SIGN para acionamento dos *servodrives*,

visando movimento suave e desempenho compatível com operações de *pick-and-place*.

1.3.2 Objetivos Específicos

- Implementar a cinemática inversa do robô Delta, considerando as dimensões reais do protótipo, para conversão de coordenadas cartesianas (X, Y, Z) em comandos por junta.
- Planejar trajetórias ponto a ponto por polinômios de quinto grau no MATLAB, definindo condições de contorno e garantindo continuidade de posição, velocidade e aceleração.
- Discretizar a trajetória e converter os comandos por junta em passos, definindo blocos de execução (direção, número de passos e temporização) adequados para transmissão.
- Desenvolver o protocolo de comunicação MATLAB–ESP32-S3 via UART sobre USB e implementar no ESP32-S3 a recepção, validação e "*bufferização*" dos pacotes.
- Implementar no ESP32-S3 o executor multieixos PULS/SIGN com temporização determinística e mecanismos de segurança (limites, parada segura e tratamento de falhas).
- Validar o sistema por testes numéricos e práticos, analisando suavidade do movimento, repetibilidade e desempenho de execução (frequência máxima, *jitter* e sincronismo).

1.4 Justificativa

A relevância deste estudo se baseia na crescente necessidade por sistemas de automação que combinem alta produtividade com precisão e custos baixos. Atualmente, o robô Delta é considerado um padrão para tarefas de manipulação rápida, especialmente nos setores alimentício e farmacêutico, onde a cadência e a higiene são críticas (CLAVEL, 1991). No entanto, apenas adotar essa arquitetura não garante o ganho de eficiência do processo se não houver uma análise sobre controle da movimentação do robô.

Nesse sentido, é justificável a implementação das trajetórias baseadas em polinômios de quinto grau, buscando preservar a integridade mecânica do

manipulador e garantir a precisão dos seus movimentos. Ao assegurar a continuidade da aceleração, os esforços internos nos componentes são minimizados, reduzindo os custos de manutenção e aumentando a vida útil dos atuadores (CRAIG, 2005).

Ademais, a escolha do ESP32-S3 para controle do robô Delta é justificada, devido ao seu custo-benefício, mostrando desempenho satisfatório, se comparado a controladores industriais proprietários, podendo demonstrar uma alternativa para pequenas empresas e instituições de ensino. Dessa forma, este projeto contribui para o desenvolvimento de uma forma de controle acessível e estável para o ambiente laboratorial da faculdade.

1.5 Estrutura do Trabalho

Este relatório está organizado em cinco capítulos, além das referências bibliográficas e dos apêndices, de forma a apresentar, de maneira progressiva, a fundamentação teórica, o desenvolvimento do sistema proposto e a validação experimental.

O Capítulo 2 apresenta a fundamentação teórica necessária para o desenvolvimento do trabalho. Inicialmente são discutidos os conceitos relacionados aos robôs paralelos, com ênfase na arquitetura Delta, suas características construtivas, aplicações e espaço de trabalho. Em seguida, são apresentados os modelos matemáticos empregados, abrangendo a cinemática direta e inversa, os limites articulares considerados e os fundamentos utilizados para o planejamento de trajetórias.

O Capítulo 3 descreve o desenvolvimento da arquitetura proposta para o controle do robô Delta. São apresentados os parâmetros geométricos adotados, o planejamento das trajetórias no MATLAB, a discretização do movimento em comandos de pulsos, o protocolo de comunicação entre o computador e o ESP32, bem como a implementação do firmware responsável pela execução sincronizada dos movimentos nos servomotores.

O Capítulo 4 apresenta os resultados obtidos durante a validação do sistema. Inicialmente são discutidos os resultados das simulações numéricas referentes ao planejamento das trajetórias e à discretização dos comandos. Em seguida, são apresentados os ensaios experimentais realizados em bancada, incluindo a execução

de trajetórias geométricas, testes de pick-and-place e a análise do desempenho do executor embarcado por meio de registros experimentais.

Por fim, o Capítulo 5 reúne as conclusões do trabalho, destacando as principais contribuições obtidas, as limitações observadas durante o desenvolvimento e sugestões para trabalhos futuros.

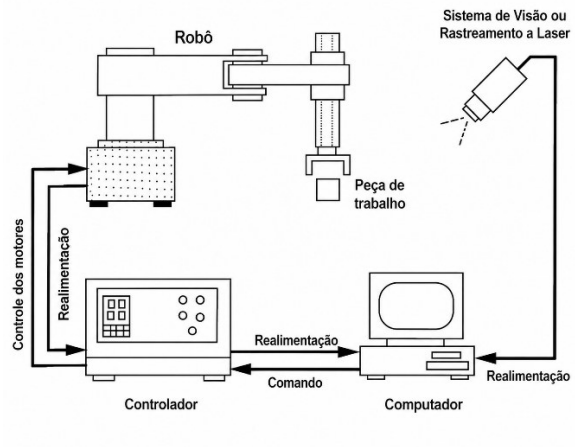
2 REVISÃO BIBLIOGRÁFICA

2.1 Sistemas Robóticos

Segundo Tsai (1999), um sistema robótico é composto por um manipulador mecânico, um elemento terminal, um controlador, um computador e, em muitos casos, um conjunto de sensores responsáveis pela aquisição de informações do ambiente e do próprio robô. Esses elementos atuam de forma integrada para executar tarefas de maneira automática, coordenando o planejamento, o controle e a execução dos movimentos.

A Figura 2.1 apresenta a arquitetura típica de um sistema robótico. Nessa configuração, o computador é responsável pelo planejamento das tarefas e pela geração dos comandos de movimento, podendo ainda processar informações provenientes de sistemas de visão ou de rastreamento a laser para identificar a posição da peça de trabalho. Esses comandos são enviados ao controlador, que executa as estratégias de controle e aciona os motores do robô por meio dos sinais de controle. Durante a execução, informações de realimentação, obtidas por sensores como encoders, são enviadas continuamente ao controlador para monitorar a posição e a velocidade dos atuadores, permitindo a correção de desvios e a sincronização dos movimentos. Em aplicações que utilizam sensores externos, como câmeras ou sistemas de visão, as informações do ambiente também são realimentadas ao computador, possibilitando a atualização dos comandos de acordo com as condições da tarefa.

Figura 2.1: Típico Sistema Robótico.



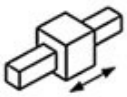
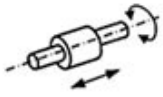
Fonte: Adaptado de Tsai (1999, p.17).

2.1.1 Elementos e Nomenclatura

O manipulador mecânico é um conjunto de *links* ou “elos”, conectados por “juntas” ou *joints*. Um desses elos é fixado em uma base, enquanto outro é considerado o terminal. Algumas das juntas do manipulador podem ser atuadas e outras passivas. As juntas atuadas estabelecem, geralmente, o número de graus de liberdade (GDL) do manipulador (TSAI, 1999). O grau de liberdade refere-se à translação ou a rotação de um corpo rígido no espaço. Um corpo rígido pode possuir até seis graus de liberdade, sendo três graus de translação em torno dos eixos ortogonais e três de rotação (MERLET, 2006).

Conforme Tsai (1999) a junta pode promover limitações físicas em relação ao movimento entre dois elos. O tipo de movimentação permitido entre dois elos é governado pela forma de contato entre as suas superfícies. Esta superfície de contato é denominada elemento de par robótico. Dois elementos de par robótico formam um par cinemático, os quais podem ser classificados como pares inferiores e superiores. Pares inferiores (Figura 2.2) são quando o contato ocorre através de uma superfície e os pares superiores quando estão em contato em um ponto ou ao longo de uma linha. As juntas mais comuns na robótica são:

Figura 2.2: Tipos de Juntas Cinemáticas Inferiores.

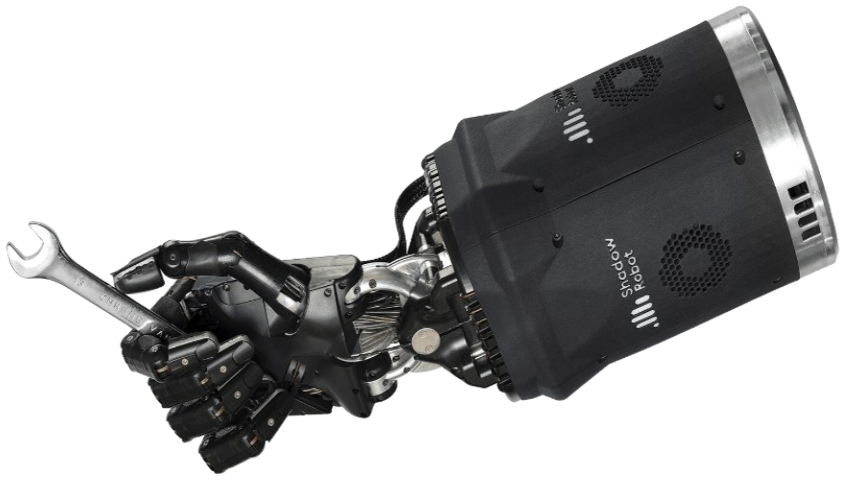
JUNTAS CINEMÁTICAS INFERIORES			
Designação	Geometria	Símbolo	Graus de liberdade
Rotação		R	1
Translação ou prismática		T	1
Esférica ou globular		E	3
Cilíndrica		C	2

Fonte: Adaptado de Aguiar ([s.d.]).

- Junta de Revolução (R): permite a rotação relativa entre dois elos em torno de um eixo comum, detendo um único grau de liberdade (1 GDL).
- Junta Prismática (P): possibilita o deslizamento linear (translação) entre dois elementos ao longo de um eixo definido, também limitando o movimento a 1 GDL.
- Junta Cilíndrica (C): permite tanto a rotação quanto a translação em relação a um eixo determinado, totalizando 2 GDL.
- Junta Universal (U): composta por dois eixos de revolução concorrentes e perpendiculares, permitindo 2 GDL (rotações em dois planos). É fundamental na estruturação de cadeias cinemáticas fechadas.
- Junta Esférica (S): funciona como uma rótula, permitindo a rotação livre em relação ao centro de uma esfera em todas as orientações, sem permitir a translação. Esta junta possui 3 GDL.

O elemento terminal ou efetuador consiste em um dispositivo que está conectado ao elo final da estrutura, sendo responsável pela interação do robô com o ambiente de trabalho. Ele pode ser uma garra, ventosa ou em ocasiões em que é necessário destreza e versatilidade, uma *dexterous hand* (Figura 2.3) (TSAI, 1999).

Figura 2.3: Exemplo de mão destra como elemento terminal.



Fonte: Shadow Robot Company (2026).

O tipo da junta e a estrutura do robô irão determinar as possíveis posições alcançáveis, que é chamado de espaço de trabalho. De maneira geral, o espaço de trabalho de um manipulador robótico é o volume total que o elemento terminal atinge

conforme o robô executa todos os movimentos possíveis. Ele se diferencia em dois tipos: o espaço de trabalho alcançável, que é o lugar geométrico total de pontos nos quais o efetuador pode ser posicionado; e o espaço de trabalho destro, sendo o subconjunto desses pontos nos quais o elemento terminal pode ser posicionado mantendo uma orientação arbitrária (WALDRON; SCHMIEDELER, 2008).

O controlador tem o propósito de controle via *feedback* (retroalimentação), recebendo os sinais lidos pelos sensores incorporados nos atuadores do robô, como *encoders*. Em sistemas mais avançados, pode ser necessário a presença de sensores de força e toque ao longo do manipulador, para um comando mais inteligente. Ademais, o controlador também pode estar equipado com um *teach pendant*, sendo capaz de programar posições, editar e executar programas. O computador pode ser um microprocessador, um computador pessoal ou um servidor capaz realizar operações complexas de raciocínio e emissão de controles inteligentes em tempo real (TSAI, 1999).

2.1.2 Classificação de mecanismos robóticos

Segundo Molina (2008), os manipuladores mecânicos podem ser classificados a partir dos graus de liberdade, estrutura cinemática, espaço de trabalho e movimentação.

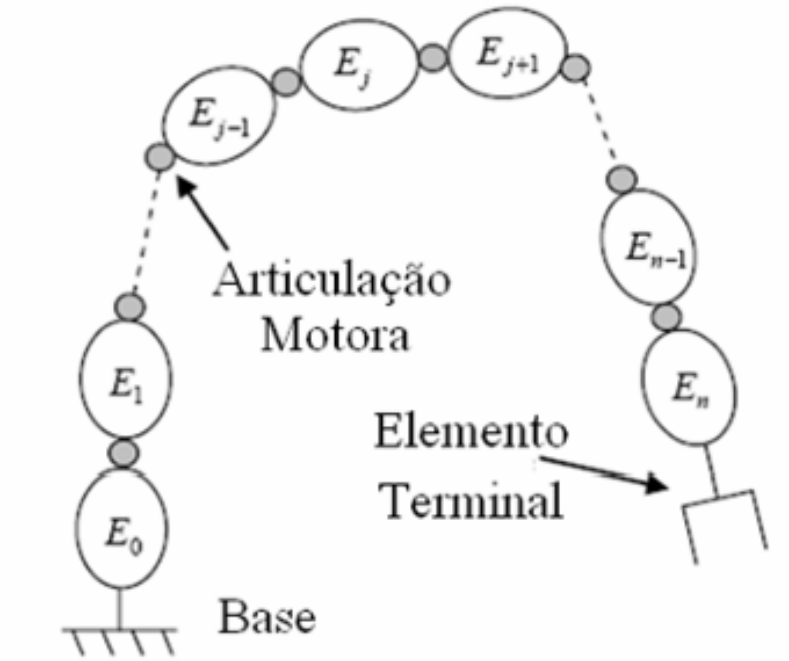
Caracterizando a partir da arquitetura cinemática, pode-se distinguir duas formas: seriais ou paralelos. Manipuladores seriais são formados por cadeias cinemáticas abertas, onde cada elo que forma a cadeia é conectado por apenas uma junta ao próximo elo, constituindo um único segmento da base ao efetuador. A maioria dos robôs seriais apresenta um caráter antropomórfico, replicando a estrutura de um braço humano (MERLET, 2006).

Mecanismos paralelos utilizam cadeias cinemáticas fechadas, sendo constituídos por um conjunto de cadeias abertas que se unem em um único ponto no elemento terminal (TSAI, 1999). Para um manipulador paralelo, geralmente sua mobilidade é determinada pela soma dos graus de liberdade de cada junta atuada. Em alguns casos, essa soma pode exceder o número máximo de graus de liberdade cinematicamente possíveis, tornando a estrutura atuada de forma redundante (PARK, 2015).

As Figuras 2.4 e 2.5 apresentam, respectivamente, as arquiteturas serial e paralela. Na Figura 2.4 observa-se um manipulador serial composto por uma única cadeia cinemática aberta, na qual cada elo é conectado sequencialmente ao elo seguinte por meio de uma articulação motora, formando um único caminho entre a base e o elemento terminal. Nessa configuração, cada atuador é responsável por movimentar todos os elos subsequentes, característica que proporciona grande espaço de trabalho e elevada flexibilidade, porém reduz a rigidez estrutural e favorece a propagação de erros de posicionamento ao longo da cadeia cinemática.

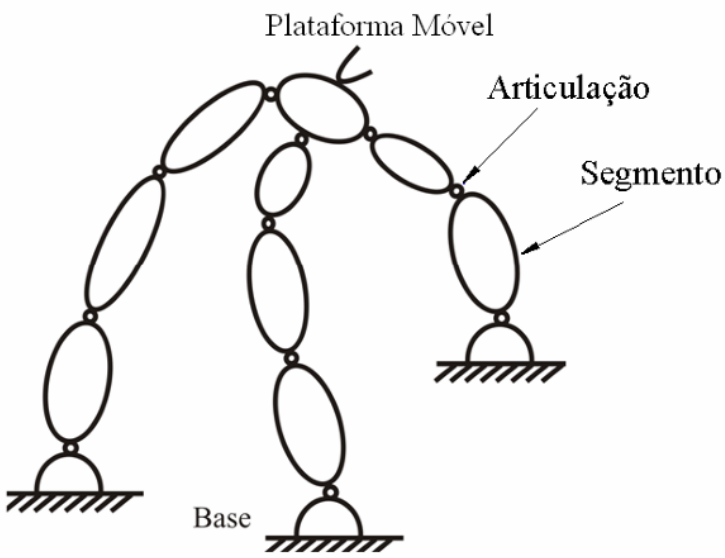
A Figura 2.5 ilustra um manipulador paralelo, cuja plataforma móvel é sustentada simultaneamente por múltiplas cadeias cinemáticas conectadas à base. Nessa arquitetura, os esforços mecânicos são distribuídos entre os diferentes segmentos da estrutura, proporcionando maior rigidez, elevada capacidade de carga e melhor precisão de posicionamento quando comparada aos manipuladores seriais. Em contrapartida, os manipuladores paralelos normalmente apresentam um espaço de trabalho mais limitado e uma modelagem cinemática mais complexa.

Figura 2.4: Arquitetura Serial Simples.



Fonte: Gonçalves (2009).

Figura 2.5: Arquitetura de um Manipulador Paralelo.



Fonte: Gonçalves (2009).

2.1.3 Comparativo entre Manipuladores Seriais e Paralelos

Para Merlet (2006), os robôs seriais operam sob a ação de dois tipos de força: a inércia e a de fricção. Visto que cada um dos elos deve suportar os elos e atuadores seguintes, a estrutura está submetida a elevados torques. Para evitar excessivas deformações durante os movimentos, os manipuladores tendem a possuir uma estrutura robusta e pesada. Por consequência, essa massa adicional em movimentos de alta velocidade faz com que o sistema sofra com a inércia dos componentes, além das forças centrífugas e de *Coriolis*. A atuação dessas forças externas ocasiona pequenas mudanças no posicionamento do elemento terminal, devido a folgas na transmissão e na flexão dos elos. Como essas alterações geralmente não podem ser medidas pelos sensores internos nos atuadores, elas não são corrigidas pelo sistema, tornando o controle do robô serial complexo.

Por outro lado, os robôs paralelos possuem vantagens significativas, como maior rigidez estrutural, maior capacidade de carga útil, e menor inércia, uma vez que os atuadores são montados na base da estrutura do robô paralelo. Gonçalves (2009) afirma que na configuração de cadeia fechada, os esforços são distribuídos entre as pernas do manipulador, resultando em deformações significativamente menores do que aquelas observadas em manipuladores seriais de mesma capacidade de carga.

Essa configuração permite que os componentes móveis sejam mais leves, possibilitando a operação em altas velocidades e acelerações. Essas características são aproveitadas em diversos ambientes, como o radioativo, locais úmidos ou com elevadas temperaturas (ARAI; SHERIDAN, 1988). Contudo, esses benefícios ocorrem ao custo de um espaço de trabalho reduzido e com um mecanismo mais complexo, principalmente tratando-se dos cálculos da cinemática direta (TSAI, 1999).

Além disso, Merlet (2006) diz que a precisão de posicionamento de manipuladores paralelos é superior por duas razões fundamentais:

- As deformações (não medidas) dos elos devido à flexão são reduzidas, se comparados com outros tipos de estruturas.
- Os erros nos sensores internos do robô (medição dos comprimentos dos elos) afetam ligeiramente os erros na posição da plataforma. Por exemplo, se todos os sensores apresentarem o mesmo erro, o cálculo da posição da plataforma com base nas medições dos sensores mostrará um erro apenas para o eixo vertical: a amplitude do erro será aproximadamente a mesma que o erro nos sensores.

A Tabela 2.1 apresenta uma análise qualitativa baseada em Arai e Sheridan (1988), onde as duas arquiteturas são comparadas em relação a determinadas características funcionais.

Tabela 2.1: Análise Qualitativa das Estruturas.

Características	**Serial**	**Paralela**
Espaço de Trabalho	Grande	Pequeno
Controle de Posição	Difícil	Fácil
Posição do Elemento Terminal	Fácil	Difícil
Controle de Força	Fácil	Difícil
Erro de Posição	Acumulado	Médio
Medição de Forças	Difícil	Fácil
Dinâmica	Complexa	Extremamente complexa

Fonte: Adaptado de Arai e Sheridan (1988).

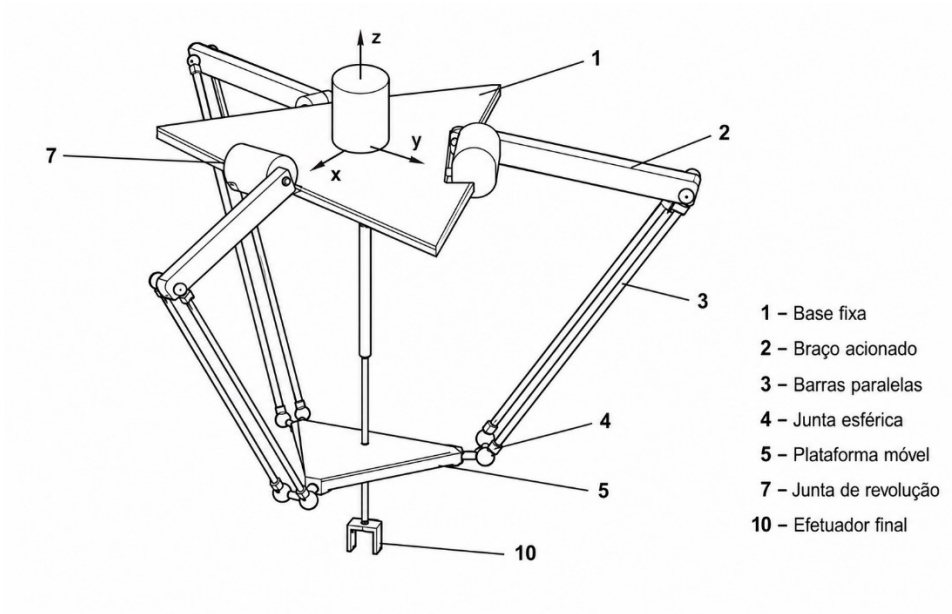
Arai e Sheridan (1988) concluem que, os manipuladores seriais são mais indicados em tarefas que requeiram movimentos complexos com controle exatos das forças aplicadas e na montagem de objetos leves em curto tempo. Em contrapartida, robôs paralelos são melhor aplicados em tarefas que exigem manipulação rápida de objetos pesados, em movimentos mais simples.

2.2 Robô Delta

2.2.1 Estrutura e Características

O robô Delta foi proposto por Reymond Clavel e seu grupo de pesquisa na *École Polytechnique Fédérale de Lausanne* (EPFL), na década de 1980, com motivação associada à automação de tarefas repetitivas e de alta cadência em linhas de produção, especialmente em aplicações de manuseio rápido. O conceito destaca-se pela arquitetura paralela e pela baixa massa em movimento, permitindo elevadas acelerações e tempos de ciclo reduzidos. Na sua configuração clássica, o robô Delta possui três graus de liberdade translacionais (3T), mantendo a orientação da plataforma móvel aproximadamente constante ao longo do movimento, conforme ilustrado na Figura 2.6 (CLAVEL, 1991).

Figura 2.6: Estrutura cinemática do robô Delta de Clavel.



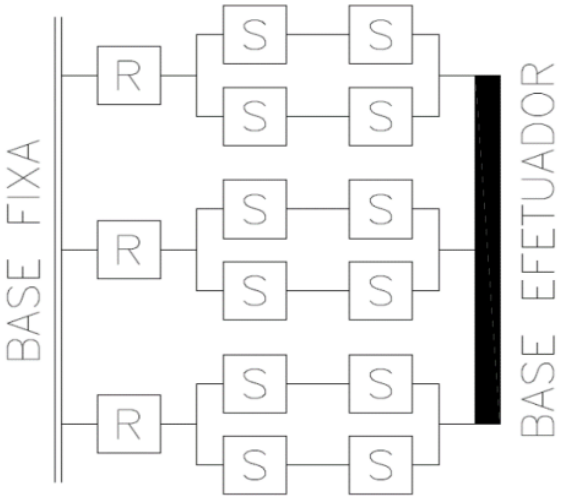
Fonte: Adaptado de Clavel (1991).

A estrutura do robô Delta é composta por uma base fixa (1) e uma plataforma móvel (5), na qual é acoplado o efetuador final (10). A ligação entre a base fixa e a plataforma móvel é realizada por três cadeias cinemáticas distribuídas simetricamente. Em cada cadeia, um braço acionado (2) é conectado à base fixa por uma junta de revolução (7), responsável pelo movimento gerado pelo atuador. A extremidade do braço acionado é ligada a um conjunto de barras paralelas (3), cujas extremidades são conectadas à plataforma móvel por juntas esféricas (4). Essas barras formam um paralelogramo articulado, que impõe restrições geométricas responsáveis por manter a orientação da plataforma móvel praticamente constante durante o deslocamento (CLAVEL, 1991).

As juntas esféricas permitem a rotação relativa necessária entre os elos para acomodar o movimento espacial, enquanto o paralelogramo garante a condição de paralelismo entre a base fixa e a plataforma móvel, resultando em um manipulador cuja plataforma realiza predominantemente movimentos de translação. Essa característica torna a arquitetura Delta particularmente adequada para operações de *pick-and-place*, em que a rapidez e a repetibilidade são requisitos importantes (CLAVEL, 1991).

A representação em forma de grafo permite visualizar de maneira simplificada as juntas e cadeias cinemáticas que formam a estrutura do robô Delta. A Figura 2.7 mostra o grafo para a estrutura Delta.

Figura 2.7: Organização das Juntas do Robô Delta.



Fonte: Santos Junior e Ramirez (2011).

Conforme ilustrado na Figura 2.7, o robô Delta é constituído por três cadeias cinemáticas idênticas que conectam a base fixa à base do efetuador, formando uma estrutura paralela. Em cada cadeia, a primeira junta é do tipo revolução (R), correspondente à articulação acionada pelo atuador. Na sequência, cada braço conecta-se a um paralelogramo articulado composto por quatro juntas esféricas (S), responsáveis por permitir a mobilidade necessária sem transmitir rotações à plataforma móvel.

2.2.2 Aplicações típicas do Delta

O robô Delta (Figura 2.8) consolidou-se como uma das arquiteturas paralelas mais bem-sucedidas em aplicações industriais de manuseio rápido (*high-speed handling*) e *pick-and-place*, principalmente devido à combinação de baixa massa móvel, alta aceleração e boa repetibilidade. Essas características tornam o manipulador especialmente adequado para a manipulação de objetos leves e, muitas vezes, frágeis, com elevada cadência, como ocorre em linhas de embalagem e classificação na indústria alimentícia (Figura 2.9) e em operações de manuseio de componentes eletrônicos (MERLET, 2006).

Além do *pick-and-place*, robôs paralelos podem ser aplicados em tarefas que exigem maior rigidez estrutural, como o posicionamento de ferramentas e operações envolvendo esforço de processo, a exemplo de aplicações em usinagem e soldagem, desde que a geometria do mecanismo e o volume de trabalho atendam aos requisitos da tarefa (MOLINA, 2008).

Figura 2.8: Robô Delta ABB IRB 360 FlexPicker.

Fonte: ABB (2026).

Figura 2.9: Robôs Delta trabalhando na indústria alimentícia.

Fonte: Bonev (2001).

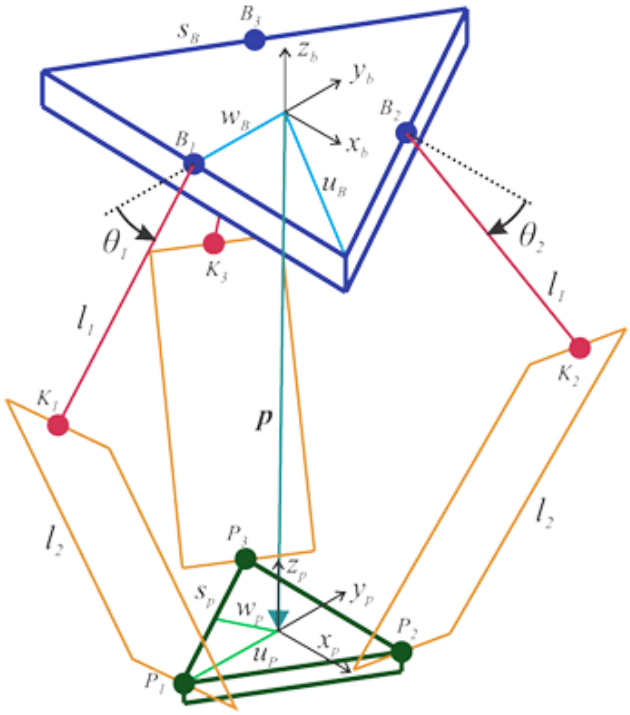
2.3 Cinemática do Robô Delta

Esta seção apresenta o modelo cinemático adotado para o robô Delta 3T, com foco na formulação implementada para cinemática inversa e nos critérios de validade utilizados para garantir que os pontos planejados sejam executáveis dentro do volume operacional. As equações finais e o procedimento de cálculo utilizados nesta implementação são apresentados nesta seção. A dedução detalhada segue o método descrito por Williams II (2016), com adaptações de notação e parâmetros do protótipo.

2.3.1 Convenções e parâmetros geométricos

A geometria do robô é descrita pelos parâmetros s_b, s_p, L e l, além do sistema de coordenadas com origem no centro da base fixa $\{B\}$. A posição da plataforma móvel é representada por $P(x, y, z)$, e as variáveis articulares são θ_1, θ_2, θ_3, medidas em relação ao plano da base superior. A Figura 2.10 mostra uma estrutura simplificada do robô Delta com os principais parâmetros geométricos analisados.

Figura 2.10: Diagrama Cinemático do Robô Delta.



Fonte: Titton e Valente (2018).

Cada um dos parâmetros geométricos é descrito abaixo:

- l_1: Comprimento do elo entre os atuadores (B_i) às juntas passivas (K_i);
- l_2: Comprimento do elo entre as juntas passivas (K_i) e a junta da base móvel (P_i);
- s_B: Comprimento da aresta do triângulo formado no plano da base fixa;
- u_B: Distância da origem do sistema de coordenadas da base fixa ao seu vértice, com valor dado por $u_B = \frac{s_B\sqrt{3}}{3}$;
- w_B: Distância da origem do sistema de coordenadas da base fixa ao ponto do atuador (B_i), dado por $w_B = \frac{s_B\sqrt{3}}{6}$;
- s_P: Comprimento da aresta do triângulo formado no plano da base móvel;
- u_P: Distância da origem do sistema de coordenadas da base móvel ao seu vértice, dado por $u_P = \frac{s_P\sqrt{3}}{3}$;
- w_P: Distância da origem do sistema de coordenadas da base móvel ao ponto central de uma aresta, dado por $w_P = \frac{s_P\sqrt{3}}{6}$;

A partir desses parâmetros, Williams II (2016) define também algumas constantes utilizadas para o cálculo da cinemática inversa, conforme (2.1).

$$a = w_B - u_P, \qquad b = \frac{s_P}{2} - \frac{\sqrt{3}}{2} w_B, \qquad c = w_P - \frac{1}{2} w_B \tag{2.1}$$

2.3.2 Cinemática Inversa

A cinemática inversa consiste em determinar os valores das variáveis articulares θ_1, θ_2 e θ_3 a partir da posição correspondente do elemento terminal $P(x, y, z)$ (CLAVEL, 1991). Segundo Angeles (2014), os métodos para obter as equações que descrevem a posição das juntas são geralmente algébricos e mais simples de serem feitos, assim como a parte computacional.

Conforme Williams II (2016), a solução é obtida analisando-se a geometria do mecanismo no plano de atuação de cada uma das três cadeias cinemáticas que forma a estrutura, resultando em uma expressão fechada para cada θ_i, onde $i = 1, 2, 3$, correspondendo a cada uma das juntas. Neste trabalho, a cinemática inversa é aplicada ponto a ponto ao longo da trajetória discretizada, pois é computacionalmente eficiente e permite verificar inviabilidade de pontos (fora do espaço de trabalho) antes da execução.

A partir da análise geométrica da estrutura, Williams II (2016) expressa o sistema de três equações que descrevem a cinemática do robô Delta:

$$2l_1(y + a)\cos\theta_1 + 2zl_1 \sin\theta_1 + x^2 + y^2 + z^2 + a^2 + {l_1}^2 + 2ya - l_2^2 = 0$$

$$-l_1\left(\sqrt{3}(x + b) + y + c\right)\cos\theta_2 + 2zl_1 \sin\theta_2 + x^2 + y^2 + z^2 + b^2 + c^2 + l_1^2 + 2xb + 2yc - l_2^2 = 0$$

$$l_1\left(\sqrt{3}(x - b) - y - c\right)\cos\theta_3 + 2zl_1 \sin\theta_3 + x^2 + y^2 + z^2 + b^2 + c^2 + l_1^2 - 2xb + 2yc - l_2^2 = 0$$

O sistema de três equações acima pode ser reduzido a Equação (2.2):

$$E_i \cos\theta_i + F_i \operatorname{sen}\theta_i + G_i = 0, i = 1, 2, 3 \tag{2.2}$$

A Equação (2.2) conforme os estudos de Williams II (2016), pode ser solucionada a partir da substituição da tangente de meio-ângulo, onde:

$$t_i = \tan\frac{\theta_i}{2}, \qquad \cos\theta_i = \frac{1 - t_i^2}{1 + t_i^2}, \qquad \operatorname{sen}\theta_i = \frac{2t_i}{1 + t_i^2}$$

Obtendo assim a equação quadrática expressa em (2.3).

$$t_{i_{1,2}} = \frac{-F_i \pm \sqrt{E_i^2 + F_i^2 - G_i^2}}{G_i - E_i} \tag{2.3}$$

$$\theta_i = 2\,\mathrm{atan}(t_i) \tag{2.4}$$

Onde, para cada junta, tem-se:

- Junta 1

$$\begin{aligned} E_1 &= 2l_1(y + a) \\ F_1 &= 2zl_1 \\ G_1 &= x^2 + y^2 + z^2 + a^2 + l_1^2 + 2ya - l_2^2 \end{aligned} \tag{2.5}$$

- Junta 2

$$\begin{aligned} E_2 &= -l_1\left(\sqrt{3(x + b)} + y + c\right) \\ F_2 &= 2zl_1 \\ G_2 &= x^2 + y^2 + z^2 + b^2 + c^2 + l_1^2 + 2(xb + yc) - l_2^2 \end{aligned} \tag{2.6}$$

- Junta 3

$$\begin{aligned} E_3 &= l_1\left(\sqrt{3(x - b)} - y - c\right) \\ F_3 &= 2zl_1 \\ G_3 &= x^2 + y^2 + z^2 + b^2 + c^2 + l_1^2 + 2(-xb + yc) - l_2^2 \end{aligned} \tag{2.7}$$

As Equações 2.3 e 2.4 são solúveis se:

$$\begin{aligned} E_i^2 + F_i^2 - G_i^2 &> 0 \\ G_i - E_i &\neq 0 \end{aligned} \tag{2.8}$$

Caso os valores atribuídos em $P(x, y, z)$ não satisfaçam as condições expressas em (2.8), os pontos estarão fora do volume de trabalho alcançado.

De acordo com Williams II (2016), existem um total de 8 possíveis soluções válidas. Normalmente, a solução escolhida é aquela em que todos os “joelhos” ficam voltados para fora ao invés das que estão voltadas para dentro.

2.3.3 Cinemática Direta

Na cinemática direta, será determinada a posição de um ponto fixo, nesse caso $\{P\}$, em relação aos ângulos θ de cada uma das juntas de revolução do modelo, sendo mais complexa de se obter que a cinemática inversa (ANGELES, 2014).

Williams II (2016) utiliza o método da intersecção das três esferas de raio l, para solucionar a cinemática direta do robô Delta, onde a intersecção é $P(x, y, z)$. Nesse método, ele explica que há duas situações possíveis: uma delas quando a

movimentação do robô é feita em x, y, z; e outra quando existe uma translação apenas no eixo z.

Para a primeira situação, tem-se a seguinte solução:

$$y_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.9}$$

$$x\pm = a_4 y_{\pm} + a_5 \tag{2.10}$$

$$z\pm = a_6 y_{\pm} + a_7 \tag{2.11}$$

Onde:

$$a = a_4^2 + 1 + a_6^2$$

$$b = 2a_4(a_5 - x_1) - 2y_1 + 2a_6(a_7 - z_1)$$

$$c = a_5(a_5 - 2x_1) + a_7(a_7 - 2z_1) + x_1^2 + y_1^2 + z_1^2 - r_1^2$$

$$a_{11} = 2(x_3 - x_1) \quad a_{21} = 2(x_3 - x_2) \quad b_1 = r_1^2 - r_3^2 - x_1^2 - y_1^2 - z_1^2 + x_3^2 + y_3^2 + z_3^2$$

$$a_{12} = 2(y_3 - y_1) \quad a_{22} = 2(y_3 - y_2) \quad b_2 = r_2^2 - r_3^2 - x_2^2 - y_2^2 - z_2^2 + x_3^2 + y_3^2 + z_3^2$$

$$a_{13} = 2(z_3 - z_1) \quad a_{23} = 2(z_3 - z_2)$$

$$a_4 = -\frac{a_2}{a_1}, \qquad a_5 = -\frac{a_3}{a_1}, \qquad a_1 = \frac{a_{11}}{a_{13}} - \frac{a_{21}}{a_{23}}, \qquad a_2 = \frac{a_{12}}{a_{13}} - \frac{a_{22}}{a_{23}}, \qquad a_3 = \frac{b_2}{a_{23}} - \frac{b_1}{a_{13}}$$

Nos casos em que a movimentação ocorre apenas ao longo do eixo z, define-se:

$$z_{p,m} = \frac{-B \pm \sqrt{B^2 - 4C}}{2} \tag{2.12}$$

Onde:

$$A = 1$$

$$B = -2z_n$$

$$C = z_n^2 - r_1^2 + (x - x_1)^2 + (y - y_1)^2$$

Ambas as soluções resultaram em dois valores para z. Conforme a geometria do robô Delta, adota-se o resultado onde o valor de z é negativo, pois é medido em relação a base superior fixa.

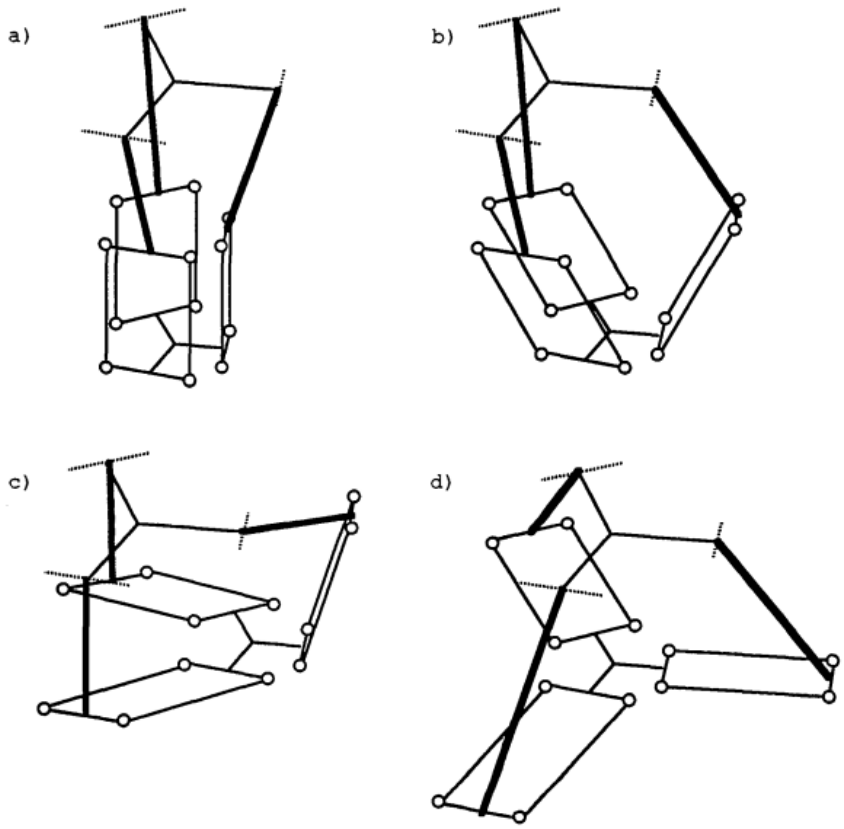
2.3.4 Espaço de Trabalho e Singularidades

Segundo Clavel (1991), em um manipulador serial, as singularidades decorrem da perda de graus de liberdade da estrutura enquanto para um manipulador paralelo, as singularidades estão associadas ao ganho de mobilidades indesejadas. Em

determinadas configurações, a plataforma móvel pode executar movimentos adicionais que não são adequadamente restringidos pelas barras, causando perda de rigidez e de controlabilidade.

Para o robô Delta, a posição da plataforma móvel é totalmente definida e sua estabilidade assegurada se as seguintes condições forem respeitadas: as barras paralelas dos elos inferiores estão situadas em três planos diferentes e não paralelos; no máximo duas barras são paralelas (CLAVEL, 1991). Essas condições evidenciam quatro tipos de singularidades, apresentadas na Figura 2.11.

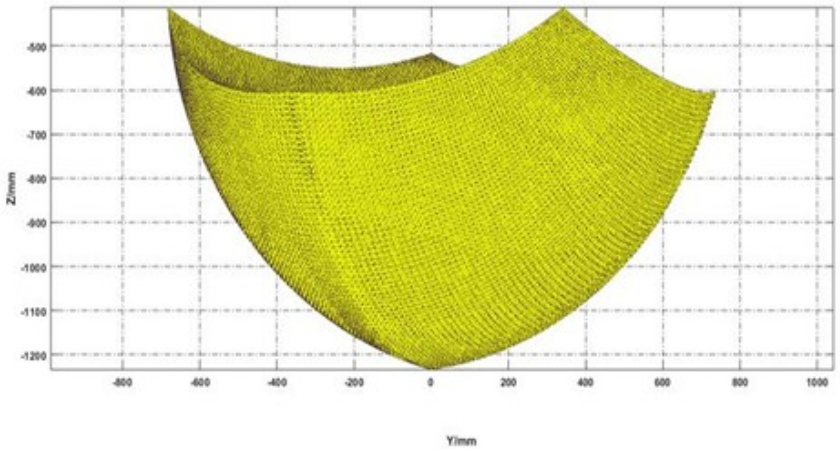
Figura 2.11: Representação dos 4 tipos de Singularidades do Robô Delta.



Fonte: Clavel (1991).

Além disso, Clavel (1991) define o espaço de trabalho do robô Delta como a interseção entre um prisma hexagonal reto e três sólidos de revolução, que resulta no modelo apresentado na Figura 2.12. Essa geometria fornece uma estimativa do volume acessível ao centro da plataforma móvel, considerando as limitações impostas pela estrutura paralela e as restrições dos movimentos dos elos.

Figura 2.12: Representação do Espaço de Trabalho do Robô Delta



Fonte: Zheng e Chen (2023).

Embora o espaço de trabalho do robô Delta dependa de restrições geométricas e possa incluir regiões próximas a configurações singulares, o foco deste trabalho não é dimensionar ou mapear integralmente o volume de trabalho válido. Em vez disso, adotam-se critérios práticos de validade voltados à execução segura e repetível das trajetórias planejadas, baseados em limites articulares e em verificações numéricas da cinemática inversa.

2.4 Planejamento de Trajetória

O planejamento de trajetória geralmente interpola ou aproxima o caminho desejado por meio de uma classe de funções polinomiais e gera uma sequência de pontos no tempo para controlar o manipulador até a posição desejada. Essa trajetória pode ser descrita em coordenadas das juntas ou no plano cartesiano (FU; GONZALEZ; LEE, 1987).

Calcular uma trajetória no espaço das juntas possui três vantagens: (1) a trajetória é calculada diretamente nas variáveis controladas durante o movimento; (2) é possível de ser feita em tempo real; e (3) as trajetórias das juntas são mais fáceis de serem calculadas. A desvantagem é a dificuldade em determinar a posição de vários elos e do elemento terminal durante a movimentação, sendo necessário garantir a não existência de obstáculos no caminho (FU; GONZALEZ; LEE, 1987).

Já no espaço cartesiano, o movimento é mais preciso durante o caminho, porém os algoritmos de controle são mais complexos, necessitando da conversão de cada ponto gerado na trajetória em coordenadas de junta. Isso é computacionalmente intensivo, exigindo longos intervalos de controle, prejudicando a operação em tempo real (FU; GONZALEZ; LEE, 1987).

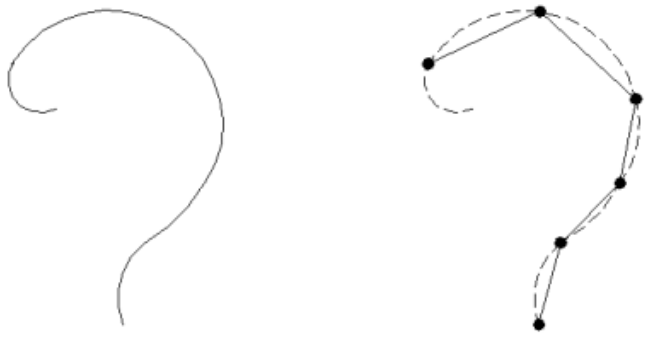
No contexto do robô Delta do presente projeto, adotou-se o planejamento no espaço cartesiano, convertendo os pontos da trajetória, por meio da cinemática inversa, em comandos articulares θ_1, θ_2 e θ_3 e posteriormente realizando a discretização para execução por trem de pulsos.

2.4.1 Movimento ponto a ponto e trajetória contínua

Segundo Angeles (2014), as movimentações feitas por um sistema robótico devem ser as mais suaves possíveis, descartando qualquer possibilidade de mudanças abruptas de direção, velocidade e aceleração. Craig (2005) também afirma que essas alterações drásticas de aceleração podem desgastar os componentes mecânicos e causar vibrações, excitando ressonâncias no manipulador. Existem dois tipos comuns de planejamento de trajetória, nomeados como:

- Operações de Pick-and-Place e;
- Trajetórias Contínuas.

Figura 2.13: Trajetória Contínua e Ponto a Ponto



Fonte: Pazos (2002).

Nas operações de *pick-and-place*, o manipulador robótico deve pegar um objeto em uma posição inicial conhecida, especificada pelos pontos e orientações do sistema de coordenadas adotado, e transportá-lo até uma posição final desejada.

Esse tipo de movimento está presente na maioria dos processos de manufatura, como carga e descarga de esteiras, trocas de ferramentas em máquinas de usinagem e até mesmo em simples operações de montagem. Nesse tipo de movimento, a trajetória intermediária não é o aspecto principal, desde que o deslocamento seja suave e não ocorram colisões durante o percurso (ANGELES, 2014).

Trajetórias contínuas estão presentes em operações de soldagem, corte, acabamento e fresamento. Nessas operações, uma ferramenta é acoplada rigidamente ao elemento terminal do robô, o qual deve se mover de forma contínua e suave no espaço de trabalho, seguindo uma trajetória especificada. Por exemplo, para fabricar a fuselagem de uma aeronave, é necessário que o caminho percorrido seja semelhante a uma curva não plana, descrita em função do tempo (ANGELES, 2014). Na soldagem, a tocha da solda deve se deslocar ao longo de um caminho pré-determinado para efetuar o procedimento corretamente (PAZOS, 2002).

2.4.2 Polinômios de quinto grau e suavidade de movimento

Clavel (1991), em seu estudo sobre a construção do robô Delta, conclui que o tipo de movimento, assim como forma das trajetórias exercem grande influência sobre o torque dos motores e sobre a amplitude do erro de acoplamento das juntas. Os valores dos fatores de velocidade e aceleração, bem como a terceira derivada do movimento em relação ao tempo constituem características relevantes na escolha do tipo de movimento.

Em polinômios onde a aceleração é descontínua, pode ocorrer, em algumas aplicações, efeitos indesejados nas cadeias cinemáticas e na inércia dos corpos. Com o objetivo de obter trajetórias com acelerações contínuas, um polinômio de quinto grau deve ser adotado para que seja possível impor as condições de posição, velocidade, e aceleração no início e no fim do movimento (BIAGIOTTI; MELCHIORRI, 2008).

Conforme Biagiotti e Melchiorri (2008), considerando um movimento escalar $q(t)$, podendo ser em $x(t)$, $y(t)$, $z(t)$ ou em coordenadas articulares $\theta_i(t)$, em um intervalo $t \in [t_0, t_1]$, o polinômio de quinto grau é dado pela Equação (2.13):

$$q(t) = a_0 + a_1 T + a_2 T^2 + a_3 T^3 + a_4 T^4 + a_5 T^5 \qquad (2.13)$$

Onde $T = t - t_0$, com as condições de contorno iguais a:

$$q(t_0) = q_0, \qquad q(t_1) = q_1$$

$$\dot{q}(t_0) = v_0, \qquad \dot{q}(t_1) = v_1$$

$$\ddot{q}(t_0) = a_0, \qquad \ddot{q}(t_1) = a_1$$

Com isso, obtém-se os coeficientes a_i, $i = 0, \dots, 5$ do polinômio (2.13), onde $h = q_1 - q_0$:

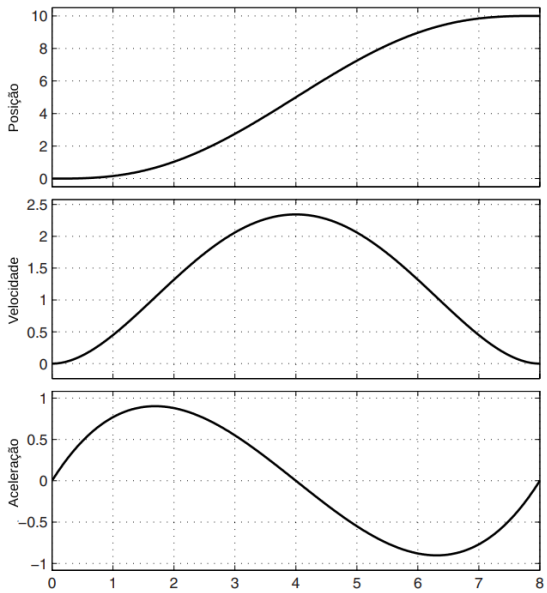
$$a_0 = q_0 \qquad a_3 = \frac{1}{2T^3[20h - (8v_1 + 12v_0)T - (3a_0 - a_1)T^2]}$$

$$a_1 = v_0 \qquad a_4 = \frac{1}{4T^4[-30h + (14v_1 + 16v_0)T + (3a_0 - 2a_1)T^2]}$$

$$a_2 = \frac{1}{2}a_0 \qquad a_5 = \frac{1}{2T^5[12h - 6(v_1 + v_0)T - (a_1 - a_0)T^2]}$$

Para operações de *pick-and-place*, como a que será utilizada no robô Delta, é comum adotar velocidades e acelerações nulas no início e no fim do movimento, produzindo transições mais suaves de q_0 até q_1, conforme a Figura 2.14 (FU; GONZALEZ; LEE, 1987).

Figura 2.14: Trajetória de Quinto Grau.



Fonte: Adaptado de Biagiotti e Melchiorri (2008).

2.5 Execução em Tempo Real e Acionamento por PULS/SIGN

A execução em tempo real refere-se à capacidade de gerar o comando de movimento com previsibilidade temporal, minimizando *jitter* no período dos pulsos e mantendo sincronismo multieixos. No sistema desenvolvido, a geração de pulsos é realizada no ESP32-S3, enquanto o MATLAB atua no planejamento e na transmissão de micro-blocos.

2.5.1 Interface PULS/SIGN do SGDV e quantização por pulsos

A execução de trajetórias em manipuladores robóticos acionados por *servodrives* industriais pode ser realizada por meio de referência em trem de pulsos, na qual a posição é comandada pelo número de pulsos aplicados na entrada do drive, enquanto o sentido do movimento é definido por um sinal digital de direção. No presente trabalho, o *servodrive* Yaskawa Σ-V (SGDV-1R6A01A) é operado no modo PULS/SIGN, em que o sinal PULS representa incrementos discretos de movimento e o sinal SIGN define o sentido de deslocamento (YASKAWA ELECTRIC CORPORATION, s. d.).

Como o comando dos servodrives é realizado por referência em trem de pulsos (PULS/SIGN), cada eixo é acionado por incrementos discretos de posição. No presente trabalho, a relação entre o pulso de comando e o deslocamento angular foi definida a partir da configuração de engrenagem eletrônica do *servodrive*, adotando-se uma resolução efetiva de comando de:

$$K_\theta = 0{,}001\ ^\circ/\text{pulso}$$

Dessa forma, o número de pulsos correspondente a um incremento angular $\Delta\theta_i$ no eixo i foi calculado por:

$$N_{pulsos} = \frac{\Delta\theta_i}{K_\theta} \tag{2.14}$$

em que K_θ representa a resolução angular efetiva configurada no acionamento. Essa abordagem permite que o planejamento no MATLAB e a verificação incremental no ESP32-S3 utilizem diretamente a relação prática entre pulsos de comando e deslocamento angular, sem depender explicitamente da resolução interna do *encoder* ou dos parâmetros internos do *servodrive* durante a geração dos frames.

A velocidade associada ao movimento é proporcional à frequência do trem de pulsos. Consequentemente, a aceleração corresponde à taxa de variação dessa

frequência ao longo do tempo, enquanto o deslocamento total é proporcional ao número acumulado de pulsos, sendo essencial que os sinais de comando respeitem as condições de interface e de temporização especificadas pelo fabricante (YASKAWA ELECTRIC CORPORATION, s. d.).

2.5.2 Executor em tempo real com RMT no ESP32-S3

Mesmo quando a trajetória é planejada com leis de movimento suaves, a qualidade do movimento pode ser degradada caso a geração do comando apresente variações relevantes no período entre pulsos (*jitter*) ou desalinhamento entre eixos, especialmente em robôs paralelos do tipo Delta (3T), em que a trajetória cartesiana depende do sincronismo entre juntas.

Para reduzir a influência de temporização baseada exclusivamente em *software*, utiliza-se o periférico RMT (*Remote Control Transceiver*) do ESP32-S3 como base do executor em tempo real. A documentação do Arduino-ESP32 descreve a inicialização do RMT com definição explícita de frequência (parâmetro *frequency_Hz*), o que permite selecionar a base de tempo com que o periférico representa durações de pulsos e, consequentemente, a granularidade temporal da geração de sinais (ESPRESSIF SYSTEMS, s. d.).

No nível do periférico, a documentação do ESP-IDF caracteriza o RMT como um recurso com múltiplos canais, aplicável como gerador de sequência de sinais e com suporte a transmissão simultânea em múltiplos canais, o que é diretamente relevante para reduzir defasagens de comando entre eixos em sistemas multieixos (ESPRESSIF SYSTEMS, s. d.). Além disso, o mesmo documento descreve o funcionamento do transmissor baseado em uma lista de valores que definem a duração e o nível lógico de pulsos, favorecendo a geração determinística de trens de pulso quando comparada a estratégias baseadas em laços de CPU (ESPRESSIF SYSTEMS, s. d.).

3 METODOLOGIA

3.1 Arquitetura do sistema proposto

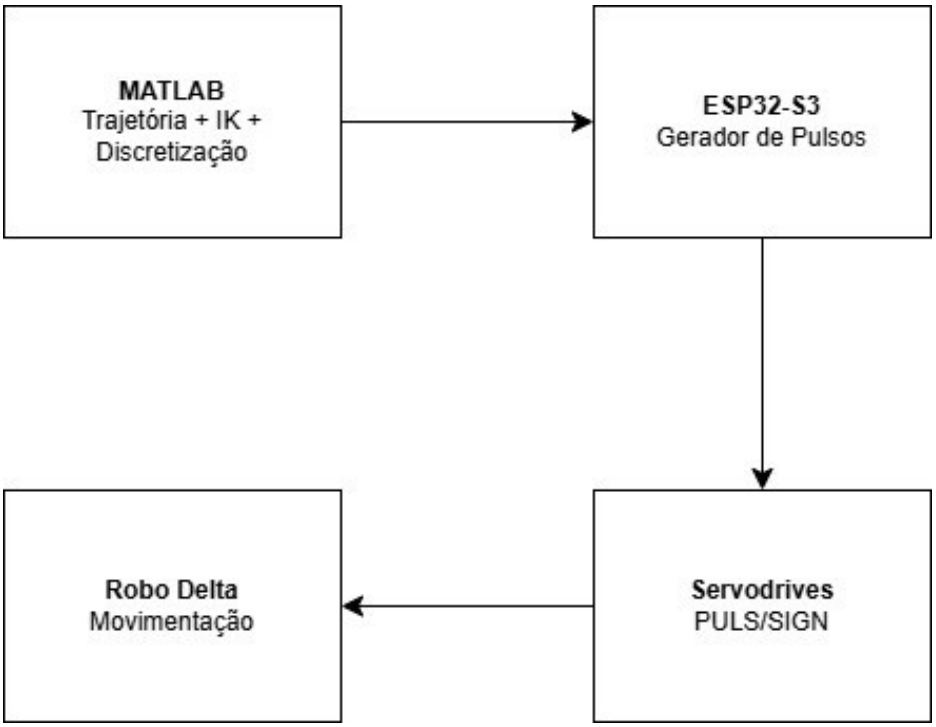
O sistema desenvolvido adota uma arquitetura híbrida, separando as funções de planejamento e execução em tempo real. O planejamento de trajetória é realizado em um computador utilizando MATLAB, responsável por: (i) gerar a trajetória cartesiana por lei de movimento suave (perfil de quinto grau no parâmetro de percurso), (ii) aplicar a cinemática inversa do robô Delta, (iii) converter os incrementos articulares em pulsos e direção, e (iv) empacotar os comandos em frames binários para transmissão.

A execução em tempo real é realizada por um microcontrolador ESP32-S3, que recebe os *frames* via UART sobre USB, valida integridade e alinhamento e armazena os comandos em uma fila circular. A geração dos sinais de comando para os servodrives *Yaskawa* Σ-V (modo PULS/SIGN) é feita localmente pelo ESP32-S3, utilizando o periférico RMT (*Remote Control Transceiver*) para temporização dos pulsos, reduzindo *jitter* e aumentando a previsibilidade da execução. Cada frame representa um micro-bloco com duração fixa T_s, definida no firmware como $DTUS = 4000\ \mu s$, garantindo que a trajetória seja executada como uma sequência de micro-blocos temporizados.

O protocolo de comunicação foi projetado para robustez em ambiente de laboratório, empregando sincronização por dois *bytes* de início de *frame* (SOF), contador de sequência, campos de direção e pulsos por eixo e verificação por CRC-8 (*Cyclic Redundancy Check*). Para evitar saturação de *buffer* durante o *streaming*, o *firmware* implementa controle de fluxo por *software* (XON/XOFF) baseado em limiares de ocupação da fila. A inicialização do sistema é realizada por um procedimento de referenciamento simultâneo utilizando três sensores ópticos, permitindo um ponto de partida conhecido e repetibilidade entre execuções.

Essa arquitetura reduz a carga computacional do microcontrolador durante o movimento (pois o cálculo complexo fica no MATLAB) e melhora o comportamento temporal do comando (pois os pulsos são gerados por hardware no ESP32-S3), fornecendo uma base adequada para movimentos suaves e repetíveis em aplicações do tipo *pick-and-place*. A Figura 3.1 mostra o fluxograma proposto para o projeto.

Figura 3.1: Arquitetura do Sistema de Controle Proposto.



Fonte: Elaborado pelo autor.

3.2 Descrição do sistema e hardware

3.2.1 Visão geral da bancada experimental

O sistema desenvolvido consiste em um robô paralelo do tipo Delta com três graus de liberdade translacionais (3T), acionado por três eixos servoacionados, destinados à execução de trajetórias suaves e repetíveis (ex.: *pick-and-place*), mostrado na Figura 3.2. A arquitetura de controle é dividida em duas camadas: (i) planejamento de trajetória no computador (MATLAB), responsável por gerar os micro-blocos de comando; e (ii) execução em tempo real no microcontrolador ESP32-S3, que recebe os dados via UART sobre USB, valida os *frames* e gera localmente os sinais de referência por trem de pulsos para cada eixo.

O desenvolvimento computacional deste trabalho foi realizado utilizando o software MATLAB R2026b, empregado na implementação dos algoritmos de cinemática, planejamento de trajetórias, discretização dos comandos em pulsos e comunicação serial com o controlador embarcado. Os ensaios computacionais foram

executados em um computador portátil equipado com processador Intel® Core™ i5-9300H com frequência base de 2,40 GHz, 16 GB de memória RAM DDR4 a 2400 MHz, unidade de armazenamento SSD de 500 GB e sistema operacional Windows 11 Home (64 bits). Essa configuração mostrou-se suficiente para a realização das simulações numéricas, processamento das trajetórias e execução dos experimentos computacionais apresentados neste trabalho.

Figura 3.2: Estrutura do Robô Delta do Laboratório de Automação e Robótica.

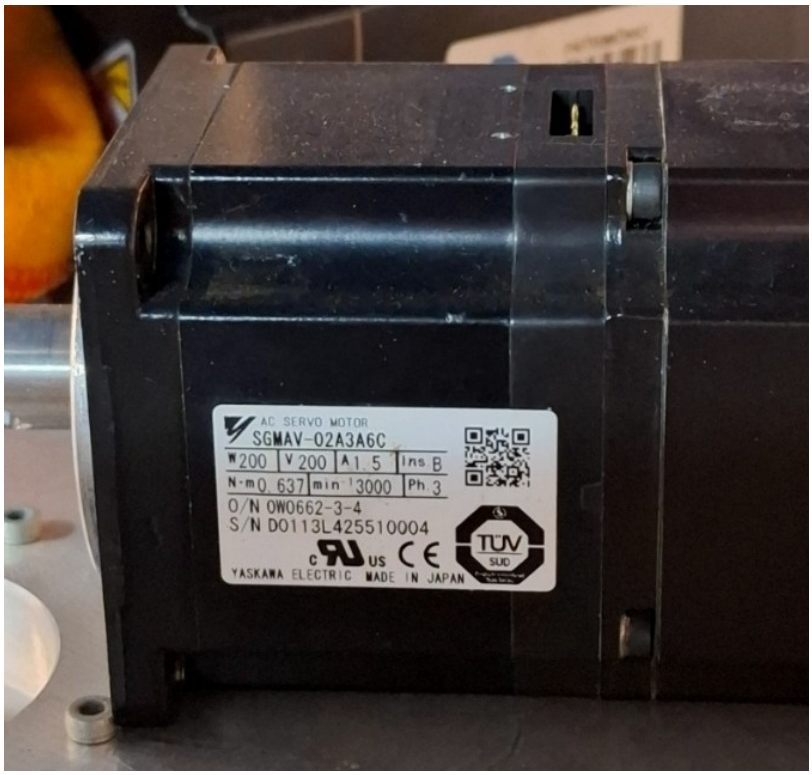
Fonte: Elaborado pelo autor.

A atuação é realizada por três conjuntos *servomotor* + *servodrive*, sendo utilizados *servodrives* Yaskawa Σ-V (modelo SGDV-1R6A01A) (Figura 3.3) operando no modo PULS/SIGN, e servomotores Yaskawa SGMAV-02A3A6C (Figura 3.4). Os *servodrives* são alimentados em 220 VAC trifásico. O diagrama de blocos representativo do sistema pode ser observado na Figura 3.5.

Figura 3.3: Servodrives Yaskawa montados no painel.

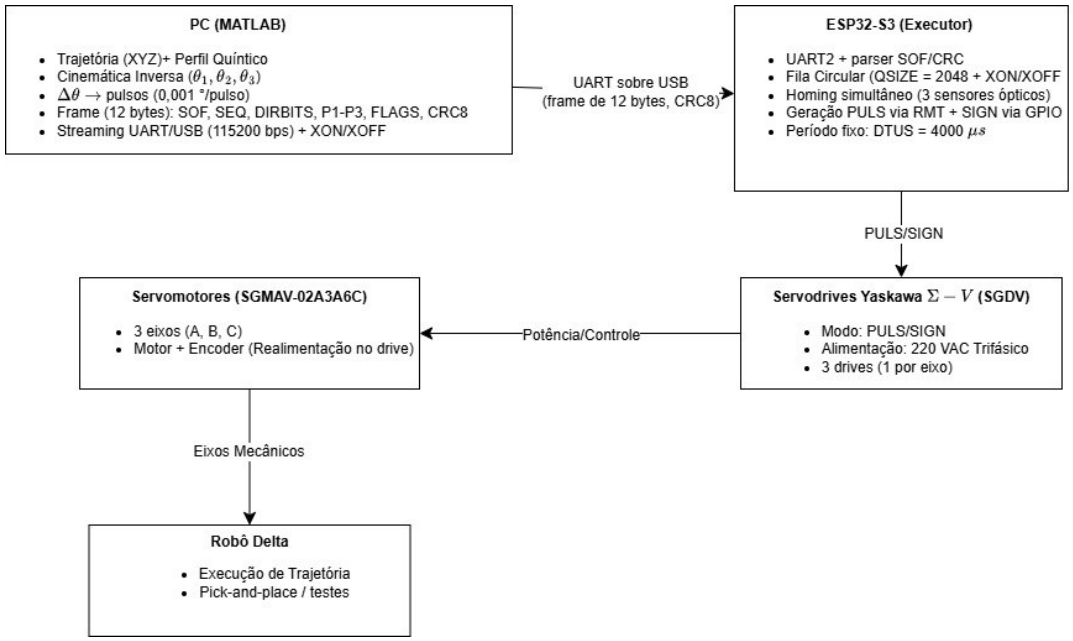
Fonte: Elaborado pelo autor.

Figura 3.4: Servo Motor Yaskawa utilizado na montagem.



Fonte: Elaborado pelo autor.

Figura 3.5: Diagrama de blocos da arquitetura.



Fonte: Elaborado pelo autor.

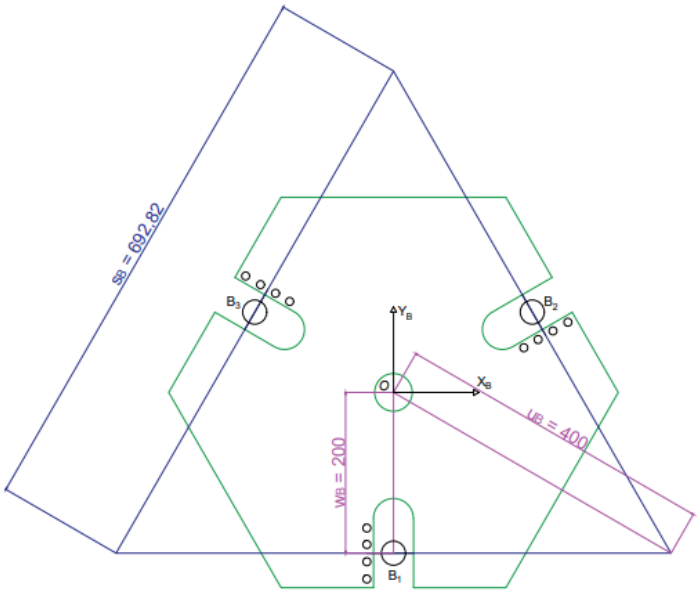
3.2.2 Parâmetros Geométricos Empregados

A estrutura mecânica do robô Delta utilizada neste trabalho é descrita por um conjunto de parâmetros geométricos que definem as dimensões relativas entre a base fixa, a plataforma móvel e os elos do mecanismo. Esses parâmetros são essenciais para a implementação da cinemática inversa, para a verificação de limites articulares e para a geração de trajetórias no espaço cartesiano. No presente projeto, os parâmetros foram adotados conforme a configuração disponível no Laboratório de Automação e Robótica e utilizados diretamente no MATLAB e no *firmware*.

As dimensões utilizadas para os cálculos de cinemática inversa da base fixa e da plataforma móvel do robô presente no laboratório são demonstradas na Figura 3.6 e na Figura 3.7. Para adequar o modelo real ao modelo de cinemática inversa utilizado, foi necessário aproximar as geometrias reais das bases do robô a formas triangulares equivalentes e posteriormente realizar a medição das dimensões características. Dessa forma, s_B e s_P são tratados como dimensões equivalentes associadas aos triângulos da base e da plataforma, respectivamente, enquanto l_1 e l_2 representam os comprimentos efetivos dos elos rígidos empregados no mecanismo. Os valores adotados para o protótipo são apresentados na Tabela 3.1, e todos os

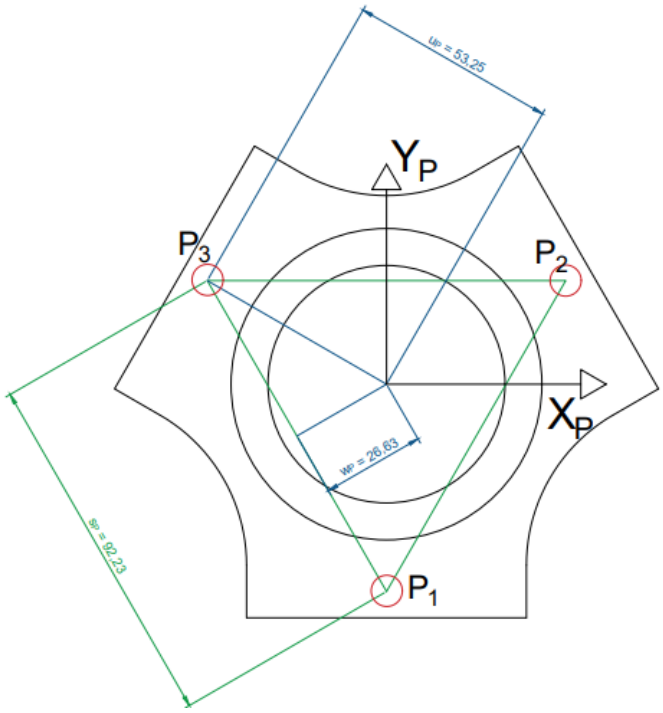
cálculos subsequentes de cinemática e planejamento de trajetória utilizam essas grandezas.

Figura 3.6: Dimensões da Base Fixa.



Fonte: Elaborado pelo autor.

Figura 3.7: Dimensões da Plataforma Móvel



Fonte: Elaborado pelo autor.

Para manter consistência de unidades, todos os parâmetros geométricos são expressos em metros e todas as coordenadas cartesianas (x, y, z) da trajetória são definidas no mesmo referencial $\{0\}$ fixo na base do robô.

Tabela 3.1: Parâmetros geométricos do robô Delta (protótipo).

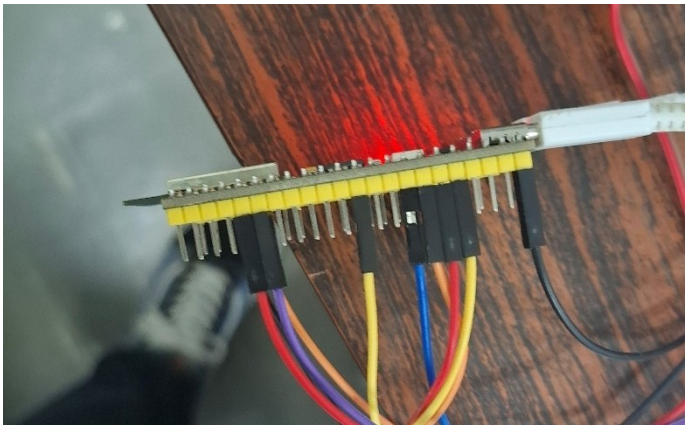
Parâmetro	Descrição	Valor	Unidade
s_B	Lado do triângulo da base superior	0,692820	m
s_P	Lado do triângulo da base inferior	0,09223	m
l_1	Comprimento do braço superior (elo acionado)	0,35	m
l_2	Comprimento do elo inferior (haste inferior)	0,84	m

Fonte: Elaborado pelo autor.

3.2.3 Eletrônica de Controle

O microcontrolador ESP32-S3, mostrado na Figura 3.8, é responsável por executar o comando de trajetória em tempo real, gerando os sinais PULS/SIGN para os três eixos. Para reduzir *jitter* e dependência de temporização em software, a geração do trem de pulsos é baseada no periférico RMT.

Figura 3.8: Microcontrolador ESP32-S3 utilizado na bancada.

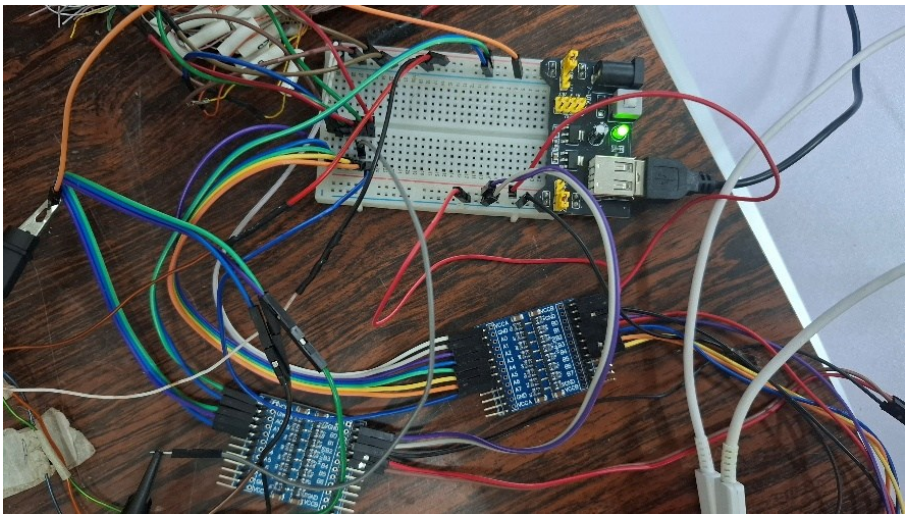


Fonte: Elaborado pelo autor.

Na camada Arduino-ESP32, o RMT é configurado com uma frequência de "*tick*" (*frequency_Hz*), que define a resolução temporal com que durações de pulsos são representadas.

Como o ESP32-S3 opera em 3,3 V, foi utilizado um conversor de nível lógico bidirecional de 8 canais (CNL8) para elevar os sinais de comando para 5 V, facilitando a compatibilidade com a interface digital do drive e dos sensores ópticos, conforme mostrado na Figura 3.9.

Figura 3.9: Bancada Experimental.

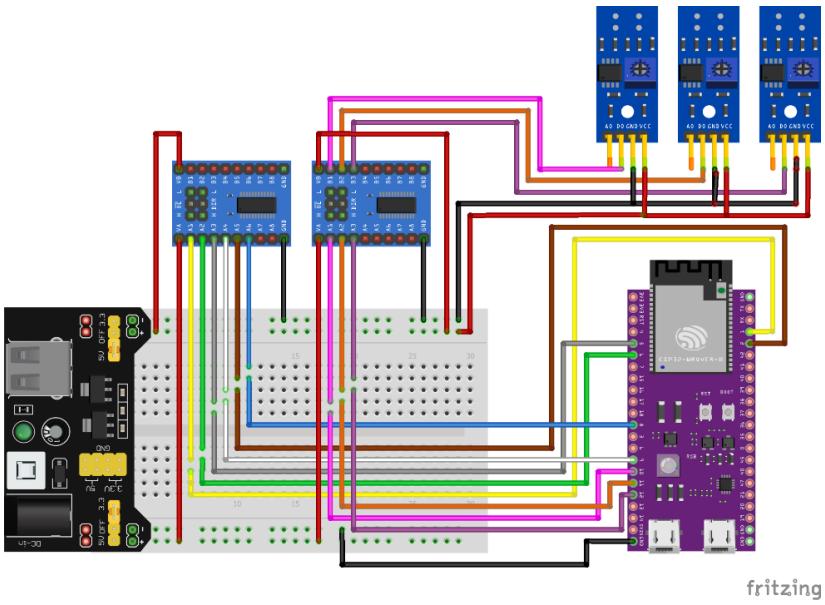


Fonte: Elaborado pelo autor.

A Figura 3.10 apresenta o esquema simplificado de ligação utilizado na bancada experimental. Para cada eixo, foram utilizados dois sinais principais: PULS, responsável pelos incrementos discretos de posição, e SIGN, responsável pela definição do sentido de movimento. Os terminais complementares /PULS e /SIGN foram referenciados ao GND, caracterizando uma ligação em modo simples referenciada ao terra comum do sistema.

Além dos sinais de comando, três sensores ópticos foram conectados às entradas digitais do ESP32-S3, sendo um sensor por eixo. Esses sensores são utilizados exclusivamente no procedimento de referenciamento inicial, permitindo que o sistema estabeleça uma condição mecânica conhecida antes da execução das trajetórias planejadas. O posicionamento dos sensores ópticos na estrutura é apresentado na Figura 3.11. Dessa forma, a ligação elétrica adotada integra a camada de planejamento, a unidade embarcada de execução, os circuitos de adaptação de nível e os dispositivos de acionamento dos eixos.

Figura 3.10: Esquema simplificado de ligação do ESP32-S3, conversor de nível e sensores ópticos.



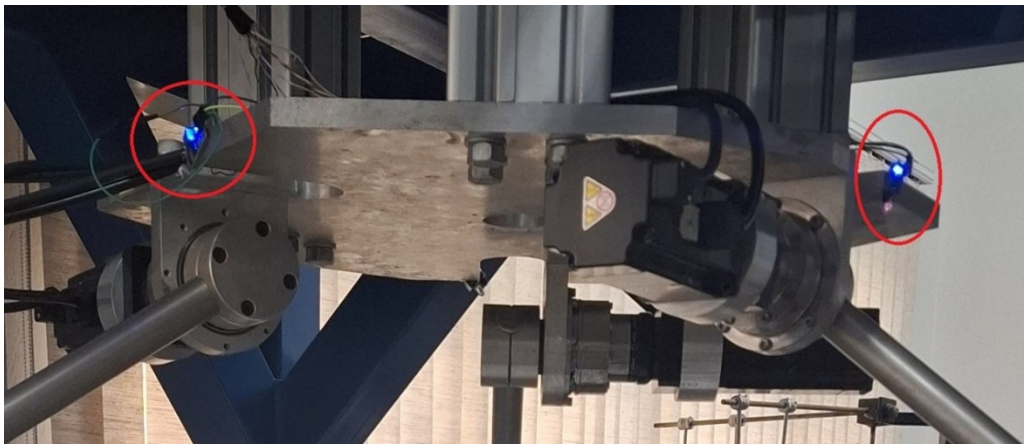
Fonte: Elaborado pelo autor.

Tabela 3.2: Conexões principais do sistema de comando e referenciamento.

Função	GPIO ESP32-S3	Conversor CNL8	Destino	Observação
PULS eixo 1	GPIO 1	LV1 → HV1	PULS eixo 1	Sinal de pulso
SIGN eixo 1	GPIO 6	LV2 → HV2	SIGN eixo 1	Sentido de giro
PULS eixo 2	GPIO 5	LV3 → HV3	PULS eixo 2	Sinal de pulso
SIGN eixo 2	GPIO 9	LV4 → HV4	SIGN eixo 2	Sentido de giro
PULS eixo 3	GPIO 2	LV5 → HV5	PULS eixo 3	Sinal de pulso
SIGN eixo 3	GPIO 8	LV6 → HV6	SIGN eixo 3	Sentido de giro
/PULS	GND	—	/PULS dos drives	Referenciado ao GND
/SIGN	GND	—	/SIGN dos drives	Referenciado ao GND
Sensor Eixo 1	GPIO 10	HV1 → LV1 (Conversor 2)	Entrada Digital ESP32-S3	Referência inicial eixo 1
Sensor Eixo 2	GPIO 11	HV2 → LV2 (Conversor 2)	Entrada Digital ESP32-S3	Referência inicial eixo 2
Sensor Eixo 3	GPIO 12	HV3 → LV3 (Conversor 2)	Entrada Digital ESP32-S3	Referência inicial eixo 3

Fonte: Elaborado pelo autor.

Figura 3.11: Posicionamento dos Sensores Ópticos na Estrutura.



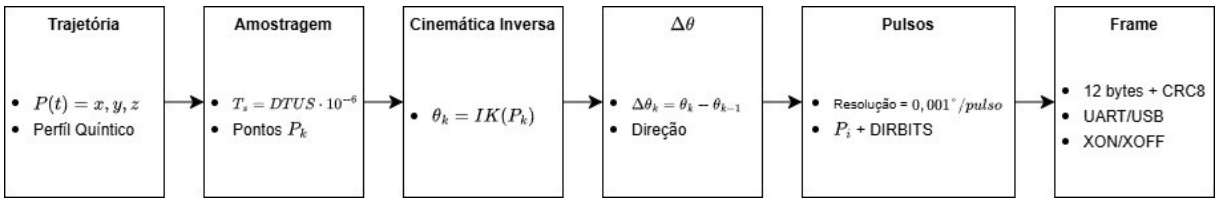
Fonte: Elaborado pelo autor.

3.3 Planejamento de Trajetória no MATLAB

3.3.1 Visão geral do *pipeline* de geração de comandos

O planejamento e a modelagem adotados neste trabalho têm como objetivo transformar uma trajetória cartesiana do robô Delta 3T em uma sequência de comandos discretos compatíveis com a interface PULS/SIGN dos *servodrives*. O processamento é realizado no MATLAB e segue o pipeline: (i) definição de trajetória $P(t) = [x(t), y(t), z(t)]$ com parametrização suave, (ii) amostragem em período fixo T_s, (iii) cálculo da cinemática inversa para obter θ_k, (iv) conversão dos incrementos articulares em pulsos e direção por micro-bloco, com mitigação de erro de quantização, e (v) empacotamento e transmissão em *frames* binários com detecção de erro e controle de fluxo. Esse encadeamento é apresentado na Figura 3.12.

Figura 3.12: Fluxo de modelagem e geração de comandos (trajetória → IK → pulsos → frame).



Fonte: Elaborado pelo autor.

Os parâmetros de discretização, conversão para pulsos e protocolo de comunicação, como $DTUS$, T_s, K_θ, limites articulares, frequência máxima e capacidade da fila circular, são apresentados na Tabela 3.3.

Tabela 3.3: Parâmetros do comando discreto e do executor embarcado.

Parâmetro	Descrição	Valor	Unidade
$\boldsymbol{DTUS}$	Período do micro-bloco no firmware	4000	μs
$\boldsymbol{T_s}$	Período de amostragem ($T_s = DTUS \cdot 10^{-6}$)	0,004	s
Baud	Taxa UART (MATLAB ↔ ESP32-S3)	115200	bps
Frame	Tamanho do frame binário	12	bytes
CRC	CRC-8 (polinômio)	0x07	—
$\boldsymbol{K_\theta}$	Resolução efetiva da engrenagem eletrônica	0,001	°/pulso
$\boldsymbol{\theta_{min}}$	Limite articular inferior	-30	°
$\boldsymbol{\theta_{max}}$	Limite articular superior	75	°
$\boldsymbol{LIM_{MAX}}$	Limite superior equivalente em pulsos $(\theta_{max} - \theta_{min})/K_\theta$	105000	pulsos
$\boldsymbol{F_{max}}$	Limite de frequência adotado no planejamento	100000	Hz
$\boldsymbol{P_{seg}}$ **(MATLAB)**	Limite de pulsos por micro-bloco (planejamento)	2000	pulsos
$\boldsymbol{P_{seg}}$ **(firmware)**	Limite de pulsos por micro-bloco (executor)	2000	pulsos
$\boldsymbol{Q_{SIZE}}$	Capacidade da fila circular no ESP32-S3	2048	segmentos
$\frac{\boldsymbol{Q_{HIGH}}}{\boldsymbol{Q_{LOW}}}$	Limiares do XON/XOFF	1638/800	segmentos

Fonte: Elaborado pelo autor.

3.3.2 Convenções, variáveis e parâmetros do protótipo

A modelagem matemática adotada neste trabalho tem como objetivo estabelecer uma relação operacional entre a trajetória definida no espaço cartesiano e os comandos discretos enviados aos *servodrives*, permitindo que trajetórias planejadas no MATLAB sejam executadas pelo sistema embarcado de forma segura e repetível. Como o manipulador é um robô Delta com três graus de liberdade translacionais (3T), a trajetória é definida no espaço cartesiano por $P(x, y, z)$ e

convertida para o espaço articular $\theta = [\theta_1, \theta_2, \theta_3]^T$ por meio do modelo de cinemática inversa apresentado no Capítulo 2 (Seção 2.3).

Adota-se um sistema de coordenadas fixo na base $\{O\}$, com z orientado verticalmente e x, y definindo o plano horizontal. A posição da plataforma móvel é representada pelo ponto $P(x, y, z)$ (centro da plataforma).

A execução do sistema depende de uma condição inicial conhecida; por isso, o procedimento de referenciamento inicial conduz cada eixo a uma configuração mecânica repetível, detectada pelos sensores ópticos apresentados na Seção 3.2.3 e tratada no firmware conforme a Seção 3.4.3. Após a conclusão do referenciamento, o firmware habilita a execução dos micro-blocos e define os contadores incrementais de posição como referência inicial. Dessa forma, o referenciamento não tem caráter de correção por offsets geométricos, mas de estabelecimento de uma condição inicial conhecida e rastreável para a execução das trajetórias planejadas.

3.3.3 Cinemática Inversa e critérios operacionais de validade

Para cada ponto $Q_k = [x_k, y_k, z_k]$, o MATLAB calcula os ângulos articulares:

$$\theta_k = IK(Q_k)$$

utilizando as equações apresentadas na Seção 2.3 (Equações (2.3) – (2.8)). Na implementação, antes de qualquer conversão em pulsos, o algoritmo verifica a validade numérica da solução, interrompendo a execução se existirem valores não finitos ($"NaN"$ ou ∞). Em seguida, aplica-se a validação por faixa articular:

$$\theta_{\min} \leq \theta_{i,k} \leq \theta_{\max}, i \in \{1,2,3\}$$

com $\theta_{\min} = -30^\circ$ e $\theta_{\max} = 75^\circ$ (Tabela 3.3). Essa etapa garante que a trajetória planejada não exceda os limites definidos para operação do protótipo, sem necessidade de mapear integralmente o espaço de trabalho.

3.3.4 Parametrização suave da trajetória e amostragem temporal

O *script* implementa a geração de trajetória cartesiana com base em um perfil quíntico de quinto grau, aplicado separadamente a cada coordenada. Para uma coordenada escalar $q(t) \in \{x(t), y(t), z(t)\}$, a função auxiliar `quintic_1d(q0,qf,T,t)` utiliza:

$$\tau = \frac{t}{T}, \qquad q(t) = q_0 + (q_f - q_0)(10\tau^3 - 15\tau^4 + 6\tau^5) \tag{3.1}$$

A equação (3.1) garante início e fim com variação suave (derivadas nulas nas extremidades do parâmetro), sendo adequada para movimentos do tipo *pick-and-place*. O vetor de tempo é construído por:

$$t = 0{:}\,T_s{:}\,T$$

$$N = |\ t\ |$$

e a trajetória cartesiana discretizada é organizada na matriz:

$$Q = \begin{bmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_N & y_N & z_N \end{bmatrix}$$

A execução é discretizada em micro-blocos de duração fixa:

$$T_s = DTUS \cdot 10^{-6}$$

No protótipo, $DTUS = 4000\ \mu s$ e $T_S = 4\ ms$ (Tabela 3.3). A trajetória é amostrada em:

$$t_k = t_0 + kT_s, \qquad k = 0{,}1, \ldots, N-1, \qquad N = \left[\frac{T}{T_s}\right] + 1$$

resultando na sequência de pontos $Q_k = [x_k, y_k, z_k]$, que servem como alvos cartesianos para cada micro-bloco.

3.3.5 Conversão para pulsos, quantização e limites no executor

Para cada amostra válida Q_k, obtém-se θ_k por cinemática inversa e calcula-se o incremento articular:

$$\Delta\theta_{i,k} = \theta_{i,k} - \theta_{i,k-1}$$

O comando dos drives é realizado por PULS/SIGN, portanto os incrementos articulares calculados pela cinemática inversa são convertidos em pulsos a partir da resolução efetiva configurada nos *servodrives*. Para isso, define-se a posição articular relativa:

$$\theta_{rel,i,k} = \theta_{i,k} - \theta_{min}$$

e a posição desejada em pulsos:

$$u_{i,k} = \frac{\theta_{rel,i,k}}{K_\theta}$$

em que $K_\theta = 0{,}001\ °/\text{pulso}$. Em seguida, calcula-se o incremento entre amostras consecutivas:

$$\Delta u_{i,k} = u_{i,k} - u_{i,k-1}$$

Como os comandos enviados ao *firmware* devem ser inteiros, foi utilizada uma estratégia de quantização com acumulação de erro, na qual a parcela fracionária não executada em um micro-bloco é preservada e somada ao incremento do bloco seguinte. Para cada eixo, o incremento real $\Delta u_{i,k}$ é somado ao erro acumulado anterior:

$$y_{i,k} = \Delta u_{i,k} + e_{i,k-1}$$

O incremento inteiro a ser executado no micro-bloco é obtido por arredondamento:

$$dq_{i,k} = round(y_{i,k})$$

e o erro remanescente é preservado para o próximo micro-bloco:

$$e_{i,k} = y_{i,k} - dq_{i,k}$$

O número de pulsos enviado ao ESP32-S3 é dado por:

$$p_{i,k} = | \, dq_{i,k} \, |$$

enquanto o *bit* de direção é definido pelo sinal de $dq_{i,k}$. Dessa forma, mesmo que alguns micro-blocos resultem em emissão nula de pulsos, o erro fracionário não é descartado, sendo redistribuído nos blocos seguintes e reduzindo o erro acumulado ao final da trajetória.

A frequência equivalente de pulsos durante o micro-bloco é estimada por:

$$f_{i,k} \approx \frac{p_{i,k}}{T_s}$$

No script MATLAB, o comando é validado antes da transmissão, verificando-se se a quantidade de pulsos por micro-bloco não excede $P_{seg,max} = 2000$ pulsos e se a frequência equivalente permanece abaixo de $F_{MAX} = 100000$ Hz. No *firmware*, a proteção é complementar e atua sobre os pulsos recebidos em cada frame, rejeitando segmentos que excedam o limite máximo definido para o executor. Além disso, o ESP32-S3 mantém contadores incrementais de posição em pulsos, utilizados para evitar deslocamentos acumulados fora da faixa permitida após o referenciamento inicial.

Em testes preliminares do algoritmo de discretização, observou-se que trajetórias de baixa velocidade podiam gerar incrementos articulares muito pequenos entre amostras consecutivas, resultando em uma quantidade elevada de micro-blocos com emissão nula de pulsos em um ou mais eixos. Esse comportamento decorre diretamente da quantização do comando em pulsos inteiros e pode comprometer a

continuidade percebida do movimento, especialmente em trechos lentos ou de pequena variação angular. Para reduzir esse efeito, foram realizados ajustes na relação entre deslocamento angular e pulsos efetivamente emitidos, bem como refinamentos na lógica do acumulador de erro, com o objetivo de redistribuir os pulsos de forma mais uniforme ao longo da trajetória discretizada e diminuir a ocorrência de blocos consecutivos com comando nulo.

3.3.6 Empacotamento, transmissão e recepção dos *frames*

Os comandos discretos por micro-bloco são empacotados em um *frame* binário de 12 *bytes*, transmitido do MATLAB para o ESP32-S3 via UART sobre USB. Cada *frame* contém dois *bytes* de sincronização, contador de sequência, *bits* de direção, quantidade de pulsos por eixo, campo de *flags* e CRC-8. A estrutura geral do *frame* é composta por SOF1 = 0xAA, SOF2 = 0x55, SEQ, DIRBITS, P1, P2 e P3 em formato *uint16 little-endian,* FLAGS e CRC-8, conforme apresentado na Figura 3.13.

Figura 3.13: Estrutura do frame binário de comando (12 bytes)

SOF1 Byte 1 0xAA	**SOF2** Byte 2 0x55	**SEQ** Byte 3 0-255	**DIRBITS** Byte 4 bits 0..2	**P1** Byte 5-6 uint16 Litlle- endian	**P2** Byte 7-8 uint16 Litlle- endian	**P3** Byte 9-10 uint16 Litlle- endian	**FLAGS** Byte 11 Bit 0 = END	**CRC8** Byte 12 Bytes 1-11

Fonte: Elaborado pelo autor.

O campo DIRBITS armazena o sentido de movimento de cada eixo, enquanto os campos P1, P2 e P3 representam a quantidade de pulsos a serem emitidos em cada micro-bloco. O campo FLAGS é utilizado para indicar condições especiais de transmissão, como a marcação do último *frame* por meio da *flag* END. O CRC-8, calculado com polinômio 0x07 sobre os *bytes* 1–11 do *frame*, é utilizado para validar a integridade dos dados antes de sua inserção na fila de execução do *firmware*, seguindo o princípio de verificação de redundância cíclica aplicado à detecção de erros em comunicação digital (WILLIAMS, 1993).

Durante o *streaming*, o MATLAB envia os *frames* sequencialmente pela porta serial configurada, enquanto o ESP32-S3 acumula os *bytes* recebidos em um *buffer* de recepção. No *firmware*, um *parser* busca os *bytes* de sincronização SOF para identificar o início de cada *frame*, valida o CRC-8 antes de aceitar o pacote, descarta *frames* corrompidos e realiza nova sincronização caso ocorra falha de integridade.

Após a validação, os campos de direção e pulsos são decodificados e o segmento é inserido na fila circular de execução.

Para evitar saturação da fila interna do ESP32-S3, foi implementado controle de fluxo por *software* XON/XOFF. Quando a ocupação da fila ultrapassa o limiar superior, o microcontrolador envia XOFF (0x13), solicitando que o MATLAB pause temporariamente a transmissão. Quando a ocupação retorna abaixo do limiar inferior, o ESP32-S3 envia XON (0x11), autorizando a retomada do envio. Esse mecanismo desacopla a velocidade de transmissão dos *frames* da velocidade de execução dos micro-blocos, reduzindo a possibilidade de *overflow* da fila.

Ao final da transmissão, o último *frame* é marcado com a *flag* END. A execução normal é concluída quando esse *frame* é recebido e todos os segmentos armazenados na fila são processados. Nessa condição, o ESP32-S3 envia a mensagem “DONE” ao MATLAB, indicando que a trajetória foi completamente executada.

3.4 Implementação no ESP32-S3

3.4.1 Arquitetura do *firmware* e temporização determinística

O ESP32-S3 atua como executor em tempo real, recebendo do MATLAB uma sequência de micro-blocos discretos e gerando localmente os sinais de referência PULS/SIGN para os *servodrives*. O *firmware* foi desenvolvido com o *framework* Arduino-ESP32 e organizado em uma estrutura de controle orientada a estados, separando as rotinas de recepção, execução, referenciamento e tratamento de falhas. Os estados principais são: IDLE (espera), BIN_RX (recepção e armazenamento dos frames), RUNNING (execução dos micro-blocos), HOMING (referenciamento inicial) e ERROR (falha).

A temporização do sistema é definida por um período fixo por micro-bloco, estabelecido no *firmware* como:

$$DTUS = 4000\ \mu s$$

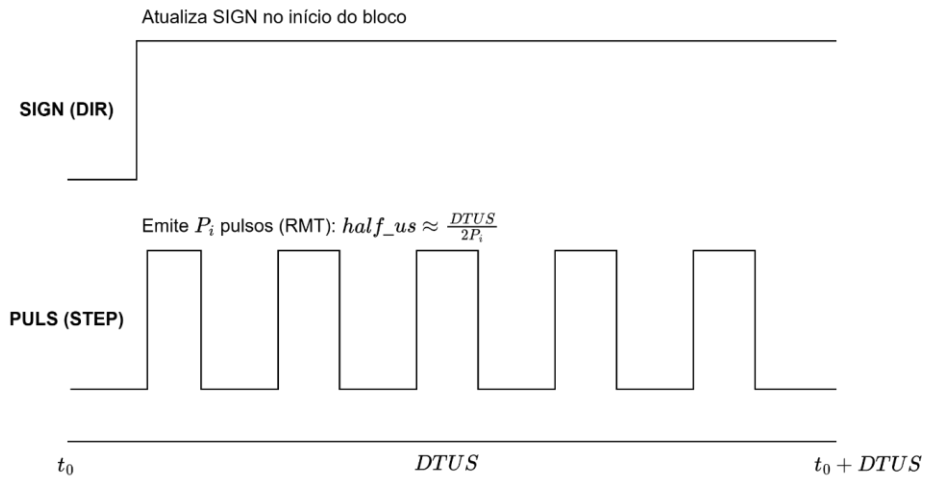
De forma que cada segmento de movimento tem duração aproximadamente constante. Essa escolha reduz a complexidade do sincronismo temporal, pois o MATLAB envia “quanto mover” (pulsos por eixo) e o ESP32-S3 garante “quando mover” (tempo de execução do micro-bloco). Ao final de cada micro-bloco, o *firmware* mede o tempo decorrido e, caso a emissão de pulsos termine antes do tempo nominal,

aplica uma espera complementar para completar $DTUS$, mantendo previsibilidade na execução.

3.4.2 Executor PULS/SIGN multieixos e sincronismo

A execução física do movimento é realizada por geração de sinais digitais: SIGN define o sentido e PULS define o deslocamento incremental, em pulsos. Para reduzir *jitter* e dependência de temporização por *software*, a geração do trem de pulsos é feita pelo periférico RMT, configurado com resolução de 1 μs por "*tick*". O sinal SIGN é atualizado por GPIO antes do início da emissão dos pulsos do micro-bloco, conforme ilustrado na Figura 3.14.

Figura 3.14: Diagrama temporal simplificado de um micro-bloco ($DTUS$) em um eixo.



Fonte: Elaborado pelo autor.

Para cada micro-bloco, o *firmware* recebe P_i pulsos a emitir no eixo i. O semi-período associado ao trem de pulsos é calculado por:

$$half_{us} \approx \frac{DTUS}{2P_i}$$

e o RMT emite uma sequência de P_i itens com duração *half_us* em nível alto e *half_us* em nível baixo. Após iniciar a transmissão em cada canal, o *firmware* aguarda a finalização e então completa o tempo restante até $DTUS$.

O sincronismo multieixos é obtido por duas estratégias complementares:

1. micro-blocos com tempo fixo, garantindo que todos os eixos respeitem a mesma janela temporal $DTUS$;

2. início próximo simultâneo da emissão, iniciando a transmissão dos canais RMT em sequência curta, mantendo defasagem pequena entre eixos.

3.4.3 Estratégias de segurança e tratamento de falhas

O executor implementa verificações e mecanismos de segurança para prevenir execução fora da faixa operacional e lidar com falhas de comunicação ou temporização. As principais estratégias são:

a) Referenciamento inicial obrigatório: a execução de segmentos é permitida apenas após o procedimento de referenciamento inicial, indicado no *firmware* pela variável *homed=true*. O procedimento utiliza três sensores ópticos, configurados como entradas digitais com *pull-up* interno e acionamento em nível lógico baixo. Após o referenciamento, os contadores incrementais de posição são zerados, estabelecendo uma condição inicial conhecida para a execução das trajetórias.
b) Limites e validações durante a execução: antes de executar um micro-bloco, o *firmware* verifica se a quantidade de pulsos recebida em cada eixo não excede o limite máximo permitido por segmento. Além disso, o *firmware* mantém contadores incrementais de posição em pulsos, utilizados como proteção complementar contra deslocamentos acumulados fora da faixa definida para operação. A validação geométrica e articular da trajetória é realizada previamente no MATLAB, enquanto o ESP32-S3 atua como camada adicional de proteção durante a execução.
c) Controle de fluxo e proteção contra *overflow/underflow*: A fila circular desacopla recepção e execução. O XON/XOFF evita *overflow*; caso ocorra ausência de dados (*underflow*), o sistema permanece aguardando novos *frames*, evitando emitir pulsos fora da sequência prevista.
d) Término controlado e sinalização ao supervisor: A execução normal termina quando o *frame* com *flag* END é recebido e todos os segmentos da fila foram consumidos. Nessa condição, o ESP32-S3 envia “DONE” ao MATLAB. Em caso de STOP solicitado ou falha (violação de limites, *timeout* do RMT, inconsistências persistentes), o sistema entra no estado ERROR, interrompe a execução e reporta a condição.

Esses mecanismos garantem rastreabilidade do estado, reduzem risco de comandos inválidos e aumentam a robustez do sistema durante ensaios laboratoriais.

4 RESULTADOS E DISCUSSÕES

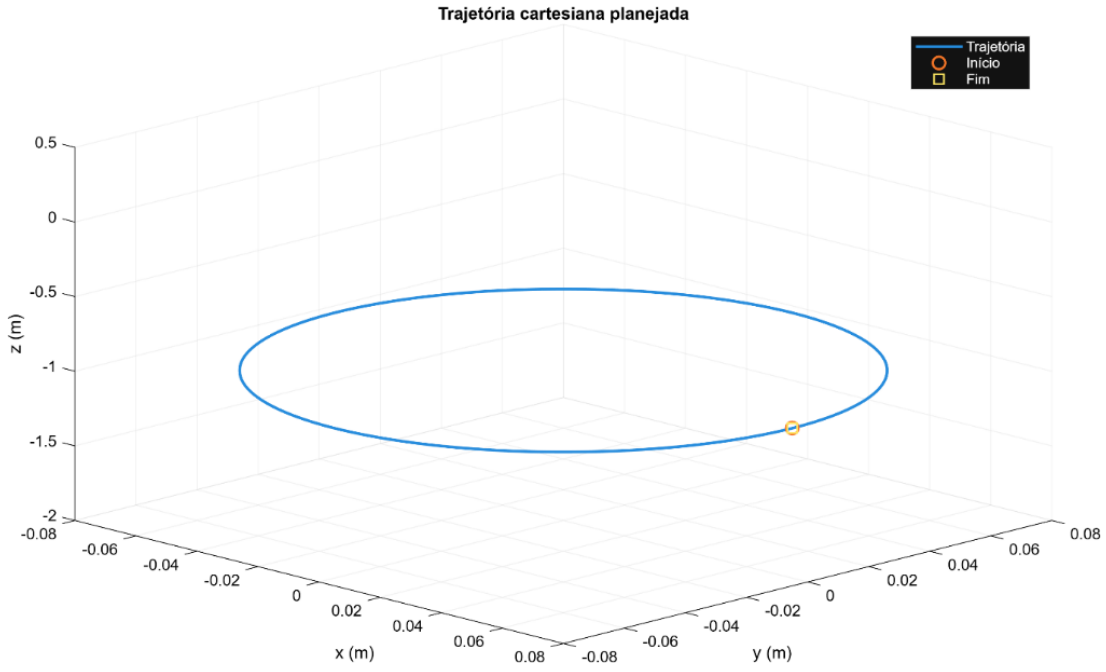
4.1 Validação numérica da trajetória e do comando discreto

4.1.1 Trajetória Cartesiana Planejada

Para a validação numérica, foi analisada uma trajetória circular no plano XY, com altura constante $z = -0{,}995\ m$, discretizada em micro-blocos compatíveis com o executor embarcado.

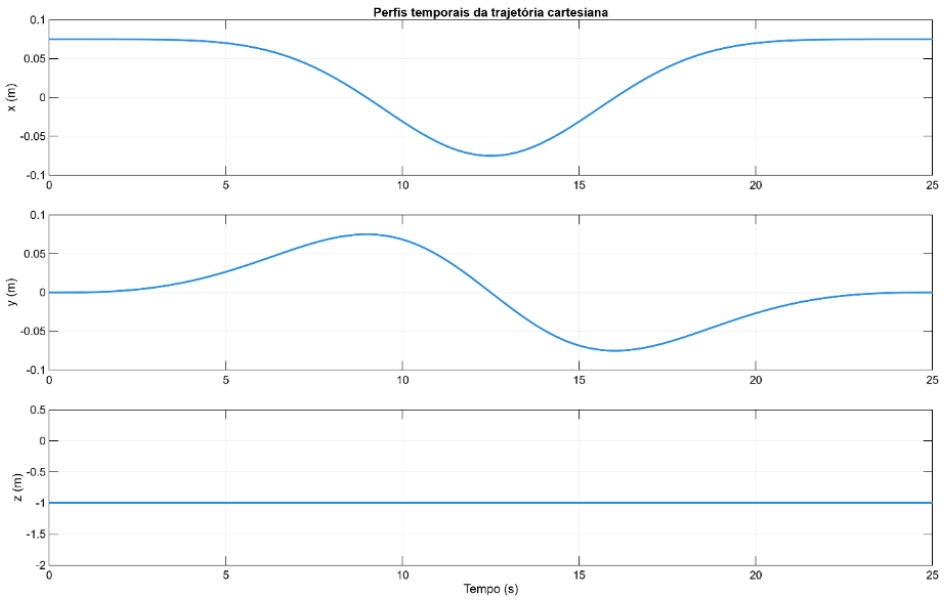
A Figura 4.1 apresenta a trajetória cartesiana no espaço tridimensional, enquanto a Figura 4.2 mostra os perfis temporais de $x(t)$, $y(t)$ e $z(t)$. A análise desses gráficos permite verificar a coerência geométrica do movimento planejado, a continuidade da curva e a suavidade associada ao perfil de parametrização utilizado. Os parâmetros utilizados para gerar essa trajetória são apresentados na Tabela 4.1.

Figura 4.1: Trajetória Circular Planejada.



Fonte: Elaborado pelo autor.

Figura 4.2: Perfis Temporais da Trajetória Cartesiana.



Fonte: Elaborado pelo autor.

Tabela 4.1: Parâmetros da Trajetória.

Parâmetros	Valores
$T_{tempo\ total}\ (s)$	25
$T_s\ (s)$	0,004
$N_{amostras}$	6251
$x_{\min}\ \ (m)$	-0,075
$x_{\max}\ \ (m)$	0,075
$y_{\min}\ \ (m)$	-0,075
$y_{\max}\ \ (m)$	0,075
$z(m)$	-0,995

Fonte: Elaborado pelo autor.

Observa-se que a trajetória planejada corresponde a uma volta circular completa no plano XY, com centro em $(0,0,-0{,}995)$ e altura constante. O raio da circunferência é de 75 milímetros, com uma discretização de 4 milissegundos, resultando em 6251 amostras para 25 segundos de movimento, o que a torna adequada para avaliar a coerência geométrica do planejamento sem introduzir variações simultâneas em z.

Na análise computacional, observou-se que a trajetória gerada apresentou comportamento contínuo e compatível com o tipo de movimento desejado, sem descontinuidades na posição. Ademais, a discretização adotada mostrou-se suficiente

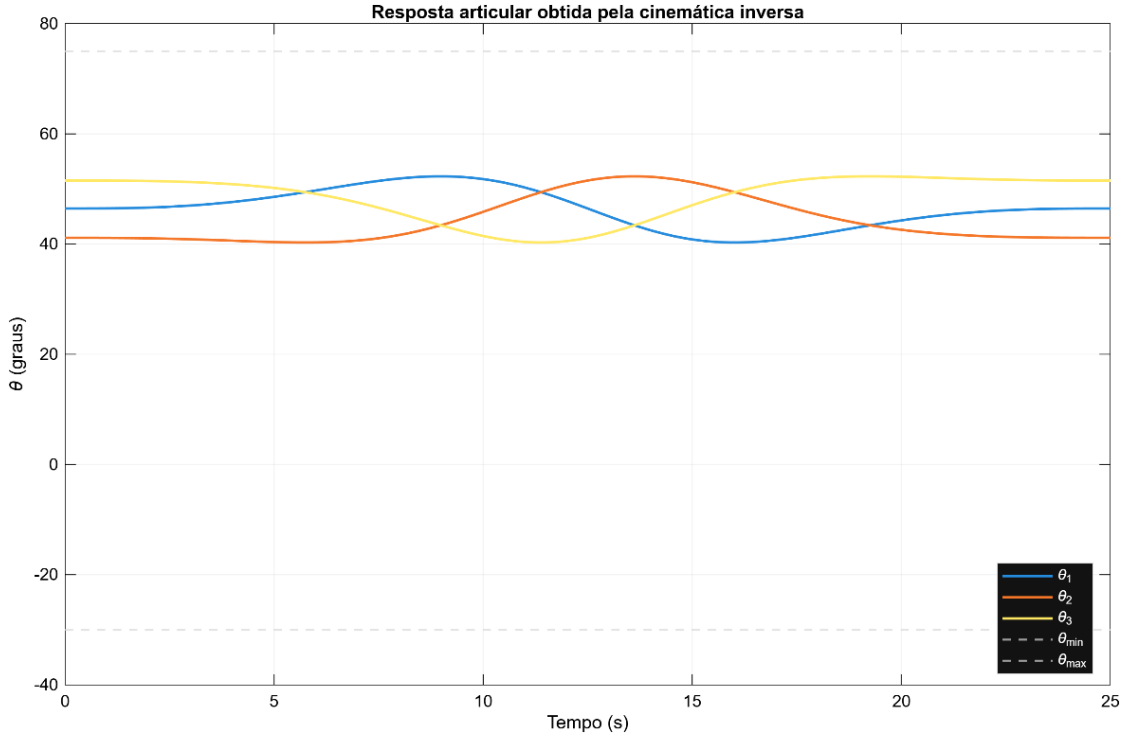
para apresentar numericamente o percurso sem alterações abruptas entre amostras consecutivas.

4.1.2 Cinemática inversa da trajetória circular

A partir da trajetória cartesiana discretizada da circunferência, foi aplicada a cinemática inversa para obtenção das variáveis articulares $\theta_1(t)$, $\theta_2(t)$ e $\theta_3(t)$. Essa etapa permitiu verificar se todos os pontos da trajetória eram geometricamente alcançáveis pelo protótipo e se os ângulos gerados permaneciam dentro da faixa articular estabelecida para operação segura.

A Figura 4.3 apresenta a evolução temporal dos ângulos articulares calculados para os três eixos. A partir desses resultados, foi possível verificar a inexistência de valores não finitos na solução numérica e o respeito aos limites articulares definidos para o sistema. A Tabela 4.2 resume os valores mínimos e máximos encontrados para cada junta ao longo da trajetória circular simulada.

Figura 4.3: Resposta Articular da trajetória circular obtida pela Cinemática Inversa.



Fonte: Elaborado pelo autor.

Tabela 4.2: Valores mínimos e máximos de cada junta

Juntas	Mínimo	Máximo	Limite Inferior	Limite Superior
θ_1	40.2851	52.2884	-30	75
θ_2	40.2851	52.2884	-30	75
θ_3	40.2851	52.2884	-30	75

Fonte: Elaborado pelo autor.

Os resultados mostraram que a trajetória analisada foi compatível com a geometria do robô, uma vez que a cinemática inversa produziu soluções válidas para todos os pontos amostrados. Ainda, os ângulos calculados permanecem dentro da faixa operacional adotada no planejamento, o que confirma a viabilidade da trajetória do ponto de vista articular. Como a trajetória escolhida é circular, centrada e geometricamente simétrica em relação ao arranjo dos três braços, os três eixos apresentaram a mesma faixa angular, diferindo apenas no instante em que cada extremo foi atingido.

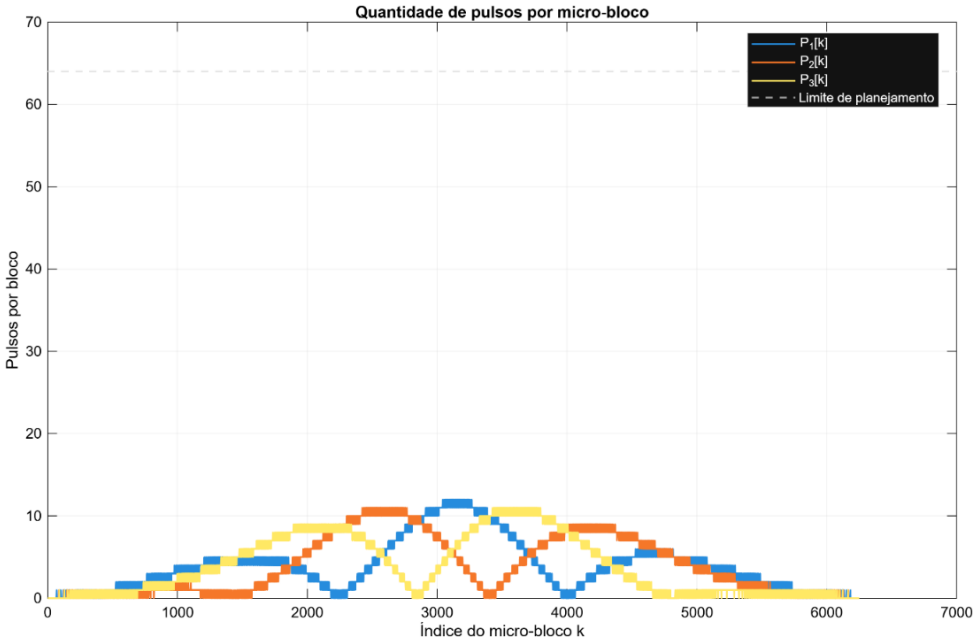
4.1.3 Discretização em pulsos e frequência por micro-bloco

Após a obtenção da resposta articular da trajetória circular, os incrementos entre amostras consecutivas foram convertidos em pulsos por micro-bloco, conforme o modelo de discretização apresentado no Capítulo 3. Nessa etapa, foi considerada a resolução angular efetiva de $0{,}001\ °$/pulso, definida pela configuração de engrenagem eletrônica dos *servodrives*.

Durante os ensaios iniciais, verificou-se que trajetórias muito lentas produziam grande quantidade de micro-blocos sem emissão de pulsos, principalmente em regiões onde a diferença angular entre pontos consecutivos era muito pequena. Na prática, esse efeito tendia a tornar o movimento menos contínuo do ponto de vista dos atuadores, apesar de a trajetória cartesiana planejada permanecer suave. Para mitigar esse problema, foram realizados ajustes na discretização do comando e na estratégia de quantização com acumulação do erro, de modo a reduzir a quantidade de blocos com pulsos nulos consecutivos. Após essas modificações, observou-se melhora na continuidade da execução em baixas velocidades, tornando o comportamento do robô mais fluido nos ensaios experimentais.

A Figura 4.4 apresenta a quantidade de pulsos por micro-bloco para os três eixos ao longo da circunferência. A partir desses resultados, foi possível verificar se os comandos gerados permaneciam dentro dos limites máximos definidos para o planejamento e para o executor embarcado. A Tabela 4.3 apresenta o resumo dos valores obtidos com a discretização.

Figura 4.4: Quantidade de pulsos por bloco.



Fonte: Elaborado pelo autor.

Tabela 4.3: Resumo da discretização do comando

Variáveis	Resultados
$f_{peak}\ (Hz)$	3000
$P_1 max$	12
$P_2 max$	11
$P_3 max$	11
Frames totais	6251
Frames totalmente nulos	539
Percentual de frames nulos (%)	8,62 %
Erro final em pulsos	[0, 0, 0]
Erro máximo absoluto (pulsos)	[0,5000; 0,5000; 0,4999]
Resolução Efetiva de Comando	0,001 °/pulso

Fonte: Elaborado pelo autor.

A análise mostrou que os pulsos por segmento permaneceram abaixo do limite de segurança adotado no algoritmo, e que a frequência de pico f_{peak} permaneceu abaixo do valor máximo estabelecido. Esses resultados indicam que a trajetória planejada não apenas era geometricamente válida, mas também compatível com a capacidade de execução do sistema, tanto do ponto de vista temporal quanto do ponto de vista da discretização.

Além da análise da quantidade de pulsos por micro-bloco e da frequência equivalente, foi realizada uma comparação entre a posição de referência em pulsos e a posição efetivamente executada na simulação discreta do comando. Para a trajetória circular analisada, observou-se a ocorrência de 539 frames totalmente nulos, correspondentes a 8,62 % do total de 6251 micro-blocos, o que indica que, mesmo após os ajustes no algoritmo de discretização, ainda existem segmentos sem emissão de pulsos em todos os eixos. Entretanto, a comparação entre a referência e a execução mostrou que o erro final em pulsos foi nulo nos três eixos, isto é, $[0, 0, 0]$ pulsos ao final da trajetória. Adicionalmente, o erro máximo absoluto observado ao longo da execução permaneceu inferior a 0,5 pulso por eixo, com valores aproximados de 0,5000 pulso para os três eixos. Esses resultados indicam que a estratégia de quantização com acumulação do erro foi capaz de preservar a coerência global do movimento, mantendo a trajetória discretizada compatível com a referência planejada, mesmo na presença de micro-blocos sem emissão de pulsos.

De forma geral, a validação numérica confirmou que o pipeline implementado no MATLAB foi capaz de gerar trajetórias cartesianas coerentes, respostas articulares válidas e comandos discretos compatíveis com os limites do sistema de execução. Assim, essa etapa forneceu suporte para a realização dos ensaios experimentais em bancada, reduzindo a probabilidade de falhas associadas a inviabilidade geométrica, violação de limites ou exigência excessiva de frequência de pulsos.

4.2 Análise do Movimento

Após a validação numérica da trajetória e da discretização do comando, foram realizados ensaios experimentais em bancada com o objetivo de avaliar o comportamento do robô durante a execução física dos movimentos planejados. Esses ensaios buscaram verificar qualitativamente a capacidade de seguimento de trajetória,

a repetibilidade funcional em ciclos de *pick-and-place* e a robustez do sistema para diferentes combinações de posições iniciais e finais.

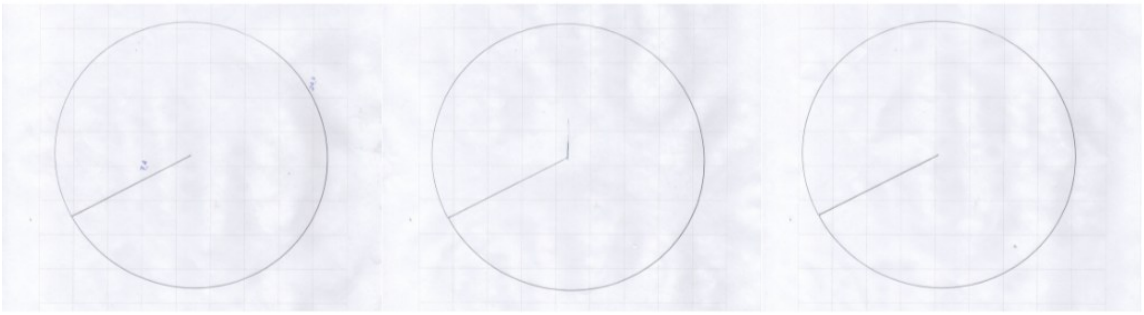
Devido às limitações do arranjo experimental empregado nos testes com marcação em papel, os resultados desta seção não são tratados como uma avaliação metrológica de alta precisão, mas sim como uma análise funcional e qualitativa da execução do movimento. Ainda assim, os ensaios realizados permitem observar a coerência entre planejamento e a trajetória executada, além de evidenciar a capacidade do sistema de realizar tarefas repetitivas de forma estável.

4.2.1 Ensaios de trajetória geométrica: círculo e quadrado

Nos primeiros ensaios experimentais, o robô foi utilizado para realizar a marcação de trajetórias geométricas sobre uma folha A4, com o objetivo de verificar qualitativamente a capacidade de seguimento de trajetória. Foram executados três ensaios com trajetória circular e dois ensaios com trajetória quadrada, utilizando a estrutura do robô em movimento real sobre a superfície de marcação.

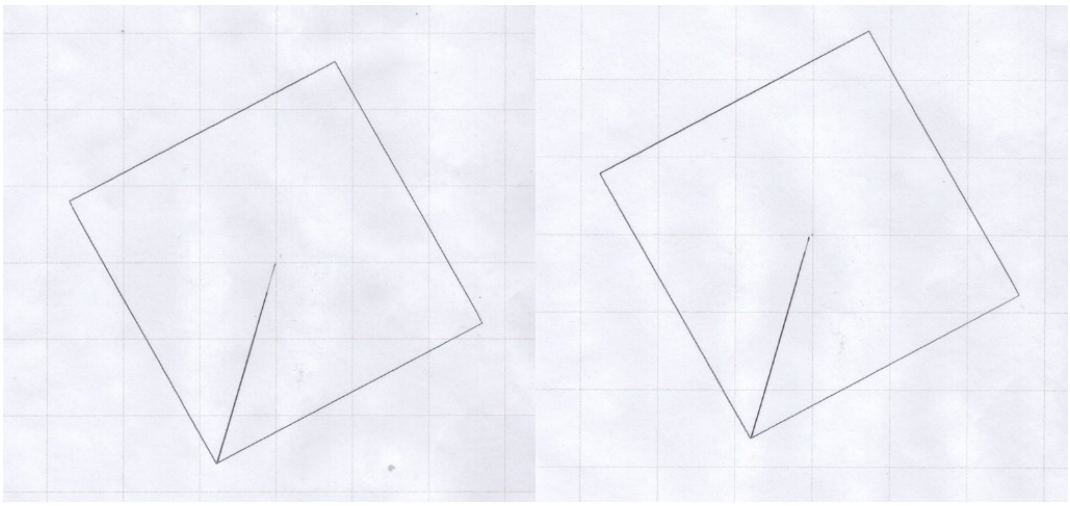
A Figura 4.5 apresenta os três círculos desenhados pelo robô, enquanto a Figura 4.6 mostra os resultados correspondentes às trajetórias em forma de quadrado. De maneira geral, observou-se que o robô foi capaz de reproduzir as formas geométricas planejadas de maneira coerente, evidenciando continuidade de movimento e compatibilidade entre a trajetória gerada no MATLAB e a trajetória efetivamente executada.

Figura 4.5: Marcação experimental da trajetória circular.



Fonte: Elaborado pelo autor.

Figura 4.6: Marcação experimental da trajetória quadrada.



Fonte: Elaborado pelo autor.

No entanto, é possível notar uma ligeira distorção tanto nos círculos quanto nos quadrados que foram desenhados, de forma que os círculos de maneira geral ficaram ligeiramente ovais e os quadrados com suas arestas mais alongadas. Essas deformações podem estar associadas a erros de montagem, imprecisões na medição das dimensões utilizadas no cálculo da cinemática inversa, folgas mecânicas e diferenças entre o modelo geométrico idealizado e a estrutura real, resultando em deslocamentos ligeiramente diferentes dos planejados.

Tabela 4.4: Resumo dos ensaios de trajetória geométrica

Ensaio	**Forma**	**Repetições**	**Tipo de Análise**	**Observações**
E1	Círculo	3	Qualitativa	Traçado contínuo, porém, com achatamento do círculo, sem quantificação do erro.
E2	Quadrado	2	Qualitativa	Traçado contínuo, porém, com distorção das arestas, sem quantificação do erro.

Fonte: Elaborado pelo autor.

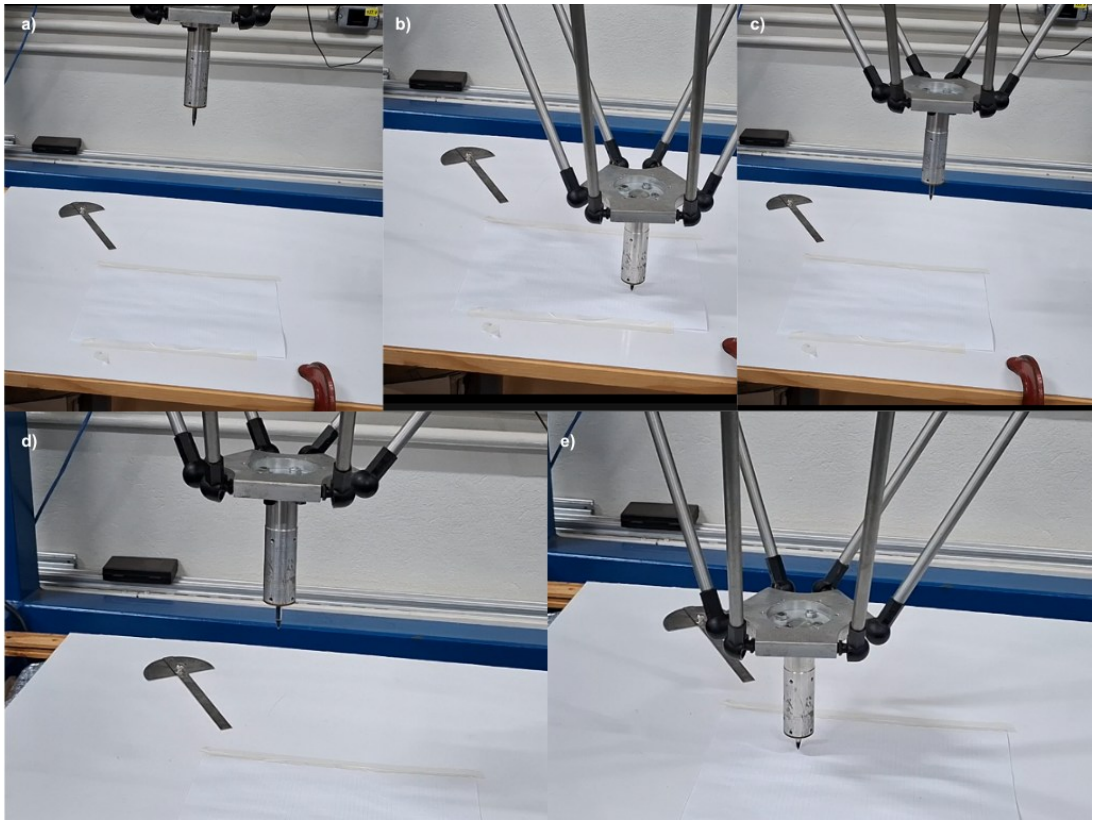
4.2.2 Ensaio de *pick-and-place* repetitivo

Em seguida, foi realizado um ensaio de *pick-and-place* repetitivo, executado 10 vezes consecutivas, com objetivo de avaliar a repetibilidade funcional do sistema em uma tarefa típica de manipulação. O movimento consistiu no deslocamento entre uma

posição inicial de coleta e uma posição final de deposição, retornando em seguida à condição de partida para início de um novo ciclo.

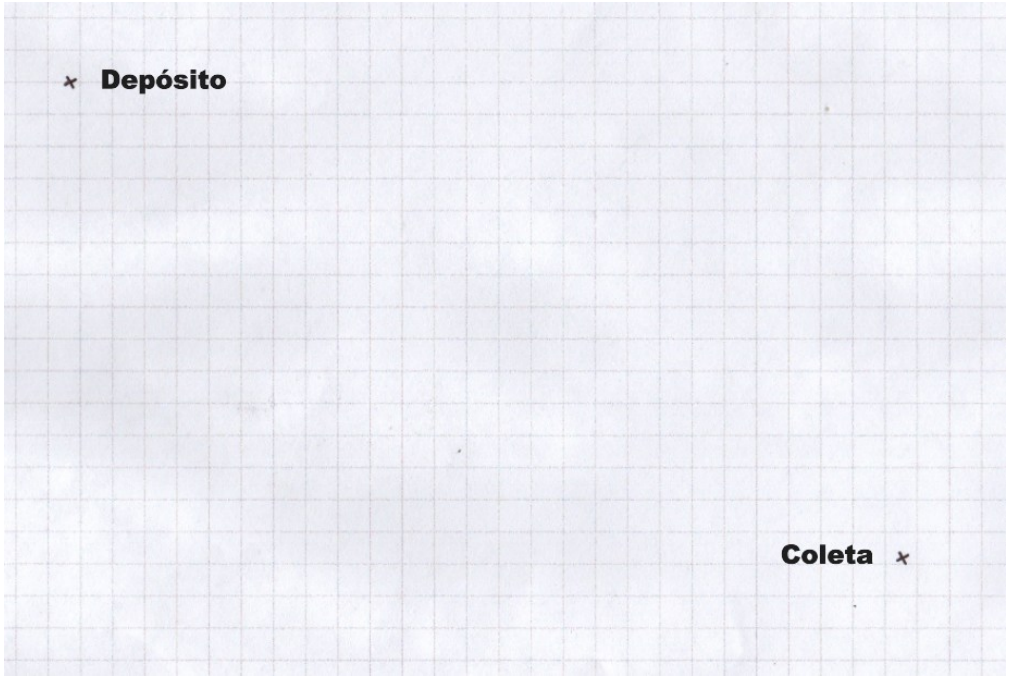
A Figura 4.7 apresenta os registros visuais do robô durante esse ensaio. Primeiro, o robô parte da posição inicial (a), até o ponto de coleta (b), realiza a marcação, se desloca até uma altura segura (c), movimenta-se até a posição de depósito na altura segura (d), abaixa até a caneta tocar a folha e realiza a marcação (e). Observou-se que o sistema foi capaz de repetir a tarefa de forma consistente ao longo das dez execuções, sem ocorrência de falhas de comunicação, travamentos ou necessidade de reinicialização do procedimento de referenciamento inicial durante a série de teste. Esse resultado indica que a integração entre planejamento, comunicação e execução embarcada foi suficientemente estável para suportar ciclos repetitivos de movimento. A Figura 4.8 apresenta as posições atingidas pelo manipulador durante o teste.

Figura 4.7: Sequência do ensaio repetitivo de pick-and-place.



Fonte: Elaborado pelo autor.

Figura 4.8: Marcações dos pontos da trajetória de pick-and-place.



Fonte: Elaborado pelo autor.

Como o arranjo experimental não foi concebido para medição de erro absoluto do ponto final, a avaliação foi conduzida em termos de taxa de sucesso funcional e de estabilidade observada na execução do percurso.

Tabela 4.5: Resultado do ensaio repetitivo de pick-and-place

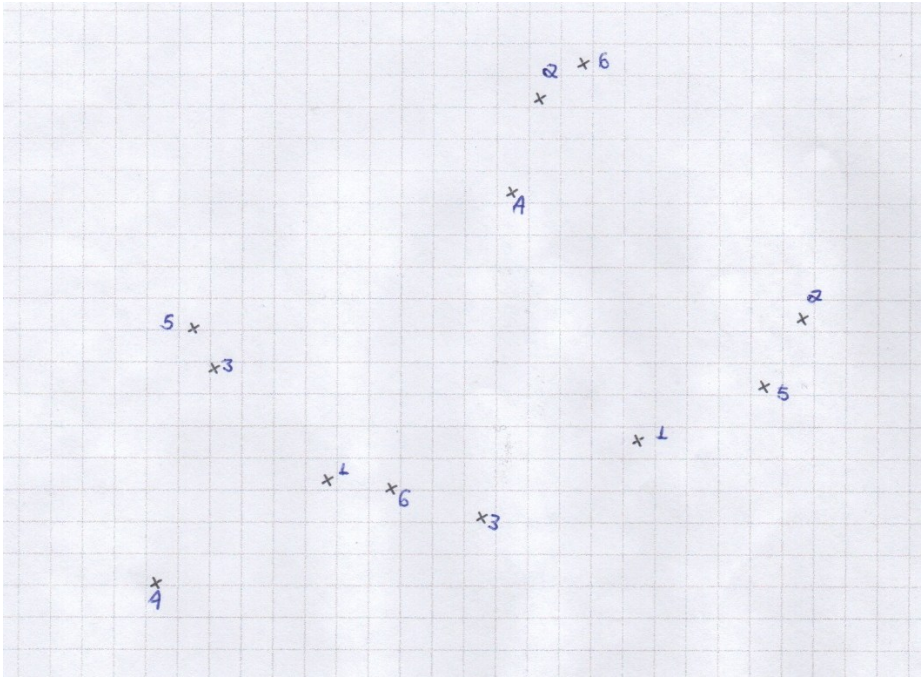
Número de repetições	Sucessos	Falhas	Taxa de sucesso
10	10	0	100 %

Fonte: Elaborado pelo autor.

4.2.3 Ensaio com múltiplos pares de posições

Com o objetivo de avaliar a versatilidade do sistema, foram realizados ensaios de *pick-and-place* utilizando seis pares distintos de posições iniciais e finais. Diferentemente do teste repetitivo anterior, nesse caso buscou-se verificar se a arquitetura proposta manteria comportamento estável ao executar deslocamentos variados, com diferentes combinações de origem e destino. A Figura 4.9 mostra seis pares de marcações feitas na folha pelo robô durante o teste.

Figura 4.9: Registros visuais de diferentes ensaios de pick-and-place.



Fonte: Elaborado pelo autor.

Os resultados do ensaio são resumidos na Tabela 4.6. De modo geral, observou-se que o sistema foi capaz de executar os diferentes pares de posições sem perda funcional do movimento, o que evidencia a robustez do pipeline de planejamento, discretização e execução embarcada para trajetórias distintas dentro da faixa operacional utilizada nos testes.

Tabela 4.6: Ensaios com diferentes pares de posições

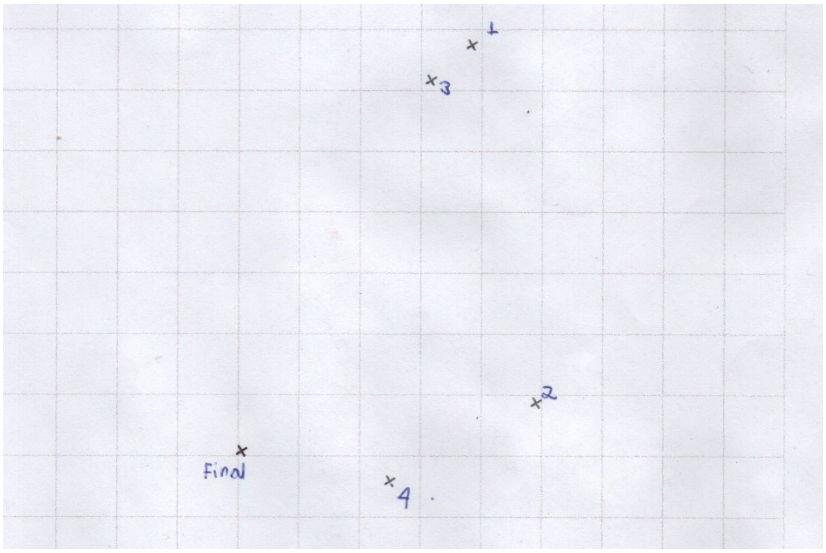
Ensaio	**Resultado**
1	Sucesso
2	Sucesso
3	Sucesso
4	Sucesso
5	Sucesso
6	Sucesso

Fonte: Elaborado pelo autor.

4.2.4 Ensaio com múltiplas posições iniciais e destino comum

Por fim, foi realizado um ensaio em que o robô partiu de quatro posições iniciais distintas, sempre convergindo para uma mesma posição final. O objetivo desse teste foi avaliar a consistência funcional do sistema ao atingir um mesmo destino a partir de diferentes condições iniciais. A Figura 4.10 mostra a marcação feita pelo robô, com as quatro posições iniciais e o destino comum.

Figura 4.10: Marcação das quatro origens e o destino comum.



Fonte: Elaborado pelo autor.

Os resultados, apresentados na Tabela 4.7, indicaram que o sistema foi capaz de atingir o destino de forma satisfatória em todos os casos avaliados. Esse resultado reforça a capacidade do algoritmo de planejar e executar trajetórias distintas para um mesmo ponto de chegada, mantendo estabilidade de execução e coerência funcional do movimento.

Tabela 4.7: Ensaios com múltiplas posições iniciais e destino comum

Ensaio	**Resultado**
1	Sucesso
2	Sucesso
3	Sucesso
4	Sucesso

Fonte: Elaborado pelo autor.

De forma geral, os ensaios em bancada indicaram que o sistema desenvolvido foi capaz de executar trajetórias geométricas e tarefas de *pick-and-place* com comportamento funcionalmente coerente com o planejamento realizado no MATLAB. Os testes qualitativos de marcação mostraram compatibilidade entre as formas planejadas e executadas, enquanto os ensaios repetitivos e com múltiplos pares de posições evidenciaram estabilidade e versatilidade operacional dentro da faixa testada. Embora a metodologia empregada nesta etapa não tenha sido voltada à medição de precisão, os resultados obtidos são suficientes para validar experimentalmente a capacidade do sistema de realizar movimentos coordenados e repetitivos de forma estável.

4.3 Desempenho do executor e comunicação

Além da validação numérica e dos ensaios em bancada, foi realizada uma análise do desempenho do executor embarcado e da comunicação entre MATLAB e ESP32-S3. Essa etapa teve como objetivo verificar se os micro-blocos gerados no planejamento eram efetivamente convertidos em sinais elétricos compatíveis com a interface de comando dos *servodrives*, bem como avaliar a robustez do protocolo de comunicação adotado.

Para essa finalidade, foram utilizados registros de osciloscópio durante a execução de trajetórias planejadas, além da observação do comportamento funcional do sistema durante os ensaios experimentais. A análise concentrou-se na geração do trem de pulsos, na consistência temporal da execução por micro-blocos e na robustez do envio de *frames* via UART sobre USB.

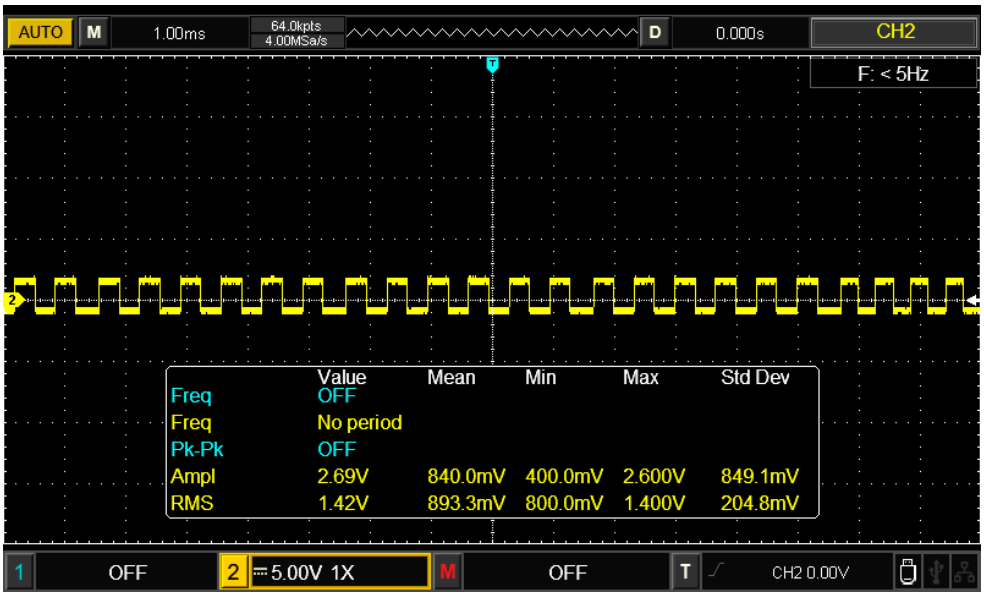
4.3.1 Análise do trem de pulsos durante a execução

Com o objetivo de validar a atuação do executor embarcado em nível de sinal, foram realizados registros em osciloscópio durante a execução de uma trajetória circular. Esses registros permitiram observar diretamente o trem de pulsos gerado pelo ESP32-S3 e encaminhado aos *servodrives*, evidenciando a conversão da trajetória discretizada em sinais elétricos de comando.

A Figura 4.11 mostra um detalhe ampliado da forma de onda observada. A análise desses registros confirma a presença de pulsos regulares durante o

movimento, compatíveis com a estratégia de execução por micro-blocos implementada no firmware.

Figura 4.11: Captura do sinal PULS no osciloscópio durante a execução.



Fonte: Elaborado pelo autor.

Do ponto de vista funcional, os sinais observados em osciloscópio foram coerentes com a lógica descrita no Capítulo 3: atualização do sentido de movimento e emissão do trem de pulsos dentro do intervalo correspondente ao micro-bloco. Esses resultados fornecem evidência experimental de que o executor embarcado foi capaz de transformar os comandos discretos recebidos em sinais compatíveis com a interface de acionamento dos *servodrives*.

4.3.2 Robustez da execução e da comunicação

A arquitetura proposta baseia-se na transmissão de micro-blocos do MATLAB para o ESP32-S3 via UART sobre USB, utilizando frames binários com sincronização por SOF, verificação de integridade por CRC-8, fila circular no executor e controle de fluxo por software XON/XOFF. Durante os ensaios experimentais realizados, essa estrutura mostrou-se funcional para o envio e execução das trajetórias planejadas.

A observação dos testes indicou que o sistema foi capaz de receber os frames, armazená-los temporariamente na fila de execução e processá-los de forma compatível com o movimento do robô, finalizando corretamente as rotinas com

sinalização de término (“DONE”). Do ponto de vista prático, não foram observadas falhas funcionais relevantes associadas à perda de sincronização da comunicação durante os ensaios que compõem este trabalho.

A robustez do sistema também foi favorecida pela adoção de mecanismos complementares de proteção, como a validação de CRC antes da aceitação do frame, o controle de fluxo por XON/XOFF para evitar saturação da fila, a execução condicionada ao referenciamento inicial e a interrupção em caso de violação de limites. Em conjunto, esses elementos contribuíram para a estabilidade do pipeline de execução durante os testes laboratoriais.

4.3.3 Discussão geral do executor embarcado

De forma geral, os resultados obtidos indicam que o ESP32-S3 foi capaz de desempenhar adequadamente o papel de executor em tempo real dentro da arquitetura proposta. Os registros em osciloscópio confirmaram a geração do trem de pulsos durante a execução das trajetórias, enquanto os ensaios funcionais em bancada mostraram que o sistema conseguiu converter o planejamento numérico em movimentos reais do robô.

Além disso, a estratégia de execução baseada em micro-blocos temporizados, associada ao uso de fila circular, validação de integridade por CRC e controle de fluxo por XON/XOFF, mostrou-se adequada para os ensaios laboratoriais realizados. Embora o presente trabalho não tenha se concentrado em uma caracterização metrológica exaustiva de *jitter* e sincronismo entre eixos, os resultados experimentais obtidos indicam que a solução adotada foi suficientemente robusta para a realização das trajetórias propostas e das tarefas de *pick-and-place* testadas em bancada.

Assim, os resultados apresentados nesta seção mostram que a arquitetura de execução embarcada implementada no ESP32-S3 foi capaz de receber, validar e executar os comandos provenientes do MATLAB de forma funcionalmente consistente com o planejamento. Em conjunto com os resultados numéricos e experimentais das seções anteriores, conclui-se que o sistema proposto atingiu o objetivo de realizar o planejamento e a execução de trajetórias em um robô Delta 3T com comando discreto por pulsos.

5 CONCLUSÃO

Este trabalho apresentou o desenvolvimento e a implementação de uma arquitetura de controle híbrida para um robô Delta de três graus de liberdade translacionais, combinando planejamento de trajetória no MATLAB e execução embarcada em tempo real no ESP32-S3. A proposta permitiu separar as etapas de maior custo computacional, como geração da trajetória, cinemática inversa e discretização em pulsos, da etapa de execução temporal, realizada localmente pelo microcontrolador por meio da geração de sinais PULS/SIGN.

A modelagem cinemática adotada permitiu converter trajetórias cartesianas em comandos articulares válidos para o protótipo, respeitando os limites operacionais definidos. A utilização de trajetórias baseadas em polinômios de quinto grau possibilitou movimentos com transições suaves de posição, velocidade e aceleração, adequados ao contexto de operações do tipo *pick-and-place*. Além disso, a discretização em micro-blocos e a estratégia de quantização com acúmulo de erro permitiram converter os incrementos articulares em pulsos inteiros, preservando a coerência global da trajetória planejada.

Nos ensaios numéricos, a trajetória circular analisada apresentou solução válida de cinemática inversa para todos os pontos avaliados, com ângulos articulares dentro da faixa operacional estabelecida. A discretização do comando apresentou frequência máxima de 3000 Hz, pulsos por micro-bloco abaixo dos limites definidos e erro final nulo em pulsos nos três eixos. Mesmo com a ocorrência de 539 frames totalmente nulos, correspondentes a 8,62 % do total, o erro máximo absoluto permaneceu limitado a aproximadamente 0,5 pulso por eixo, indicando que a estratégia de quantização empregada foi adequada para preservar a referência planejada.

Nos ensaios experimentais em bancada, o robô foi capaz de executar trajetórias geométricas, ciclos repetitivos de *pick-and-place* e deslocamentos entre diferentes pares de posições. Os resultados de marcação em papel demonstraram comportamento funcionalmente coerente com o planejamento realizado no MATLAB, embora tenham sido observadas distorções nas formas desenhadas, possivelmente associadas a imprecisões de montagem, aproximações geométricas, folgas mecânicas e limitações do método de medição empregado. Ainda assim, os testes

repetitivos apresentaram execução estável, sem falhas de comunicação ou travamentos durante os ciclos avaliados.

A análise do executor embarcado mostrou que o ESP32-S3 foi capaz de receber, validar e executar os micro-blocos enviados pelo MATLAB, gerando sinais compatíveis com a interface PULS/SIGN dos *servodrives*. O uso do periférico RMT contribuiu para a geração temporizada dos pulsos, enquanto o protocolo com SOF, CRC-8, fila circular e controle de fluxo XON/XOFF aumentou a robustez da comunicação durante os ensaios. Dessa forma, a arquitetura proposta atingiu o objetivo de realizar o planejamento e a execução de trajetórias em um robô Delta 3T com comando discreto por pulsos.

Como limitações, destaca-se que a avaliação experimental foi conduzida de forma funcional e qualitativa, sem instrumentação metrológica dedicada para medição precisa de erro absoluto, repetibilidade ou *jitter* entre eixos. Também foram observadas deformações nas trajetórias desenhadas, indicando a necessidade de calibração geométrica mais rigorosa e de melhor caracterização das fontes de erro mecânico. Como trabalhos futuros, recomenda-se realizar a calibração completa dos parâmetros geométricos do robô, implementar ensaios quantitativos com instrumentos de medição externos, avaliar o sincronismo entre eixos com maior precisão, estudar estratégias de compensação de erro para melhorar a fidelidade entre trajetória planejada e executada e a utilização do robô de maneira prática durante a disciplina de robótica.

REFERÊNCIAS

ABB. *IRB 360 FlexPicker*. [S. l.], [s. d.]. Disponível em: https://www.abb.com/global/en/areas/robotics/products/robots/delta-robots/irb-360. Acesso em: 22 jan. 2026.

AGUIAR, J. M. de. Juntas. Acionamentos. Disponível em: http://www.um.pro.br/acionamentos/index.php?c=juntas. Acesso em: 22 jan. 2026.

ANGELES, Jorge. Fundamentals of robotic mechanical systems: theory, methods, and algorithms. 4. ed. Cham: Springer, 2014.

ARAI, T.; SHERIDAN, B. Characteristics and mechanism analysis of parallel link manipulator. In: IFAC Robot Control, 1988. Proceedings [...]. [S. l.: s. n.], 1988. p. 119-124.

BIAGIOTTI, Luigi; MELCHIORRI, Claudio. Trajectory planning for automatic machines and robots. Berlin; Heidelberg: Springer, 2008.

BONEV, I. The Delta Parallel Robot: the story of success. Parallemic, 2001. Disponível em: https://www.parallemic.org/Reviews/Review002.html. Acesso em: 9 jan. 2026.

BOUTEILLE, D.; BOUTEILLE, N.; CHANTREUIL, S. et al. Les automatismes programables. 2. ed. Toulouse: Cépaduès-éditions, 1997.

CLAVEL, Raymond. Conception d'un robot parallèle rapide à quatre degrés de liberté. 1991. Tese (Doutorado) – École Polytechnique Fédérale de Lausanne, Lausanne, 1991.

CRAIG, John J. Introduction to robotics: mechanics and control. 3. ed. Upper Saddle River: Pearson Prentice Hall, 2005.

ESPRESSIF SYSTEMS. ESP32 Technical Reference Manual. [S. l.], s. d. Disponível em: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf. Acesso em: 19 fev. 2026.

ESPRESSIF SYSTEMS. Remote Control (RMT): Arduino ESP32 documentation. [S. l.], s. d. Disponível em: https://docs.espressif.com/projects/arduino-esp32/en/latest/api/rmt.html. Acesso em: 19 fev. 2026.

ESPRESSIF SYSTEMS. Remote Control (RMT): ESP-IDF Programming Guide (ESP32), versão 4.4.1. [S. l.], s. d. Disponível em:

https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/rmt.html. Acesso em: 19 fev. 2026.

FU, K. S.; GONZALEZ, R. C.; LEE, C. S. G. Robotics: control, sensing, vision, and intelligence. 2. ed. New York: McGraw-Hill, 1987.

GONÇALVES, R. S. Estudo da rigidez de cadeias cinemáticas fechadas. 2009. 147 f. Tese (Doutorado em Engenharia Mecânica) – Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Campinas, 2009.

LARA MOLINA, F. A. Ambiente de simulação de manipuladores paralelos: modelagem, Asimulação e controle de uma plataforma Stewart. 2008. 148 p. Dissertação (Mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Mecânica, Campinas, SP, 2008. Disponível em: https://hdl.handle.net/20.500.12733/1608811. Acesso em: 22 out. 2024.

MERLET, Jean-Pierre. Parallel robots. 2. ed. Dordrecht: Springer, 2006.

PARK, Frank C. Parallel robots. In: BAILLIEUL, J.; SAMAD, T. (ed.). Encyclopedia of systems and control. London: Springer-Verlag, 2015. DOI: 10.1007/978-1-4471-5058-9.

PAZOS, Fernando. Automação de sistemas & robótica. Rio de Janeiro: Axcel Books, 2002.

ROMANO, Vitor F. Robótica industrial: aplicação na indústria de manufatura e de processos. São Paulo: Edgard Blücher, 2002.

SANTOS JUNIOR, E. F.; RAMIREZ, A. R. G. Desenvolvimento de um robô delta para fins educacionais. In: CONGRESSO INTERNACIONAL DE ENGENHARIA MECÂNICA, 21., 2011, Natal. Anais [...]. Natal: ABCM, 2011. p. 1-8.

SHADOW ROBOT COMPANY. *Shadow Dexterous Hand Series*. [S. l.], 2026. Disponível em: https://shadowrobot.com/dexterous-hand-series/. Acesso em: 8 jan. 2026.

TITTON, Mathias Giordani; VALENTE, Vítor Tumelero. Modelagem, otimização do espaço de trabalho, controle e simulação de um robô delta via torque computado. 2018. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Controle e Automação) – Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul, Campus Farroupilha, Farroupilha, 2018. Disponível em: https://dspace.ifrs.edu.br/xmlui/handle/123456789/2086. Acesso em: 27 fev. 2026.

TSAI, Lung-Wen. Robot analysis: the mechanics of serial and parallel manipulators. New York: John Wiley & Sons, 1999.

WALDRON, K. J.; SCHMIEDELER, J. Kinematics. In: SICILIANO, B.; KHATIB, O. (ed.). Springer handbook of robotics. Berlin: Springer, 2008. p. 9-33.

WILLIAMS II, R. L. The Delta Parallel Robot: kinematics solutions. 2016. Disponível em: http://www.ohio.edu/people/williar4/html/pdf/DeltaKin.pdf. Acesso em: 22 out. 2024.

WILLIAMS, Ross N. A painless guide to CRC error detection algorithms. Adelaide: Rocksoft Pty Ltd., 1993. Disponível em: https://ceng2.ktu.edu.tr/~cevhers/ders_materyal/bil311_bilgisayar_mimarisi/supplementary_docs/crc_algorithms.pdf. Acesso em: 19 fev. 2026.

YASKAWA ELECTRIC CORPORATION. Catálogo SGDV (Σ-V): informações de referência por trem de pulsos (Direção + Trem de Pulso; coletor aberto). [S. l.], s. d. Disponível em: https://catalogo.yaskawa.com.br/Asset/Cat%C3%A1logo_SGDV.pdf. Acesso em: 19 fev. 2026.

YASKAWA ELECTRIC CORPORATION. Série Σ-V: trem de pulsos/tensão analógica: manual do usuário: projeto e manutenção. [S. l.], s. d. Disponível em: https://catalogo.yaskawa.com.br/Asset/SGDV---Trem-de-Pulsos-Tensao-Analogica---Manual-Portugues.pdf. Acesso em: 19 fev. 2026.

ZHENG, Laining; CHEN, Yongpeng. Kinematics and workspace analysis of the Delta Robot. IOSR Journal of Mechanical and Civil Engineering (IOSR-JMCE), [S. l.], v. 20, n. 5, ser. III, p. 39-45, set./out. 2023. DOI: 10.9790/1684-2005033945. Disponível em: https://www.iosrjournals.org/iosr-jmce/papers/vol20-issue5/Ser-3/D2005033945.pdf. Acesso em: 28 fev. 2026.

APÊNDICE A — CÓDIGO MATLAB UTILIZADO PARA PLANEJAMENTO E ENVIO DAS TRAJETÓRIAS

Neste apêndice é apresentado o código MATLAB utilizado para geração das trajetórias, cálculo da cinemática inversa, conversão dos comandos em pulsos e transmissão dos frames para o ESP32-S3.

```
%% ================ CONFIG ================
clear; clc;

% ========= MODO DE EXECUCAO =========
SIM_ONLY = true;      % true = nao envia serial, apenas simula e plota
SEND_SERIAL = ~SIM_ONLY;

% ========= SERIAL =========
port = "/dev/ttyACM0";    % <<< ajuste
baud = 115200;

% ========= TEMPORIZACAO =========
DTus = 4000;               % fixo no firmware
DT   = DTus*1e-6;

% ========= LIMITES =========
FMAX = 500000;
TH_MIN = -30; TH_MAX = 75;

% ========= CONVERSAO =========
PPR = 4000; GEAR = 25; %#ok<NASGU>
deg_per_pulse = 0.001;

% ========= PACING =========
SEND_BLOCK  = 400;          % frames por bloco
BLOCK_PAUSE = 0.05;        % s

% ========= INSPECAO DA SIMULACAO =========
SIM_PLOT_PULSE_TRAIN = true;  % plota o trem de pulsos de 1 frame
SIM_AX = 1;                  % eixo para inspecao detalhada: 1, 2 ou 3
SIM_K  = 200;                % frame para inspecao detalhada

%% ===== Parâmetros do robô =====
s_b = 0.2*6/sqrt(3);
s_p = 0.10392;
L = 0.35;
l = 0.84;
param = [s_b s_p L l]; %#ok<NASGU>

%% ===== Trajetória (exemplo) =====
% Posição HOME aproximada = [0.0054 0.0324 -0.5714] Teste Laboratório
%Circulo [0.075 0.0 -0.992] Quadrado qf = [-0.0375 -0.0375 -0.992]
% q0 = [0.0 0.0 -0.7];
% qf = [0.075 0.0 -0.989];
% T  = 10.0;
%
% t = 0:DT:T;
% N = numel(t);
%
% x = quintic_1d(q0(1), qf(1), T, t);
% y = quintic_1d(q0(2), qf(2), T, t);
```

```
% z = quintic_1d(q0(3), qf(3), T, t);
% Q = [x(:) y(:) z(:)];

% %% ===== Trajetória: Pick and Place configurável =====
cfg.q_start = [0.000  0.000  -0.7];   % posição inicial da máquina

cfg.q_pick  = [0.075    0.0  -0.992];   % ponto de coleta
cfg.q_place = [-0.075   -0.0   -0.992];   % ponto de depósito

cfg.z_safe = -0.8;    % altura segura para deslocamento
cfg.nCycles = 1;        % quantidade de ciclos

% tempos de cada etapa [s]
cfg.T_start_to_pickSafe = 2.0;
cfg.T_desc_pick         = 1.0;
cfg.T_hold_pick         = 0.5;
cfg.T_pick_to_placeSafe = 2.0;
cfg.T_desc_place        = 1.0;
cfg.T_hold_place        = 0.5;
cfg.T_return_home       = 2.0;

cfg.cross_enable_pick  = true;    % faz cruz no pick
cfg.cross_enable_place = true;    % faz cruz no place

cfg.cross_arm = 0.001;            % metade do braço da cruz [m]
cfg.cross_Tseg = 0.20;            % tempo de cada segmento da cruz [s]

cfg.returnHome = true;  % volta ao início ao final

[Q, t] = traj_pick_place_quintico(cfg, DT);
N = size(Q,1);

%% ===== Trajetória: Quadrado/Retângulo no plano XY =====
% Para quadrado, use Lx = Ly
% xc = 0.0;        % centro em X
% yc = 0.0;        % centro em Y
% zc = -0.989;     % altura fixa
% Lx = 0.075;      % largura  em X [m]
% Ly = 0.075;      % altura   em Y [m]
% T  = 10.0;       % tempo total [s]
% nVoltas = 1;     % número de voltas
%
% [Q, t] = traj_retangulo_quintico(xc, yc, zc, Lx, Ly, T, DT, nVoltas);
% N = size(Q,1);

% % ===== Trajetória: Círculo em XY + seno em Z =====
% t0 = 0;
% t1 = 10.0;
% t  = t0:DT:t1;
% N = numel(t);
%
% % ---- parâmetros da curva (ajuste) ----
% xc = 0.0;
% yc = 0.0;
% zc = -0.989;
% R  = 0.075;
% Az = 0.05;
% nVoltas = 1;
%
% tau = (t - t0) / (t1 - t0);
% s = 10*tau.^3 - 15*tau.^4 + 6*tau.^5;
```

```
% phi = 2*pi*nVoltas*s;
%
% x = xc + R*cos(phi);
% y = yc + R*sin(phi);
% %z = zc + Az*sin(2*phi);
% z = zc*ones(size(t));
%
% Q = [x(:) y(:) z(:)];

%% ===== IK =====
theta_deg = CinematicaInversa(Q, param);

if any(~isfinite(theta_deg(:)))
    error("IK inválida (NaN/Inf).");
end

%% ===== Pulsos por frame via sigma-delta =====
N = size(theta_deg,1);

theta_rel = theta_deg - TH_MIN;
TH_RANGE = TH_MAX - TH_MIN;
if any(theta_rel(:) < 0 | theta_rel(:) > TH_RANGE)
    error("Limite articular violado (theta_rel fora de [0, TH_RANGE]).");
end

% posição desejada em pulsos (float)
u = theta_rel / deg_per_pulse;   % Nx3 double

pulses  = zeros(3,N,'uint16');
dirbits = zeros(1,N,'uint8');
err     = zeros(3,1);            % erro acumulado sigma-delta (em pulsos)

dq_hist = zeros(3,N);            % incremento quantizado assinado por frame

for k = 2:N
    du = u(k,:).' - u(k-1,:).';

    y   = du + err;
    dq  = round(y);
    err = y - dq;

    dq_hist(:,k) = dq;

    db = uint8(0);
    if dq(1) >= 0, db = bitor(db, uint8(1)); end
    if dq(2) >= 0, db = bitor(db, uint8(2)); end
    if dq(3) >= 0, db = bitor(db, uint8(4)); end
    dirbits(k) = db;

    pulses(:,k) = uint16(abs(dq));
end

maxP = max(pulses,[],'all');
if maxP > 2000
    error("Pulsos por frame > 2000 (maxP=%d). Aumente DTus ou aumente
tempos.", maxP);
end

freq  = double(pulses)/(DTus*1e-6);
fpeak = max(freq,[],'all');
if fpeak > FMAX
```

```
    error("Exigiu %.1f Hz (> %.1f). Aumente DTus ou aumente tempos.", fpeak,
FMAX);
end

allZero = all(pulses==0,1);
fprintf("All-zero frames: %d (%.2f%%)\n", sum(allZero), 100*mean(allZero));
fprintf("maxP=%d | fpeak=%.1f Hz | totalABS=[%d %d %d]\n", ...
    maxP, fpeak, sum(pulses(1,:)), sum(pulses(2,:)), sum(pulses(3,:)));

%% ===== SIMULACAO DO DESLOCAMENTO EXECUTADO =====
% Convenção igual ao firmware:
% dir = 1 -> deslocamento positivo
% dir = 0 -> deslocamento negativo

d1 = bitget(dirbits,1);
d2 = bitget(dirbits,2);
d3 = bitget(dirbits,3);

signMat = [2*double(d1)-1;
           2*double(d2)-1;
           2*double(d3)-1];                % 3 x N -> +1 ou -1

dq_exec = double(pulses) .* signMat;       % incremento executado por frame
[pulsos]            % posição acumulada executada [pulsos]
u_ref   = u.';                             % referência desejada [pulsos], 3
x N
u0 = u_ref(:,1);                              % posição absoluta inicial
u_exec_rel = cumsum(dq_exec, 2);              % deslocamento executado
u_exec_abs = u0 + u_exec_rel;                 % posição absoluta executada

erro_exec = u_ref - u_exec_abs;


theta_exec_deg = TH_MIN + (u_exec_abs.' * deg_per_pulse);   % N x 3
theta_ref_deg  = theta_deg;                                 % N x 3

net = [ ...
    sum(double(pulses(1,:)).*signMat(1,:)), ...
    sum(double(pulses(2,:)).*signMat(2,:)), ...
    sum(double(pulses(3,:)).*signMat(3,:)) ...
];
absSum = [sum(pulses(1,:)), sum(pulses(2,:)), sum(pulses(3,:))];

fprintf('\n===== RESUMO DA SIMULAÇÃO =====\n');
fprintf('ABS           = [%d %d %d] pulsos\n', ...
    round(sum(abs(dq_exec(1,:)))), ...
    round(sum(abs(dq_exec(2,:)))), ...
    round(sum(abs(dq_exec(3,:)))));

fprintf('NET executado = [%d %d %d] pulsos\n', ...
    round(sum(dq_exec(1,:))), ...
    round(sum(dq_exec(2,:))), ...
    round(sum(dq_exec(3,:))));

fprintf('Final ref     = [%.3f %.3f %.3f] pulsos\n', ...
    u_ref(1,end), u_ref(2,end), u_ref(3,end));

fprintf('Final exec    = [%.3f %.3f %.3f] pulsos\n', ...
    u_exec_abs(1,end), u_exec_abs(2,end), u_exec_abs(3,end));

fprintf('Erro final    = [%.6f %.6f %.6f] pulsos\n', ...
```

```
    erro_exec(1,end), erro_exec(2,end), erro_exec(3,end));

% ---- plots de pulsos ----
figure('Name','Posicao em pulsos - referencia x executado - Eixo 1');
plot(1:N, u_ref(1,:), '--', 1:N, u_exec_abs(1,:), '-'); grid on;
xlabel('Frame'); ylabel('Pulsos'); title('Eixo 1: referência x executado');
legend('u ref','u exec');

figure('Name','Posicao em pulsos - referencia x executado - Eixo 2');
plot(1:N, u_ref(2,:), '--', 1:N, u_exec_abs(2,:), '-'); grid on;
xlabel('Frame'); ylabel('Pulsos'); title('Eixo 2: referência x executado');
legend('u ref','u exec');

figure('Name','Posicao em pulsos - referencia x executado - Eixo 3');
plot(1:N, u_ref(3,:), '--', 1:N, u_exec_abs(3,:), '-'); grid on;
xlabel('Frame'); ylabel('Pulsos'); title('Eixo 3: referência x executado');
legend('u ref','u exec');

figure('Name','Incremento executado por frame');
stairs(1:N, dq_exec(1,:), 'LineWidth', 1.0); hold on;
stairs(1:N, dq_exec(2,:), 'LineWidth', 1.0);
stairs(1:N, dq_exec(3,:), 'LineWidth', 1.0);
grid on;
xlabel('Frame'); ylabel('\Deltaq executado [pulsos/frame]');
title('Incremento executado por frame');
legend('Eixo 1','Eixo 2','Eixo 3');

% ---- plots angulares ----
figure('Name','Theta desejado x theta executado - Eixo 1');
plot(theta_ref_deg(:,1), '--'); hold on;
plot(theta_exec_deg(:,1), '-');
grid on;
xlabel('Frame'); ylabel('\theta [graus]');
title('Eixo 1: theta desejado x theta executado');
legend('Desejado','Executado');

figure('Name','Theta desejado x theta executado - Eixo 2');
plot(theta_ref_deg(:,2), '--'); hold on;
plot(theta_exec_deg(:,2), '-');
grid on;
xlabel('Frame'); ylabel('\theta [graus]');
title('Eixo 2: theta desejado x theta executado');
legend('Desejado','Executado');

figure('Name','Theta desejado x theta executado - Eixo 3');
plot(theta_ref_deg(:,3), '--'); hold on;
plot(theta_exec_deg(:,3), '-');
grid on;
xlabel('Frame'); ylabel('\theta [graus]');
title('Eixo 3: theta desejado x theta executado');
legend('Desejado','Executado');

% ---- erro acumulado em pulsos ----
figure('Name','Erro de execucao em pulsos');
plot(erro_exec(1,:)); hold on;
plot(erro_exec(2,:));
plot(erro_exec(3,:));
grid on;
xlabel('Frame'); ylabel('Erro [pulsos]');
title('Erro de execucao: u ref - u exec');
legend('Eixo 1','Eixo 2','Eixo 3');
```

```
%% ===== SIMULACAO DETALHADA DO TREM DE PULSOS =====
if SIM_PLOT_PULSE_TRAIN
    ax = max(1, min(3, SIM_AX));
    k  = max(1, min(N, SIM_K));

    p = double(pulses(ax,k));
    if ax == 1
        dir = bitget(dirbits(k),1);
    elseif ax == 2
        dir = bitget(dirbits(k),2);
    else
        dir = bitget(dirbits(k),3);
    end

    fprintf('\n===== FRAME %d / EIXO %d =====\n', k, ax);
    fprintf('dq assinado = %.0f pulsos\n', dq_hist(ax,k));
    fprintf('pulsos      = %d\n', p);
    fprintf('dir         = %d\n', dir);

    if p == 0
        disp('Esse frame não gera pulsos nesse eixo.');
    else
        half_us = round(DTus / (2*p));
        half_us = max(half_us, 2);   % igual ao firmware

        t_edges = zeros(1, 2*p+1);
        level   = zeros(1, 2*p+1);

        t_now = 0;
        level_now = 0;
        t_edges(1) = t_now;
        level(1)   = level_now;

        idx = 2;
        for n = 1:p
            t_now = t_now + half_us;
            level_now = 1;
            t_edges(idx) = t_now;
            level(idx)   = level_now;
            idx = idx + 1;

            t_now = t_now + half_us;
            level_now = 0;
            t_edges(idx) = t_now;
            level(idx)   = level_now;
            idx = idx + 1;
        end

        fprintf('half_us             = %d us\n', half_us);
        fprintf('tempo total do burst= %d us\n', t_now);

        figure('Name','Trem de pulsos detalhado');
        stairs(t_edges, level, 'LineWidth', 1.2);
        ylim([-0.2 1.2]); grid on;
        xlabel('Tempo [us]'); ylabel('STEP');
        title(sprintf('Trem de pulsos - eixo %d, frame %d, dir=%d, p=%d', ax,
k, dir, p));

        t_rise = half_us : 2*half_us : (2*p-1)*half_us;
        disp('Primeiras bordas de subida [us]:');
```

```
        disp(t_rise(1:min(end,20)));
    end
end

if SIM_ONLY
    disp('SIM_ONLY=true -> nenhuma serial foi aberta.');
    return;
end

%% ================ SERIAL ================
s = serialport(port, baud);
configureTerminator(s, "LF");
s.Timeout = 2;
flush(s);

pause(0.10);
drainSerial(s);

% Garante estado limpo e entra em BIN explicitamente
try
    writeline(s, "STOP");
    waitAnyAsciiLine(s, 0.5); % ignora eventual OK STOP
catch
end

drainSerial(s);
writeline(s, "BIN");
waitLineContainsRaw(s, "OK BIN", 3);

paused = false;
seq = uint8(0);

fprintf("Iniciando envio de %d frames...\n", N);

for k = 1:N
    paused = processControlRx(s, paused);

    while paused
        pause(0.002);
        paused = processControlRx(s, paused);
    end

    flags = uint8(0);
    if k == N
        flags = bitor(flags, uint8(1)); % END no último frame
    end

    p1 = pulses(1,k);
    p2 = pulses(2,k);
    p3 = pulses(3,k);
    db = dirbits(k);

    frame = buildFrame(seq, db, p1, p2, p3, flags);
    write(s, frame, "uint8");

    seq = uint8(mod(double(seq) + 1, 256));

    if mod(k, SEND_BLOCK) == 0
        pause(BLOCK_PAUSE);
        paused = processControlRx(s, paused);
    end
```

```
    if mod(k, 200) == 0 || k == N
        fprintf("Enviados %d/%d frames\n", k, N);
    end
end

paused = processControlRx(s, paused);

%% ===== Resumo apos envio =====
fprintf("ABS=[%d %d %d] NET=[%d %d %d]\n", absSum, round(net));
waitLineContainsRaw(s, "DONE", 60);
disp("Concluído.");

%% ================ FUNÇÕES AUX ================
function q = quintic_1d(q0, qf, T, t)
tau = t./T;
q = q0 + (qf-q0).*(10*tau.^3 - 15*tau.^4 + 6*tau.^5);
end

function frame = buildFrame(seq, dirbits, p1, p2, p3, flags)
frame = zeros(1,12,'uint8');
frame(1) = hex2dec('AA');
frame(2) = hex2dec('55');
frame(3) = uint8(seq);
frame(4) = uint8(dirbits);

frame(5)  = bitand(uint16(p1),255);
frame(6)  = bitshift(uint16(p1),-8);
frame(7)  = bitand(uint16(p2),255);
frame(8)  = bitshift(uint16(p2),-8);
frame(9)  = bitand(uint16(p3),255);
frame(10) = bitshift(uint16(p3),-8);
frame(11) = uint8(flags);
frame(12) = crc8_07(frame(1:11));
end

function crc = crc8_07(data)
crc = uint8(0);
poly = uint8(7);
for i = 1:numel(data)
    crc = bitxor(crc, uint8(data(i)));
    for b = 1:8
        if bitand(crc, uint8(128))
            crc = bitxor(bitshift(crc,1), poly);
        else
            crc = bitshift(crc,1);
        end
        crc = uint8(bitand(crc,255));
    end
end
end

function drainSerial(s)
pause(0.02);
while s.NumBytesAvailable > 0
    read(s, s.NumBytesAvailable, "uint8");
    pause(0.01);
end
end

function paused = processControlRx(s, paused)
```

```
persistent asciiBuf;
if isempty(asciiBuf)
    asciiBuf = uint8([]);
end

while s.NumBytesAvailable > 0
    b = read(s, s.NumBytesAvailable, "uint8");

    for ii = 1:numel(b)
        bi = b(ii);

        if bi == hex2dec('13')
            paused = true;
            continue;
        elseif bi == hex2dec('11')
            paused = false;
            continue;
        end

        if bi == 10
            if ~isempty(asciiBuf)
                lineBytes = asciiBuf(asciiBuf >= 32 & asciiBuf <= 126);
                line = string(char(lineBytes));
                if strlength(line) > 0
                    disp(line);
                    if contains(line, "ERR")
                        error(line);
                    end
                end
                asciiBuf = uint8([]);
            end
        elseif bi ~= 13
            asciiBuf(end+1) = bi; %#ok<AGROW>
            if numel(asciiBuf) > 256
                asciiBuf = asciiBuf(end-127:end);
            end
        end
    end
end
end

function line = waitAnyAsciiLine(s, tmax)
line = "";
buf = uint8([]);
t0 = tic;
while toc(t0) < tmax
    if s.NumBytesAvailable > 0
        b = read(s, s.NumBytesAvailable, "uint8");
        for ii = 1:numel(b)
            if b(ii) == 10
                printable = buf(buf >= 32 & buf <= 126);
                line = string(char(printable));
                return;
            elseif b(ii) ~= 13 && b(ii) ~= hex2dec('11') && b(ii) ~=
hex2dec('13')
                buf(end+1) = b(ii); %#ok<AGROW>
            end
        end
    else
        pause(0.01);
    end
```

```
end
end

function waitLineContainsRaw(s, token, tmax)
buf = uint8([]);
paused = false; %#ok<NASGU>
t0 = tic;

while toc(t0) < tmax
    if s.NumBytesAvailable > 0
        b = read(s, s.NumBytesAvailable, "uint8");

        for ii = 1:numel(b)
            bi = b(ii);
            if bi == hex2dec('13')
                paused = true; %#ok<NASGU>
                continue;
            elseif bi == hex2dec('11')
                paused = false; %#ok<NASGU>
                continue;
            end

            if bi == 10
                lineBytes = buf(buf >= 32 & buf <= 126);
                line = string(char(lineBytes));
                if strlength(line) > 0
                    disp(line);
                    if contains(line, token)
                        return;
                    end
                    if contains(line, "ERR")
                        error(line);
                    end
                end
                buf = uint8([]);
            elseif bi ~= 13
                buf(end+1) = bi; %#ok<AGROW>
                if numel(buf) > 512
                    buf = buf(end-255:end);
                end
            end
        end
    else
        pause(0.01);
    end
end

error("Timeout esperando: %s", token);
end

function [Q, t] = traj_retangulo_quintico(xc, yc, zc, Lx, Ly, T, DT, nVoltas)
% Gera trajetória retangular/quadrada no plano XY com perfil quíntico
% em cada lado. O robô desacelera nos vértices e segue para o próximo lado.
%
% xc, yc, zc  -> centro da figura
% Lx, Ly      -> dimensões em X e Y
% T           -> tempo total
% DT          -> período de amostragem
% nVoltas     -> número de voltas

    if nargin < 8
```

```
        nVoltas = 1;
    end

    % vértices (sentido anti-horário), começando no canto inferior esquerdo
    P = [ xc - Lx/2,  yc - Ly/2, zc;
          xc + Lx/2,  yc - Ly/2, zc;
          xc + Lx/2,  yc + Ly/2, zc;
          xc - Lx/2,  yc + Ly/2, zc;
          xc - Lx/2,  yc - Ly/2, zc ];

    % comprimentos dos lados
    segLensLap = [Lx, Ly, Lx, Ly];
    segLens    = repmat(segLensLap, 1, nVoltas);

    % número total de passos para bater com T e DT
    Ntotal = round(T/DT) + 1;
    Nmove  = Ntotal - 1;

    % distribui os frames proporcionalmente ao comprimento de cada lado
    idealSteps = Nmove * (segLens / sum(segLens));
    segSteps   = floor(idealSteps);
    faltam     = Nmove - sum(segSteps);

    [~, idx] = sort(idealSteps - segSteps, 'descend');
    segSteps(idx(1:faltam)) = segSteps(idx(1:faltam)) + 1;

    Q = [];
    segCount = 0;

    for lap = 1:nVoltas
        for s = 1:4
            segCount = segCount + 1;

            p0 = P(s,:);
            p1 = P(s+1,:);

            ns = segSteps(segCount);   % nº de incrementos nesse lado
            tau = linspace(0, 1, ns+1);

            sigma = 10*tau.^3 - 15*tau.^4 + 6*tau.^5;

            x = p0(1) + (p1(1)-p0(1)) * sigma;
            y = p0(2) + (p1(2)-p0(2)) * sigma;
            z = p0(3) + (p1(3)-p0(3)) * sigma;

            Qi = [x(:), y(:), z(:)];

            % evita repetir o ponto entre segmentos
            if ~isempty(Q)
                Qi = Qi(2:end,:);
            end

            Q = [Q; Qi]; %#ok<AGROW>
        end
    end

    t = (0:size(Q,1)-1)' * DT;
end

function [Q, t] = traj_pick_place_quintico(cfg, DT)
% Gera trajetória pick-and-place com segmentos suaves (quintic)
```

```
%
% Campos esperados em cfg:
% q_start, q_pick, q_place : [x y z]
% z_safe                   : altura segura
% nCycles                  : número de ciclos
% T_start_to_pickSafe
% T_desc_pick
% T_hold_pick
% T_pick_to_placeSafe
% T_desc_place
% T_hold_place
% T_return_home
% returnHome               : true/false

    q_start = cfg.q_start(:).';
    q_pick  = cfg.q_pick(:).';
    q_place = cfg.q_place(:).';

    z_safe = cfg.z_safe;
    nCycles = cfg.nCycles;

    q_pick_safe  = [q_pick(1),  q_pick(2),  z_safe];
    q_place_safe = [q_place(1), q_place(2), z_safe];

    % validação simples
    if z_safe <= q_pick(3)
        error('cfg.z_safe deve ser maior que z_pick (mais alto / menos
negativo).');
    end
    if z_safe <= q_place(3)
        error('cfg.z_safe deve ser maior que z_place (mais alto / menos
negativo).');
    end

    Q = q_start;   % começa da posição inicial
    q_cur = q_start;

    % ---- aproximação inicial até acima do pick ----
    Q = append_segment(Q, q_cur, q_pick_safe, cfg.T_start_to_pickSafe, DT);
    q_cur = q_pick_safe;

    for c = 1:nCycles
        % 1) desce no pick
        Q = append_segment(Q, q_cur, q_pick, cfg.T_desc_pick, DT);
        q_cur = q_pick;

        % 2) faz cruz no pick
        if isfield(cfg,'cross_enable_pick') && cfg.cross_enable_pick
            Q = append_cross_xy(Q, q_cur, cfg.cross_arm, cfg.cross_Tseg, DT);
        end

        % 3) espera no pick
        Q = append_hold(Q, q_cur, cfg.T_hold_pick, DT);

        % 3) sobe do pick
        Q = append_segment(Q, q_cur, q_pick_safe, cfg.T_desc_pick, DT);
        q_cur = q_pick_safe;

        % 4) vai até acima do place
        Q = append_segment(Q, q_cur, q_place_safe, cfg.T_pick_to_placeSafe,
DT);
```

```
        q_cur = q_place_safe;

        % 5) desce no place
        Q = append_segment(Q, q_cur, q_place, cfg.T_desc_place, DT);
        q_cur = q_place;

        % 6) faz cruz no place
        if isfield(cfg,'cross_enable_place') && cfg.cross_enable_place
            Q = append_cross_xy(Q, q_cur, cfg.cross_arm, cfg.cross_Tseg, DT);
        end

        % 7) espera no place
        Q = append_hold(Q, q_cur, cfg.T_hold_place, DT);

        % 7) sobe do place
        Q = append_segment(Q, q_cur, q_place_safe, cfg.T_desc_place, DT);
        q_cur = q_place_safe;

        % 8) se ainda houver próximo ciclo, volta acima do pick
        if c < nCycles
            Q       =       append_segment(Q,       q_cur,       q_pick_safe,
cfg.T_pick_to_placeSafe, DT);
            q_cur = q_pick_safe;
        end
    end

    % ---- retorno para casa ----
    if isfield(cfg, 'returnHome') && cfg.returnHome
        Q = append_segment(Q, q_cur, q_start, cfg.T_return_home, DT);
    end

    t = (0:size(Q,1)-1)' * DT;
end

function Q = append_segment(Q, q0, qf, T, DT)
% acrescenta um segmento quíntico entre q0 e qf

    if T <= 0
        return;
    end

    tt = 0:DT:T;
    if tt(end) < T
        tt = [tt T];
    end

    x = quintic_1d(q0(1), qf(1), T, tt);
    y = quintic_1d(q0(2), qf(2), T, tt);
    z = quintic_1d(q0(3), qf(3), T, tt);

    Qi = [x(:) y(:) z(:)];

    % evita repetir o primeiro ponto do novo segmento
    if ~isempty(Q)
        Qi = Qi(2:end,:);
    end

    Q = [Q; Qi];
end

function Q = append_hold(Q, q, T, DT)
```

```
% mantém a posição por um tempo T

    if T <= 0
        return;
    end

    n = max(1, round(T/DT));
    Qi = repmat(q, n, 1);

    Q = [Q; Qi];
end

function Q = append_cross_xy(Q, qc, arm, Tseg, DT)
% Desenha uma cruz no plano XY centrada em qc = [x y z]
% A trajetória retorna ao centro ao final.

    x = qc(1);
    y = qc(2);
    z = qc(3);

    pC = [x,      y,      z];
    pL = [x-arm,  y,      z];
    pR = [x+arm,  y,      z];
    pD = [x,      y-arm,  z];
    pU = [x,      y+arm,  z];

    % Horizontal
    Q = append_segment(Q, pC, pL, Tseg, DT);
    Q = append_segment(Q, pL, pR, Tseg, DT);
    Q = append_segment(Q, pR, pC, Tseg, DT);

    % Vertical
    Q = append_segment(Q, pC, pD, Tseg, DT);
    Q = append_segment(Q, pD, pU, Tseg, DT);
    Q = append_segment(Q, pU, pC, Tseg, DT);
end
```

APÊNDICE B — CÓDIGO DO FIRMWARE DO ESP32-S3 PARA EXECUÇÃO DOS COMANDOS PULS/SIGN

Neste apêndice é apresentado o firmware embarcado no ESP32-S3, responsável pela recepção dos frames, validação por CRC-8, armazenamento em fila circular, controle de fluxo XON/XOFF, referenciamento inicial e geração dos sinais PULS/SIGN por meio do periférico RMT.

```
#include <Arduino.h>
#include "driver/rmt.h"
#include <driver/gpio.h>
#include "freertos/stream_buffer.h"

#if __has_include("USB.h")
#include "USB.h"
#else
#error "USB.h nao encontrado. Use Arduino-ESP32 com suporte a USB nativa para
ESP32-S3."
#endif

#if !ARDUINO_USB_CDC_ON_BOOT
USBCDC USBSerial;
#endif

//
================================================================================
=
// ESP32-S3 N16R8
// - Debug / upload: porta USB-C ligada ao conversor USB-UART da placa
(Serial0)
// - Controle / MATLAB: porta USB-C OTG como USB CDC (USBSerial)
// - Recepcao do canal de controle em task dedicada, com StreamBuffer
//
================================================================================
=

// ================= DEBUG E CONTROLE =================
HardwareSerial &DBG = Serial;     // USB-UART bridge da placa
#define CTRL USBSerial             // USB CDC na porta OTG

static const uint32_t DBG_BAUD  = 115200;
static const uint32_t CTRL_BAUD = 115200;  // valor nominal para o host; CDC
nao depende do baud fisico

// ================= PINOS (ESP32-S3 N16R8) =================
// Evita GPIO19/20 (USB nativa), 26..32 (flash/PSRAM em muitas placas S3) e
33..37 (comuns em R8)
const int STEP_PIN[3] = {1, 5, 2};
const int DIR_PIN [3] = {6, 9, 8};
const int HOME_PIN[3] = {10, 11, 12};

// ================= RMT =================
static const int RMT_CLK_DIV = 80; // 1 tick = 1 us
static  const  rmt_channel_t  RMT_CH[3]  =  {RMT_CHANNEL_0,  RMT_CHANNEL_1,
RMT_CHANNEL_2};
static const uint16_t MAX_P_PER_SEG = 2000;
static rmt_item32_t itemsBuf[3][MAX_P_PER_SEG];
```

```
// ================ TEMPO ================
static const uint32_t DTUS = 4000;

// ================ LIMITES ================
static const int   PPR  = 1000;
static const int   GEAR = 25;
static const float DEG_PER_PULSE = 0.001;
static const float TH_MIN_DEG = -30.0f;
static const float TH_MAX_DEG =  75.0f;
static const float TH_RANGE_DEG = (TH_MAX_DEG - TH_MIN_DEG);
static const int32_t LIM_MIN_PULSES = -10000;
static  const  int32_t  LIM_MAX_PULSES  =  (int32_t)lroundf(TH_RANGE_DEG  /
DEG_PER_PULSE);

volatile int32_t pos_pulses[3] = {0, 0, 0};
volatile bool homed = false;

// ================ HOME ================
static const int HOME_ACTIVE = LOW;
static const uint16_t HOME_BACKOFF = 2000;
static const uint8_t HOME_DIR[3] = {0, 0, 0};
static const uint32_t HOME_TIMEOUT_MS = 50000;

static const uint16_t HOME_CHUNK_FAST = 20; // Corrigido: nao pode exceder
MAX_P_PER_SEG
static const uint16_t HOME_CHUNK_SLOW = 10;
static const uint32_t HOME_FREQ_FAST = 1500; // Hz
static const uint32_t HOME_FREQ_SLOW = 500;  // Hz

// ================ ESTADO ================
enum State { IDLE, BIN_RX, RUNNING, HOMING, ERROR_ST };
volatile State st = IDLE;
volatile bool stopRequested = false;
volatile bool endSeen = false;

// ================ ESTATISTICAS ================
static int32_t sumSigned[3] = {0, 0, 0};
static uint32_t sumAbs[3] = {0, 0, 0};
static uint32_t dirPosCnt[3] = {0, 0, 0};
static uint32_t dirNegCnt[3] = {0, 0, 0};
static uint32_t rxCount = 0;
static uint32_t execCount = 0;
static volatile uint32_t ctrlDroppedBytes = 0;

// ================ FILA DE SEGMENTOS ================
struct Segment {
  uint16_t p[3];
  uint8_t  d[3];
  uint8_t  flags;
};

static const int QSIZE = 2048;
Segment q[QSIZE];
volatile int q_wr = 0, q_rd = 0, q_count = 0;
portMUX_TYPE mux = portMUX_INITIALIZER_UNLOCKED;

// ================ FLOW CONTROL ================
static const int Q_HIGH = 1638;
static const int Q_LOW  = 800;
static bool flowPaused = false;
```

```
// ================= RX DO CANAL DE CONTROLE =================
static StreamBufferHandle_t ctrlRxSb = nullptr;
static TaskHandle_t ctrlRxTaskHandle = nullptr;
static const size_t CTRL_STREAM_BUF_SIZE = 16384; // 16 KB
static const size_t CTRL_RX_TASK_STACK   = 4096;

static uint8_t rxbuf[256];
static int rxlen = 0;
static char ctrlAsciiBuf[96];
static size_t ctrlAsciiLen = 0;
static char dbgAsciiBuf[96];
static size_t dbgAsciiLen = 0;

// ================= HELPERS DE LOG / CTRL =================
void ctrlPrint(const char *s) {
  CTRL.print(s);
}

void ctrlPrintln(const char *s) {
  CTRL.println(s);
}

void ctrlWriteByte(uint8_t b) {
  CTRL.write(&b, 1);
}

void ctrlPrintf(const char *fmt, ...) {
  char buf[160];
  va_list ap;
  va_start(ap, fmt);
  vsnprintf(buf, sizeof(buf), fmt, ap);
  va_end(ap);
  CTRL.print(buf);
}

void bothPrintln(const char *s) {
  DBG.println(s);
  CTRL.println(s);
}

// ================= CRC8 =================
static uint8_t crc8_07(const uint8_t *data, size_t len) {
  uint8_t crc = 0x00;
  for (size_t i = 0; i < len; i++) {
    crc ^= data[i];
    for (int b = 0; b < 8; b++) {
      crc = (crc & 0x80) ? (uint8_t)((crc << 1) ^ 0x07) : (uint8_t)(crc <<
1);
    }
  }
  return crc;
}

// ================= FLOW CONTROL =================
static inline void flowUpdate() {
  if (!flowPaused && q_count >= Q_HIGH) {
    ctrlWriteByte(0x13); // XOFF
    flowPaused = true;
  } else if (flowPaused && q_count <= Q_LOW) {
    ctrlWriteByte(0x11); // XON
```

```
    flowPaused = false;
  }
}

// ================ RMT HELPERS ================
void rmtInitAxis(int ax) {
  rmt_config_t config = RMT_DEFAULT_CONFIG_TX((gpio_num_t)STEP_PIN[ax],
RMT_CH[ax]);
  config.clk_div = RMT_CLK_DIV;
  config.tx_config.loop_en = false;
  config.tx_config.carrier_en = false;
  config.tx_config.idle_output_en = true;
  config.tx_config.idle_level = RMT_IDLE_LEVEL_LOW;

  esp_err_t e1 = rmt_config(&config);
  if (e1 != ESP_OK) {
    DBG.printf("ERR RMT_CONFIG AX=%d CH=%d PIN=%d ERR=%d\n",
               ax, (int)RMT_CH[ax], STEP_PIN[ax], (int)e1);
    return;
  }

  esp_err_t e2 = rmt_driver_install(RMT_CH[ax], 0, 0);
  if (e2 != ESP_OK) {
    DBG.printf("ERR RMT_INSTALL AX=%d CH=%d PIN=%d ERR=%d\n",
               ax, (int)RMT_CH[ax], STEP_PIN[ax], (int)e2);
    return;
  }

  DBG.printf("OK RMT AX=%d CH=%d PIN=%d\n",
             ax, (int)RMT_CH[ax], STEP_PIN[ax]);
}

bool rmtStartPulses(int ax, uint32_t half_us, uint16_t pulses) {
  if (pulses == 0) return true;
  if (pulses > MAX_P_PER_SEG) return false;
  if (half_us < 2) half_us = 2;

  rmt_item32_t item;
  item.level0 = 1; item.duration0 = half_us;
  item.level1 = 0; item.duration1 = half_us;

  for (uint16_t i = 0; i < pulses; i++) itemsBuf[ax][i] = item;
  return (rmt_write_items(RMT_CH[ax], itemsBuf[ax], pulses, false) ==
ESP_OK);
}

bool rmtWaitDone(int ax, uint32_t timeout_ms) {
  return (rmt_wait_tx_done(RMT_CH[ax], pdMS_TO_TICKS(timeout_ms)) ==
ESP_OK);
}

void testAxisPulse(int ax, uint16_t pulses, uint32_t half_us) {
  if (ax < 0 || ax > 2) return;

  digitalWrite(DIR_PIN[ax], HIGH);
  DBG.printf("TEST AX=%d PIN=%d CH=%d P=%u HU=%lu\n",
             ax, STEP_PIN[ax], (int)RMT_CH[ax],
             pulses, (unsigned long)half_us);

  bool ok1 = rmtStartPulses(ax, half_us, pulses);
  DBG.printf("rmtStartPulses -> %d\n", (int)ok1);
```

```
  if (!ok1) return;

  bool ok2 = rmtWaitDone(ax, 1000);
  DBG.printf("rmtWaitDone -> %d\n", (int)ok2);
}

// ================ QUEUE ================
bool qPush(const Segment &s) {
  portENTER_CRITICAL(&mux);
  if (q_count >= QSIZE) {
    portEXIT_CRITICAL(&mux);
    return false;
  }
  q[q_wr] = s;
  q_wr = (q_wr + 1) % QSIZE;
  q_count++;
  portEXIT_CRITICAL(&mux);
  return true;
}

bool qPop(Segment &s) {
  portENTER_CRITICAL(&mux);
  if (q_count <= 0) {
    portEXIT_CRITICAL(&mux);
    return false;
  }
  s = q[q_rd];
  q_rd = (q_rd + 1) % QSIZE;
  q_count--;
  portEXIT_CRITICAL(&mux);
  return true;
}

void qReset() {
  portENTER_CRITICAL(&mux);
  q_wr = q_rd = q_count = 0;
  portEXIT_CRITICAL(&mux);
}

// ================ HOME HELPERS ================
static inline bool homeTriggered(int ax) {
  return digitalRead(HOME_PIN[ax]) == HOME_ACTIVE;
}

static inline uint32_t half_us_from_freq(double freq) {
  double hu = (1e6 / freq) / 2.0;
  uint32_t r = (uint32_t)lround(hu);
  return (r < 2) ? 2 : r;
}

static inline bool allTrue3(const bool a[3]) {
  return a[0] && a[1] && a[2];
}

static inline uint32_t wait_ms_from_burst(uint32_t half_us, uint16_t pulses)
{
  uint32_t  burst_us  =  (uint32_t)((uint64_t)2  *  (uint64_t)half_us  *
(uint64_t)pulses);
  uint32_t burst_ms = (burst_us + 999U) / 1000U;
  return burst_ms + 20U;
}
```

```
bool homeStepSimul(const bool active[3], uint32_t half_us, uint16_t
pulsesPerAx) {
  uint32_t wait_ms = wait_ms_from_burst(half_us, pulsesPerAx);

  for (int ax = 0; ax < 3; ax++) {
    if (!active[ax]) continue;
    if (!rmtStartPulses(ax, half_us, pulsesPerAx)) {
      ctrlPrintf("ERR HOME_RMT_START AX=%d\n", ax);
      return false;
    }
  }

  for (int ax = 0; ax < 3; ax++) {
    if (!active[ax]) continue;
    if (!rmtWaitDone(ax, wait_ms)) {
      ctrlPrintf("ERR HOME_RMT_WAIT AX=%d\n", ax);
      return false;
    }
  }
  return true;
}

bool releaseIfTriggered() {
  bool any = homeTriggered(0) || homeTriggered(1) || homeTriggered(2);
  if (!any) return true;

  DBG.println("OK HOME_RELEASE");
  uint32_t half_us = half_us_from_freq(HOME_FREQ_FAST);
  uint32_t t0 = millis();

  while (homeTriggered(0) || homeTriggered(1) || homeTriggered(2)) {
    if (stopRequested) {
      ctrlPrintln("ERR HOME_STOP_REL");
      return false;
    }
    if ((millis() - t0) > HOME_TIMEOUT_MS) {
      DBG.println("ERR HOME_RELEASE_TIMEOUT");
      return false;
    }

    bool active[3];
    for (int ax = 0; ax < 3; ax++) {
      active[ax] = homeTriggered(ax);
      if (active[ax]) {
        uint8_t backDir = HOME_DIR[ax] ? 0 : 1;
        digitalWrite(DIR_PIN[ax], backDir ? HIGH : LOW);
      }
    }

    if (!homeStepSimul(active, half_us, HOME_CHUNK_FAST)) return false;
    delay(1);
  }
  return true;
}

bool doHomingSimultaneo() {
  st = HOMING;
  stopRequested = false;
  DBG.println("OK HOMING");
```

```
if (!releaseIfTriggered()) {
  st = ERROR_ST;
  return false;
}

// 1) Aproximacao rapida simultanea
for (int ax = 0; ax < 3; ax++) {
  digitalWrite(DIR_PIN[ax], HOME_DIR[ax] ? HIGH : LOW);
}

bool doneAx[3] = {false, false, false};
uint32_t half_fast = half_us_from_freq(HOME_FREQ_FAST);
uint32_t t0 = millis();

while (!allTrue3(doneAx)) {
  if (stopRequested) {
    DBG.println("ERR HOME_STOP");
    st = ERROR_ST;
    return false;
  }
  if ((millis() - t0) > HOME_TIMEOUT_MS) {
    DBG.println("ERR HOME_TIMEOUT_FAST");
    st = ERROR_ST;
    return false;
  }

  bool active[3];
  for (int ax = 0; ax < 3; ax++) {
    if (doneAx[ax]) {
      active[ax] = false;
      continue;
    }
    if (homeTriggered(ax)) {
      doneAx[ax] = true;
      active[ax] = false;
    } else {
      active[ax] = true;
    }
  }

  if (!homeStepSimul(active, half_fast, HOME_CHUNK_FAST)) {
    st = ERROR_ST;
    return false;
  }
  delay(1);
}

// 2) Backoff simultaneo
for (int ax = 0; ax < 3; ax++) {
  uint8_t backDir = HOME_DIR[ax] ? 0 : 1;
  digitalWrite(DIR_PIN[ax], backDir ? HIGH : LOW);
}

uint16_t left = HOME_BACKOFF;
while (left) {
  if (stopRequested) {
    DBG.println("ERR HOME_STOP_BACK");
    st = ERROR_ST;
    return false;
  }
  uint16_t n = min<uint16_t>(left, MAX_P_PER_SEG);
```

```
    bool active[3] = {true, true, true};
    if (!homeStepSimul(active, half_fast, n)) {
      st = ERROR_ST;
      return false;
    }
    left -= n;
    delay(1);
  }

  // 3) Aproximacao lenta simultanea
  for (int ax = 0; ax < 3; ax++) {
    digitalWrite(DIR_PIN[ax], HOME_DIR[ax] ? HIGH : LOW);
  }

  doneAx[0] = doneAx[1] = doneAx[2] = false;
  uint32_t half_slow = half_us_from_freq(HOME_FREQ_SLOW);
  t0 = millis();

  while (!allTrue3(doneAx)) {
    if (stopRequested) {
      DBG.println("ERR HOME_STOP2");
      st = ERROR_ST;
      return false;
    }
    if ((millis() - t0) > HOME_TIMEOUT_MS) {
      DBG.println("ERR HOME_TIMEOUT_SLOW");
      st = ERROR_ST;
      return false;
    }

    bool active[3];
    for (int ax = 0; ax < 3; ax++) {
      if (doneAx[ax]) {
        active[ax] = false;
        continue;
      }
      if (homeTriggered(ax)) {
        doneAx[ax] = true;
        active[ax] = false;
      } else {
        active[ax] = true;
      }
    }

    if (!homeStepSimul(active, half_slow, HOME_CHUNK_SLOW)) {
      st = ERROR_ST;
      return false;
    }
    delay(1);
  }

  for (int ax = 0; ax < 3; ax++) pos_pulses[ax] = 0;
  homed = true;

  DBG.println("OK HOME_DONE");
  st = IDLE;
  return true;
}

// ================ EXECUCAO DE 1 SEGMENTO ================
bool runOneSegment(const Segment &s) {
```

```
  if (!homed) {
    DBG.println("ERR NOT_HOMED");
    return false;
  }

  for (int ax = 0; ax < 3; ax++) {
    int32_t delta = s.p[ax];
    int32_t next  = pos_pulses[ax] + (s.d[ax] ? +delta : -delta);
    if (next < LIM_MIN_PULSES || next > LIM_MAX_PULSES) {
      DBG.printf("ERR LIM AX=%d NEXT=%ld\n", ax, (long)next);
      return false;
    }
  }

  for (int ax = 0; ax < 3; ax++) digitalWrite(DIR_PIN[ax], s.d[ax] ? HIGH :
LOW);

  uint32_t half_us[3] = {0, 0, 0};
  for (int ax = 0; ax < 3; ax++) {
    if (s.p[ax] == 0) continue;
    double hu = (double)DTUS / (2.0 * (double)s.p[ax]);
    half_us[ax] = (uint32_t)lround(hu);
    if (half_us[ax] < 2) half_us[ax] = 2;
  }

  uint32_t t0 = micros();
  for (int ax = 0; ax < 3; ax++) {
    if (stopRequested) return false;
    if (s.p[ax] == 0) continue;
    if (!rmtStartPulses(ax, half_us[ax], s.p[ax])) {
      ctrlPrintf("ERR RMT_START AX=%d\n", ax);
      return false;
    }
  }

  for (int ax = 0; ax < 3; ax++) {
    if (s.p[ax] == 0) continue;
    if (!rmtWaitDone(ax, 50)) {
      ctrlPrintf("ERR RMT_WAIT AX=%d\n", ax);
      return false;
    }
  }

  uint32_t elapsed = (uint32_t)(micros() - t0);
  if (elapsed < DTUS) delayMicroseconds(DTUS - elapsed);

  for (int ax = 0; ax < 3; ax++) {
    int32_t delta = s.p[ax];
    pos_pulses[ax] += (s.d[ax] ? +delta : -delta);
  }

  for (int ax = 0; ax < 3; ax++) {
    int32_t delta = s.p[ax];
    sumAbs[ax] += (uint32_t)delta;
    if (s.d[ax]) {
      sumSigned[ax] += delta;
      dirPosCnt[ax]++;
    } else {
      sumSigned[ax] -= delta;
      dirNegCnt[ax]++;
    }
```

```
  }
  return true;
}

// ================ PARSER BINARIO ================
bool tryParseOneFrame() {
  while (rxlen >= 12) {
    int i = 0;
    while (i + 1 < rxlen) {
      if (rxbuf[i] == 0xAA && rxbuf[i + 1] == 0x55) break;
      i++;
    }
    if (i > 0) {
      memmove(rxbuf, rxbuf + i, rxlen - i);
      rxlen -= i;
      continue;
    }
    if (rxlen < 12) return false;

    uint8_t crc = crc8_07(rxbuf, 11);
    if (crc != rxbuf[11]) {
      memmove(rxbuf, rxbuf + 1, rxlen - 1);
      rxlen -= 1;
      continue;
    }

    uint8_t dirbits = rxbuf[3];
    uint16_t p1 = (uint16_t)(rxbuf[4] | (rxbuf[5] << 8));
    uint16_t p2 = (uint16_t)(rxbuf[6] | (rxbuf[7] << 8));
    uint16_t p3 = (uint16_t)(rxbuf[8] | (rxbuf[9] << 8));
    uint8_t flags = rxbuf[10];

    Segment s{};
    s.p[0] = p1; s.p[1] = p2; s.p[2] = p3;
    s.d[0] = (dirbits & 0x01) ? 1 : 0;
    s.d[1] = (dirbits & 0x02) ? 1 : 0;
    s.d[2] = (dirbits & 0x04) ? 1 : 0;
    s.flags = flags;

    if (p1 > MAX_P_PER_SEG || p2 > MAX_P_PER_SEG || p3 > MAX_P_PER_SEG) {
      DBG.println("ERR BIN_P_TOO_BIG");
    } else {
      if (!qPush(s)) {
        DBG.println("ERR Q_FULL");
      } else {
        if (flags & 0x01) endSeen = true;
        rxCount++;
        if ((rxCount % 500U) == 0U) {
          DBG.printf("RX=%lu Q=%d\n", (unsigned long)rxCount, (int)q_count);
        }
      }
    }

    memmove(rxbuf, rxbuf + 12, rxlen - 12);
    rxlen -= 12;
    flowUpdate();
    return true;
  }
  return false;
}
```

```
// ================= TASK DE RX DO CONTROLE =================
void ctrlRxTask(void *arg) {
  (void)arg;
  uint8_t temp[128];

  for (;;) {
    int avail = CTRL.available();
    if (avail <= 0) {
      vTaskDelay(pdMS_TO_TICKS(1));
      continue;
    }

    size_t want = (size_t)min<int>(avail, (int)sizeof(temp));
    size_t n = CTRL.read(temp, want);
    if (n == 0) {
      vTaskDelay(pdMS_TO_TICKS(1));
      continue;
    }

    size_t sent = xStreamBufferSend(ctrlRxSb, temp, n, 0);
    if (sent < n) {
      ctrlDroppedBytes += (uint32_t)(n - sent);
    }
  }
}

// ================= COMANDOS ASCII =================
void resetStatsForBin() {
  for (int ax = 0; ax < 3; ax++) {
    sumSigned[ax] = 0;
    sumAbs[ax] = 0;
    dirPosCnt[ax] = 0;
    dirNegCnt[ax] = 0;
  }
  rxCount = 0;
  execCount = 0;
}

void enterBinMode() {
  resetStatsForBin();
  qReset();
  rxlen = 0;
  endSeen = false;
  stopRequested = false;
  flowPaused = false;
  st = BIN_RX;
  bothPrintln("OK BIN");
}

void doStop() {
  stopRequested = true;
  st = IDLE;
  qReset();
  endSeen = false;
  bothPrintln("OK STOP");
}

void printStatus() {
  DBG.printf("OK ST=%d Q=%d HOMED=%d END=%d POS=[%ld,%ld,%ld] DROP=%lu\n",
             (int)st, (int)q_count, (int)homed, (int)endSeen,
             (long)pos_pulses[0], (long)pos_pulses[1], (long)pos_pulses[2],
```

```
              (unsigned long)ctrlDroppedBytes);
}

void handleAsciiCommand(const char *line, bool allowBin) {
  if (!line || !line[0]) return;

  if (strncmp(line, "TEST1", 5) == 0) { testAxisPulse(0, 1000, 100); return;
}
  if (strncmp(line, "TEST2", 5) == 0) { testAxisPulse(1, 1000, 100); return;
}
  if (strncmp(line, "TEST3", 5) == 0) { testAxisPulse(2, 1000, 100); return;
}

  if (strcmp(line, "STOP") == 0) {
    doStop();
    return;
  }
  if (strcmp(line, "HOME") == 0) {
    doHomingSimultaneo();
    return;
  }
  if (strcmp(line, "STATUS") == 0) {
    printStatus();
    return;
  }
  if (strcmp(line, "HOME_RAW") == 0) {
    DBG.printf("HOME_RAW: %d %d %d\n",
               digitalRead(HOME_PIN[0]),
               digitalRead(HOME_PIN[1]),
               digitalRead(HOME_PIN[2]));
    return;
  }
  if (allowBin && strcmp(line, "BIN") == 0) {
    enterBinMode();
    return;
  }

  DBG.print("CMD? ");
  DBG.println(line);
}

void serviceDebugConsole() {
  while (DBG.available() > 0) {
    char c = (char)DBG.read();
    if (c == '\r') continue;

    if (c == '\n') {
      dbgAsciiBuf[dbgAsciiLen] = '\0';
      if (dbgAsciiLen > 0) handleAsciiCommand(dbgAsciiBuf, true);
      dbgAsciiLen = 0;
      continue;
    }

    if (dbgAsciiLen < sizeof(dbgAsciiBuf) - 1) {
      dbgAsciiBuf[dbgAsciiLen++] = c;
    } else {
      dbgAsciiLen = 0;
    }
  }
}
```

```
void serviceCtrlAsciiAndBinary() {
  uint8_t temp[128];
  size_t n = xStreamBufferReceive(ctrlRxSb, temp, sizeof(temp), 0);
  if (n == 0) return;

  for (size_t i = 0; i < n; i++) {
    uint8_t b = temp[i];

    if (st == BIN_RX || st == RUNNING) {
      if (rxlen < (int)sizeof(rxbuf)) {
        rxbuf[rxlen++] = b;
      } else {
        ctrlDroppedBytes++;
      }
      continue;
    }

    if (b == '\r') continue;
    if (b == '\n') {
      ctrlAsciiBuf[ctrlAsciiLen] = '\0';
      if (ctrlAsciiLen > 0) handleAsciiCommand(ctrlAsciiBuf, true);
      ctrlAsciiLen = 0;
      continue;
    }

    if (ctrlAsciiLen < sizeof(ctrlAsciiBuf) - 1) {
      ctrlAsciiBuf[ctrlAsciiLen++] = (char)b;
    } else {
      ctrlAsciiLen = 0;
    }
  }

  if (st == BIN_RX || st == RUNNING) {
    while (tryParseOneFrame()) {
      // continua enquanto houver frame completo
    }
  }
}

// ================ SETUP / LOOP ================
void setup() {
  DBG.begin(DBG_BAUD);
  delay(250);

  // Inicializa USB CDC na porta OTG
  CTRL.setRxBufferSize(8192);
  CTRL.setTxTimeoutMs(10);
  CTRL.enableReboot(false); // evita reset por DTR/RTS do host
  CTRL.begin(CTRL_BAUD);
  USB.begin();

  DBG.println("FIRMWARE ESP32-S3 - USB CDC CTRL + Serial0 DBG");

  for (int ax = 0; ax < 3; ax++) {
    pinMode(HOME_PIN[ax], INPUT_PULLUP);
  }
  DBG.printf("HOME_RAW_BOOT: %d %d %d\n",
             digitalRead(HOME_PIN[0]),
             digitalRead(HOME_PIN[1]),
             digitalRead(HOME_PIN[2]));
```

```
  for (int ax = 0; ax < 3; ax++) {
    pinMode(STEP_PIN[ax], OUTPUT);
    digitalWrite(STEP_PIN[ax], LOW);

    pinMode(DIR_PIN[ax], OUTPUT);
    digitalWrite(DIR_PIN[ax], LOW);

    rmtInitAxis(ax);
  }

  ctrlRxSb = xStreamBufferCreate(CTRL_STREAM_BUF_SIZE, 1);
  if (ctrlRxSb == nullptr) {
    DBG.println("ERR CTRL_STREAM_ALLOC");
    st = ERROR_ST;
    return;
  }

  BaseType_t ok = xTaskCreatePinnedToCore(
      ctrlRxTask,
      "ctrlRxTask",
      CTRL_RX_TASK_STACK,
      nullptr,
      2,
      &ctrlRxTaskHandle,
      0);

  if (ok != pdPASS) {
    DBG.println("ERR CTRL_TASK_CREATE");
    st = ERROR_ST;
    return;
  }

  qReset();
  st = IDLE;

  ctrlPrintln("OK BOOT");
  DBG.println("OK BOOT");
}

void loop() {
  serviceDebugConsole();
  serviceCtrlAsciiAndBinary();

  if (stopRequested) {
    st = IDLE;
    qReset();
    ctrlPrintln("DONE (STOP)");
    DBG.println("DONE (STOP)");
    stopRequested = false;
    return;
  }

  if (st == BIN_RX || st == RUNNING) {
    if (q_count > 0) {
      Segment s;
      if (qPop(s)) {
        execCount++;
        if ((execCount % 500U) == 0U) {
          DBG.printf("EXEC=%lu p=[%u %u %u]\n",
                     (unsigned long)execCount,
                     s.p[0], s.p[1], s.p[2]);
```

```
        }
        st = RUNNING;
        if (!runOneSegment(s)) {
          st = ERROR_ST;
          DBG.println("ERR RUN");
          return;
        }
        flowUpdate();
      }
    } else {
      if (endSeen) {
        st = IDLE;
        DBG.printf("SUM         ABS=[%lu,%lu,%lu]         SIGN=[%ld,%ld,%ld]
DIR+=[%lu,%lu,%lu] DIR-=[%lu,%lu,%lu]\n",
                   (unsigned   long)sumAbs[0],   (unsigned   long)sumAbs[1],
(unsigned long)sumAbs[2],
                   (long)sumSigned[0],                  (long)sumSigned[1],
(long)sumSigned[2],
                   (unsigned long)dirPosCnt[0], (unsigned long)dirPosCnt[1],
(unsigned long)dirPosCnt[2],
                   (unsigned long)dirNegCnt[0], (unsigned long)dirNegCnt[1],
(unsigned long)dirNegCnt[2]);
        ctrlPrintln("DONE");
        DBG.println("DONE");
        return;
      }
      delay(1);
    }
  }
}
```