Centro de Engenharia, Modelagem e Ciências Sociais Aplicadas

Programa de Graduação em Engenharia de Instrumentação, Automação e Robótica

CONTROLE DE ROBÔ SEGUIDOR COM ACIONAMENTO DIFERENCIAL

GUSTAVO SILVA DE PAULA

Santo André - SP

2021

GUSTAVO SILVA DE PAULA

CONTROLE DE ROBÔ SEGUIDOR COM ACIONAMENTO DIFERENCIAL

Monografia apresentada ao Curso de Engenharia de Instrumentação, Automação e Robótica da Universidade Federal do ABC, como parte dos requisitos para obtenção do Grau de Engenheiro de Controle e Automação.

Orientador: Prof. Drº Mario Alexandre Gazziro

Santo André - SP

2021

ALUNO(s):

Gustavo Silva de Paula – 21085415

ORIENTADOR: Mario Alexandre Gazziro

TÍTULO DO TRABALHO: Controle de robô seguidor com acionamento diferencial

AVALIADOR: Luis Alberto Martinez Riascos

Parte 3: ATRIBUIÇÃO DO CONCEITO (AVALIADOR E ORIENTADOR)

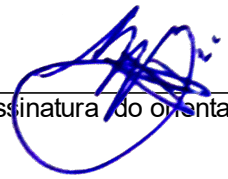
Conceito Sugerido pelo **AVALIADOR**: A-

Santo André, 15/03/2021
(Data)

Assinatura do avaliador

Conceito Sugerido pelo **ORIENTADOR**: A+

Santo André, 17/03/2021
(Data)



Assinatura do orientador

Conceito final (em caso de discordância, preenchido pelo **COORDENADOR DE TG**):

AGRADECIMENTOS

Agradeço aos meus amigos dessa universidade, em especial os membros da equipe de robótica *Project Neon* que me acompanharam no desenvolvimento de novas habilidades que jamais seriam desenvolvidas em outros lugares. Agradeço ao meu parceiro Marcelo Farias que juntos desenvolvemos trabalhos incríveis e minha namorada Girlene Cavalcanti por motivar a continuidade dos trabalhos. Meu obrigado ao orientador Mario Gazziro por todos os conselhos e ajuda nesse período.

"You can't connect the dots looking forward; you can only connect them looking backwards. So, you have to trust that the dots will somehow connect in your future."

Steve Jobs.

RESUMO

O robô diferencial é composto por dois motores posicionados simetricamente em relação ao eixo, seu deslocamento é dado de forma linear pela velocidade das rodas e angular de acordo com a diferença de rotação entre os motores. Existe um problema de estabilização no robô que consiste em atingir uma dada configuração final desejada que é definida pela localização no espaço cartesiano e uma determinada orientação, para isto é necessário realizar a implementação de um sistema controle para otimizar o problema de estabilização. Este trabalho realiza a implementação da modelagem cinemática e dinâmica de um robô diferencial baseado na literatura já existente, projeta e desenvolve um robô seguidor de linha com acionamento diferencial para competições de robótica, implementa um sistema de controle PD e otimiza o rendimento do robô em um percurso por meio do mapeamento de pista. Para atingir os objetivos propostos foram realizadas pesquisas sobre modelagem cinemática e entendido os diferentes parâmetros que afetam o desempenho do robô. Em seguida foi desenvolvido o projeto do robô por modelagem mecânica 3D, modelagem elétrica e modelagem dinâmica do sistema de controle PD. Posteriormente foi implementado o protótipo e otimizado com um *firmware* capaz de realizar o mapeamento de pista. Por fim, o robô foi testado em competições de robótica sendo capaz de conquistar 2 troféus, validando toda implementação.

Palavras-chave: Robôs; controle PID; controle PD, acionamento diferencial; modelagem dinâmica; robôs móveis; competição de robótica.

ABSTRACT

The differential robot consists of two motors positioned symmetrically in relation to the shaft, its displacement is given linearly by the speed of the wheels and angular according to the difference in rotation between the motors. There is a stabilization problem in the robot that consists of achieving a given desired final configuration that is defined by the location in Cartesian space and a certain orientation, it would be necessary to implement a control system to optimize the stabilization problem. This work performs the implementation of the kinematic and dynamic modeling of a differential robot based on existing literature, designs, and develops a line follower robot with differential drive for robotics competitions, implements a PD control system and optimizes the robot's performance in a path through track mapping. To achieve the proposed objectives, research was carried out on kinematic modeling and understood the different parameters that affect the performance of the robot. In the next steps, the robot was designed by 3D mechanical modeling, electrical modeling, and dynamic modeling of the PD control system. Subsequently, the prototype was implemented and optimized *with a firmware* capable of performing the track mapping. Finally, the robot was tested in robotics competitions wining 2 trophies, validating all implementation.

Keywords: Robots; PID control; PD control, differential drive; dynamic modeling; mobile robots; robotics competition.

LISTA DE SIGLAS

PID	-	Proporcional Integral Derivativo
PD	-	Proporcional Derivativo
IDE	-	*Integrated development environment*
MCU	-	Microcontrolador
BT	-	*Bluetooth*
PCB	-	*Printed Circuit Board*

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

Robôs são utilizados para exploração de locais em que a segurança para os seres humanos está em risco ou na realização de tarefas repetitivas, no geral são desenvolvidos para se comportarem de forma totalmente autônoma.

Sistemas robóticos podem ser classificados como robôs manipuladores e robôs móveis, ambas as classes estão em crescente avanço tecnológico. Estes sistemas cada vez mais passam uma maior confiabilidade, autonomia e segurança, refletindo diretamente em uma aceitação maior do mercado, seja este industrial, doméstico ou militar. A classe de robôs móveis é constituída por robôs que têm a capacidade de locomoção no espaço tridimensional.

As aplicações da robótica móvel são muito variadas e normalmente relacionadas a algumas tarefas que são perigosas para os humanos, como por exemplo, realizar a exploração em ambientes hostis.

Existem diversas competições de robótica que buscam ampliar os estudos da robótica de forma prática. Estas competições são amplas que envolvem categorias como combate de robôs, futebol de robôs, sumô, drone e seguidor de linha (CAMPOS).

Com o crescimento do número de competições no Brasil, a participação dos estudantes das áreas de tecnologia e engenharia, a robótica está se ampliando a cada dia, assim como já é também uma realidade em ambientes domésticos. Entre os principais campeonatos realizados no Brasil estão a *RoboCup*, *Winter Challenge*, *Iron Cup* e *Smiles*.

A iniciativa também fortalece a capacidade de inovação, criatividade e raciocínio lógico, inspirando jovens a seguir carreira no ramo da engenharia, matemática e tecnologia. Por meio de uma experiência criativa, os competidores são desafiados a investigar problemas e buscar soluções inovadoras para situações da vida real.

O foco deste trabalho está no desenvolvimento de robôs seguidores de linha, uma categoria de robôs autônomos que se enquadram na classe de robôs móveis. Dada a combinação de motores, sensores e *software*, o principal objetivo desta categoria é percorrer um percurso circular no menor tempo possível. O circuito é definido com base em regras pré-estabelecidas, que padronizam trechos retos e curvas. O robô deve percorrer o circuito de forma que o corpo

dele permaneça sempre sobre a linha, sendo que o robô inicia sua volta sempre dentro de um trecho delimitado da pista e deverá parar automaticamente dentro desse mesmo espaço, conhecido como "área de partida- e chegada" (Robocore, 2016).

Um robô diferencial é um robô onde a sua locomoção consiste em duas rodas montadas num eixo comum, controladas por motores independentes, um para cada roda e rodas livres ou apoios para assegurar a estabilidade do robô. A movimentação é dada pela contribuição individual da velocidade de cada roda, ou seja, ambas as rodas devem girar à mesma velocidade para que o robô avança em uma só direção.

Para o caso específico de robô diferenciais, deve-se considerar que as rodas destes podem apresentar restrições ao seu movimento. As restrições que fazem com que a dimensão do espaço das velocidades seja menor que a dimensão do espaço de configuração do robô são chamadas de restrições não-holonômicas (Figueiredo, 2004). Automóveis são um exemplo nativo disso, pois seus movimentos são limitados a serem colineares com a direção das rodas, não permitindo o movimento lateral.

A robótica móvel é a área que estuda a locomoção de robôs, constituído por uma arquitetura de *hardware* e *software* responsável por garantir a locomoção de forma correta e desejada. O *software* fica responsável por garantir a geração da trajetória e dentro dele o sistema de controle fica responsável por executar esta trajetória. Normalmente os robôs seguidores de linha são robôs diferenciais e o controle é baseado na realimentação com o estado do robô, isto é, velocidade e posição.

Tipicamente para realizar este controle é usado estratégia de controle linear, como controlador PID (Proporcional Integral Derivativo) que é uma técnica de controle que atua sobre uma variável a ser manipulada a partir da ação de três controladores (BORGES, et al., 2003).

Os três controladores são gerados a partir de constantes independentes chamadas de ganhos que comumente são obtidas empiricamente através da realização de testes.

1.1. Fundamentação teórica

1.1.1. Modelagem cinemática

O sistema modelado é um robô diferencial, acionado por dois motores de corrente contínua conforme a figura 1 abaixo (VIEIRA, 2005):

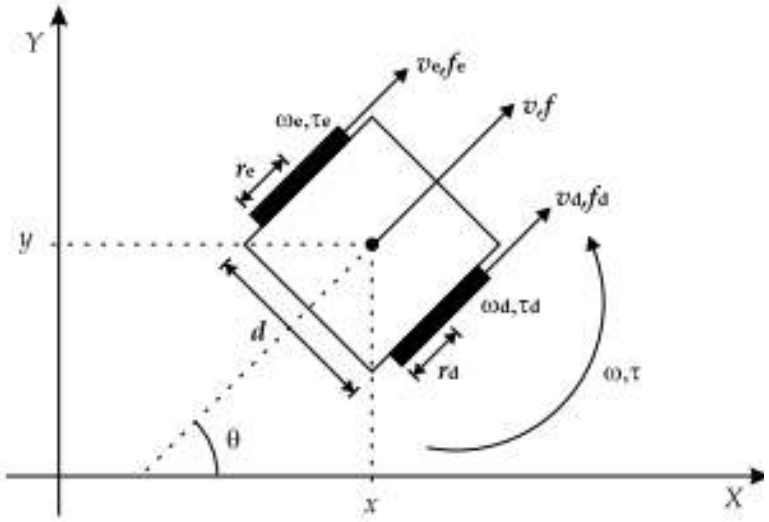


Figura 1 - Modelo cinemático do robô diferencial. (Vieira, 2005)

A nomenclatura adotada é a seguinte:

d é o comprimento do eixo;

$r_{d,e}$ são os raios das rodas;

$\omega_{d,e}$ são as velocidades angulares;

$v_{d,e}$ são as velocidades lineares;

v é a velocidade linear do robô;

ω é a velocidade angular do robô;

A partir destes parâmetros pode-se obter a relação entre as velocidades lineares e angulares do robô:

$$v_d = \omega_d r_d \qquad\qquad v_e = \omega_e r_e$$

Para o robô movendo-se de forma linear, pode-se obter:

$$v_d = v + \frac{\omega \ d}{2} \qquad\qquad v_e = v - \frac{\omega \ d}{2}$$

Das equações acima, chega-se em:

$$v \ = \omega_d \frac{r_d}{2} \ + \omega_e \frac{r_e}{2} \qquad\qquad \omega = \omega_d \frac{r_d}{d} \ - \omega_e \frac{r_e}{d}$$

De forma visual, a figura 2 representa os cálculos realizados:

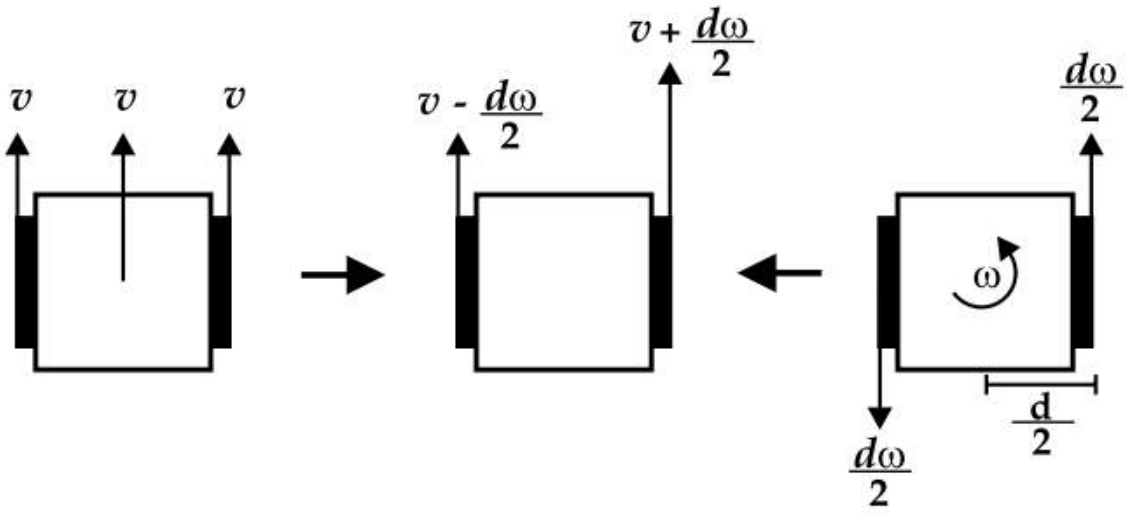


Figura 2 - Velocidades no robô diferencial. (Vieira, 2005)

1.1.2. Modelagem Estática

Conforme já descrito, o robô modelado é composto por dois motores de corrente contínua que são responsáveis pela locomoção.

O projeto do controlador pode ser baseado na representação em segunda ordem dos sistemas de propulsão, apresentada na equação:

$$G(s) = \frac{\Omega(s)}{U(s)} = \frac{b_1 s + b_0}{s^2 + a_1 s + a_0}$$

Que relaciona, no domínio de Laplace, a velocidade de rotação $\Omega(s)$ com a tensão de entrada do motor $U(s)$.

Os motores estão diretamente conectados aos motores do robô, desta forma pode-se obter a relação dinâmica a partir do modelo abaixo:

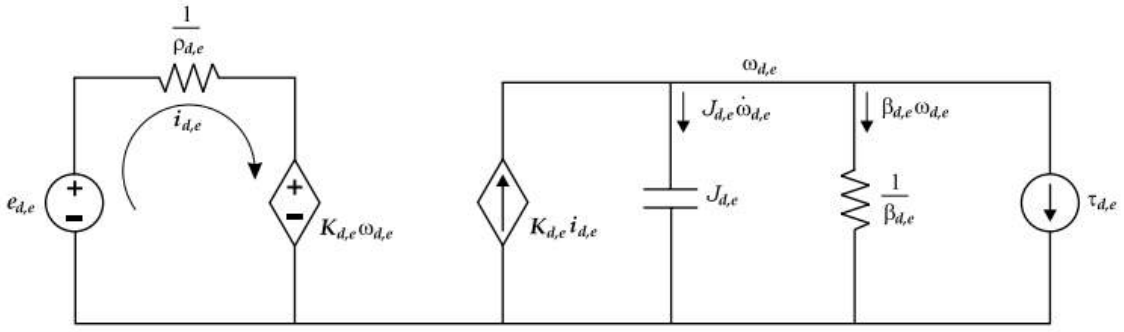


Figura 3 - Esquema elétrico dos motores do robô. (Vieira, 2005)

Desprezando a indutância de armadura de um motor de corrente contínua, pode-se obter em malha aberta, um sistema de primeira ordem dado pela seguinte função de transferência:

$$G'(s) = \frac{\Omega(s)}{U(s)} = \frac{b_0}{a_1 s + a_0}$$

O controlador PID combina as três ações de controle do erro entre a referência de velocidade de rotação $\omega^*(s)$ e a saída $\omega\ (s)$ aplicadas na entrada de excitação do sistema. Dessa forma, a lei de controle, no domínio do tempo, é dada por:

$$u(t) = u(0) + K_p e(t) + K_i \int_0^t e(\iota) d\iota + K_d \frac{d\, e(t)}{dt}$$

Com $e(t) = \omega^*(t) - \omega(t)$ e em que K_pé o ganho proporcional, K_io ganho integral e K_do ganho derivativo. O diagrama de blocos do controlador PID é exibido na figura 4.

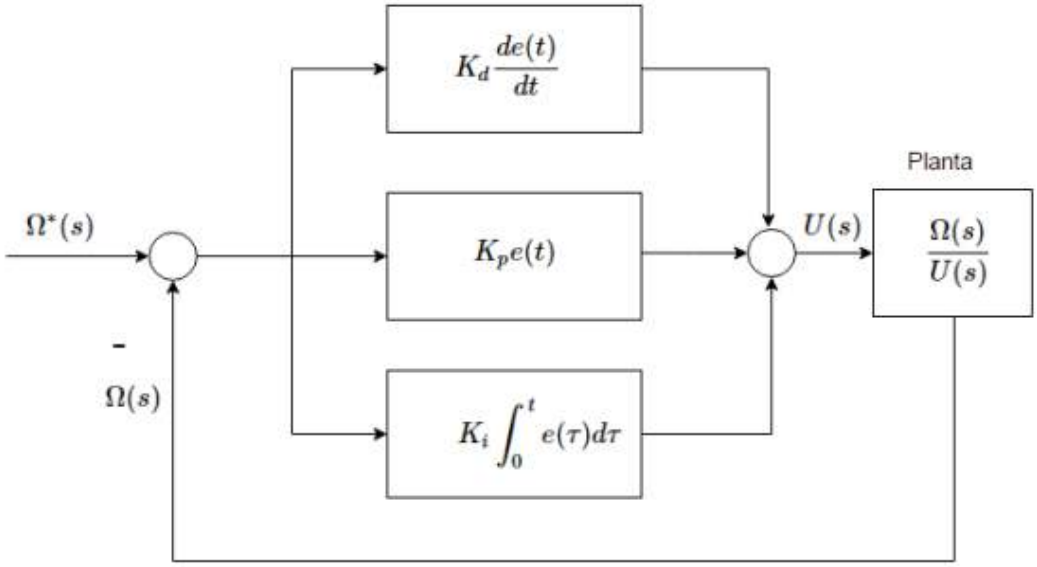


Figura 4 - Diagrama de blocos de um controlador PID. (NISE, 2012)

Aplicando a transformada de Laplace a fim de obter a lei de controle, obtém-se:

$$\frac{U(s)}{E(s)} = K_p + \frac{K_i}{s} + K_d s$$

1.2. Justificativa

Considerando o cenário brasileiro de robótica, ainda existem poucas competições de robótica que podem competir com o cenário mundial. O sucesso no desenvolvimento do projeto pode contribuir para elevar a qualidade dos robôs desenvolvimentos para estas competições.

1.3. Competições de robótica

1.3.1. Conceito e regras

Robôs seguidores de linha são robôs que têm o objetivo de seguir um percurso circular de maneira autônoma, no Brasil o trajeto é composto por uma linha branca contínua sobre uma superfície preta, normalmente feita de borracha. Na pista existem marcações do lado esquerdo e direito, sendo que a marcação esquerda representa início e fim de curvas, já marcações a direita representam o início e fim do trajeto. A figura abaixo ilustra um exemplo de pista.

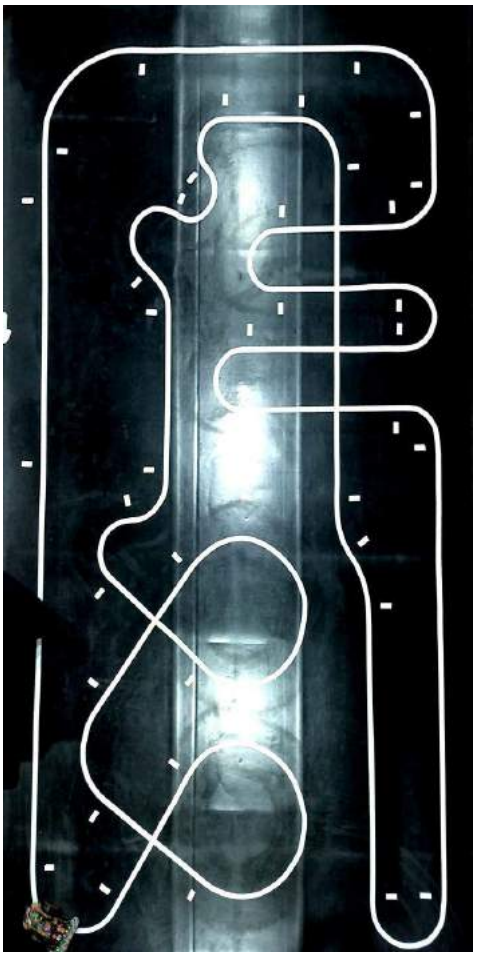


Figura 5 - Trajeto de pista da competição Iron Cup. Fonte: Autoria própria

As principais regras são definidas pela empresa *Robocore*, no qual existem limitações para o tamanho do robô. Dessa forma o robô deve ser construído para atender ao tamanho máximo

de 250x250x200 mm. Também não é permitida nenhuma ação que interfira com o vetor força normal (gerado pelo robô com relação ao solo) como o uso de ventoinhas (Robocore, 2016).

Vence a competição o robô que percorrer o percurso no menor tempo, sem que o corpo do robô saia inteiramente da pista. Os competidores podem realizar testes na pista antes da tomada de tempo oficial de forma livre e durante a tomada de tempo oficial. O competidor pode realizar 3 tentativas, de 3 minutos cada, sendo que a volta é validada quando o robô completar o trajeto sem que ele saia completamente da linha branca no solo (Robocore, 2016).

Também não é permitido que o robô faça o uso de qualquer tipo de comunicação com o competidor e nem com servidores externos durante a tomada de tempo oficial, dessa forma o robô deve ser capaz de parar dentre as marcações a direita que representam o início e fim de pista. A figura abaixo ilustra essa área de partida/chegada (Robocore, 2016).

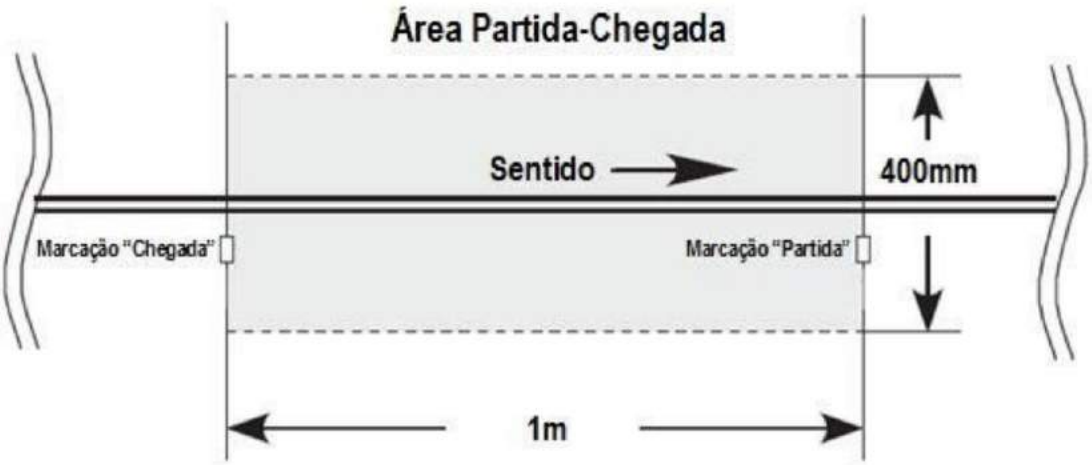


Figura 6 - Área de partida/chegada em competições de robótica (Robocore, 2016)

Todas as regras e especificações complementares estão disponíveis no site da *Robocore*, principal organizadora das competições.

1.4. Estrutura do documento

O trabalho está dividido em 6 capítulos. No primeiro capítulo apresenta-se a contextualização do projeto, descrevendo a problemática. Em seguida, no segundo capítulo apresenta o objetivo deste trabalho.

No terceiro capítulo é apresentado o referencial teórico para modelagem cinemática de robôs móveis com acionamento diferencial, além de apresentar sobre competições de robótica.

Faz-se a apresentação da metodologia para o desenvolvimento da modelagem cinemática, elétrica e todo o projeto do robô no quarto capítulo.

São apresentados os resultados obtidos com o projeto, tanto na modelagem cinemática, quanto no desenvolvimento proposto durante o quinto capítulo. Por fim o sexto capítulo apresenta as conclusões geradas a partir do trabalho realizado e algumas perspectivas para trabalhos futuros.

2. OBJETIVOS

O objetivo principal do trabalho é realizar a modelagem dinâmica de um robô seguidor de linha com acionamento diferencial e desenvolver o projeto do robô implementando um sistema de controle, a fim de obter um modelo estável, que seja capaz de participar de competições de robótica.

Como objetivos específicos este projeto visa:

- Modelar um sistema de controle para o robô;
- Projetar o modelo 3D do robô;
- Projetar o circuito elétrico do robô;
- Projetar o sistema embarcado do robô;
- Unir os sistemas acima em protótipo funcional;
- Otimizar o rendimento do robô.

3. METODOLOGIA

O trabalho se desenvolve a partir da prototipagem de um robô móvel com acionamento diferencial com aplicação para competição de robótica na categoria seguidor de linha, organizada pela empresa *Robocore*.

3.1. Descrição das etapas de implementação do projeto

A primeira etapa do projeto do robô consiste em projetar sua arquitetura de funcionamento quanto ao *firmware* baseadas em todas as tarefas que o dispositivo precisa realizar. Uma vez atribuídas as tarefas, são definidos quais os *inputs* e os *outputs* que o robô irá realizar, como por exemplo, sensores de entrada e atuadores de saída. Posteriormente é definido o microcontrolador responsável por realizar a interface entre o *hardware* e o *firmware* embarcado.

O projeto de funcionamento do robô pode ser fundamentado em módulos de alto nível, em diagramas de bloco sem especificar seus componentes eletrônicos de hardware necessários, porém é interessante selecionar os componentes nesta etapa, pois desta forma é possível saber com antecedência quais os recursos e informações serão necessárias.

Ao se trabalhar com projetos que envolvam alta velocidade a massa dos componentes se torna uma característica relevante, havendo desta forma a necessidade de se analisar os *datasheets* (folha de dados) dos componentes com mais detalhe. A massa é uma característica importante, pois baseando-se no caso particular da segunda lei de Newton, tem-se que:

$$F = m.a$$

Desta forma, pela relação de linearidade da equação, quanto menor a massa menor será a força necessária e assim menor também a aceleração para uma mesma força. Como a força também é proporcional ao torque de um motor, necessariamente quanto menor a massa do robô, melhor será para o seu deslocamento pois dessa forma é possível um melhor rendimento ao motor.

$$T = F \times D$$

Em relação ao projeto do robô, deve-se definir como serão tratadas cada uma das condições possíveis que o robô estará exposto, assim como desenhar o fluxo lógico.

Como segunda etapa para o projeto é necessário modelar um chassi que seja capaz de atender as condições de funcionamento, sejam estas: tamanho máximo, massa, material, modelo e dimensões.

Posteriormente na terceira etapa, prototipar o circuito elétrico com base nos componentes e dimensões do modelo do chassi, nesta etapa é necessária uma ótima sintonia entre a modelagem do circuito elétrico e a modelagem do chassis. Realizando modificações constantemente visando encontrar a melhor disposição dos componentes e aproveitamento de espaço.

Na quarta etapa, definir a linguagem de programação a ser implementada no microcontrolador, assim como a *Integrated development* environment (*IDE* – Ambiente de Desenvolvimento Integrado) de desenvolvimento e caso atenda, definir quais classes e bibliotecas serão desenvolvidas.

A quinta etapa se concentra em realizar a implementação do firmware junto do algoritmo de controle para realizar o controle do robô durante o seu funcionamento. Como já dito na introdução para que um robô móvel realize seus movimentos de forma precisa, é necessário realizar a implementação de um sistema de controle em malha fechada, pois desta forma é possível validar por meio de sensores se a saída projetada pelo microcontrolador está realmente sendo realizada com sucesso. É necessário então implementar um algoritmo de controle que funcione no sistema embarcado, o qual no caso, foi escolhido o algoritmo de controle proporcional, integrativo e derivativo (*PID*).

Para esta etapa foi desenvolvido um algoritmo simples de PID que recebe como parâmetros de entrada a velocidade atual, a velocidade desejada, constante proporcional, constante integral e constante derivativa.

A sexta etapa do projeto consiste em realizar a construção de todo o protótipo, ou seja, basicamente é realizar a integração dos resultados de cada uma das etapas anteriores.

Por fim, a sétima etapa do projeto é realizar os testes com o protótipo visando obter o melhor resultado para o modelo de controle e dinâmica veicular.

Como dito, pode ser necessária uma ou mais interações entre as etapas do projeto com o objetivo de otimizar cada uma das etapas, que não necessariamente precisam ser realizadas de forma separada ou sequenciais.

De forma visual, o fluxo abaixo representa o fluxo de implementação do projeto em uma forma resumida.

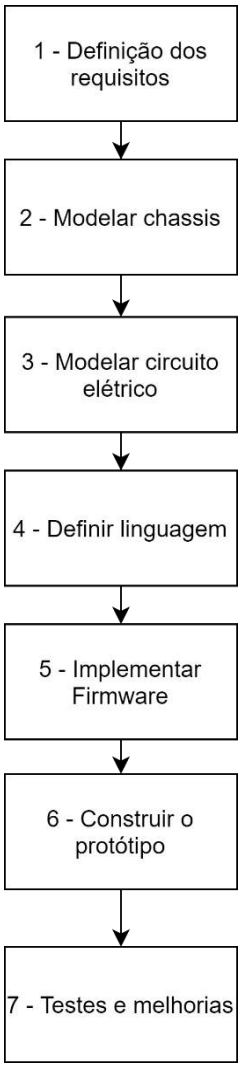


Figura 7 - Fluxo de implementação do projeto. Fonte: Autoria própria

3.2. Implementação do projeto

3.2.1. Etapa 1 - Definição dos requisitos

Um robô seguidor de linha é um sistema artificialmente inteligente capaz de detectar uma linha traçada no chão, no caso, em uma pista de borracha. No geral este tipo de robô percebe a linha através de sensores infravermelho (IR), e a partir disso um processador realiza a tomada de decisão com base na leitura dos dados dos sensores.

Analisando o objetivo no qual o projeto almeja alcançar e baseado em experiências técnicas passadas, verificou-se que o projeto deve atender aos seguintes requisitos:

- Identificar a linha a ser seguida;
- Identificar os marcadores de curva;
- Identificar marcador de início e fim de pista;
- Mensurar a rotação dos motores;
- Exibir status de funcionamento do robô;
- Ser leve, aproximadamente 200 gramas;
- Dimensão máxima: 250x250x200mm;
- Sistema de controle embarcado;
- Processamento rápido dos sensores de entrada;
- Inversão de polaridade dos motores;
- Velocidade linear mínima de 1 m/s;
- Comunicação para coleta de dados do robô;

Com estes requisitos em mente, tem-se o necessário para seguir para a concepção do diagrama de blocos do robô, definindo a arquitetura inicial do projeto e já alguns de seus componentes elétricos.

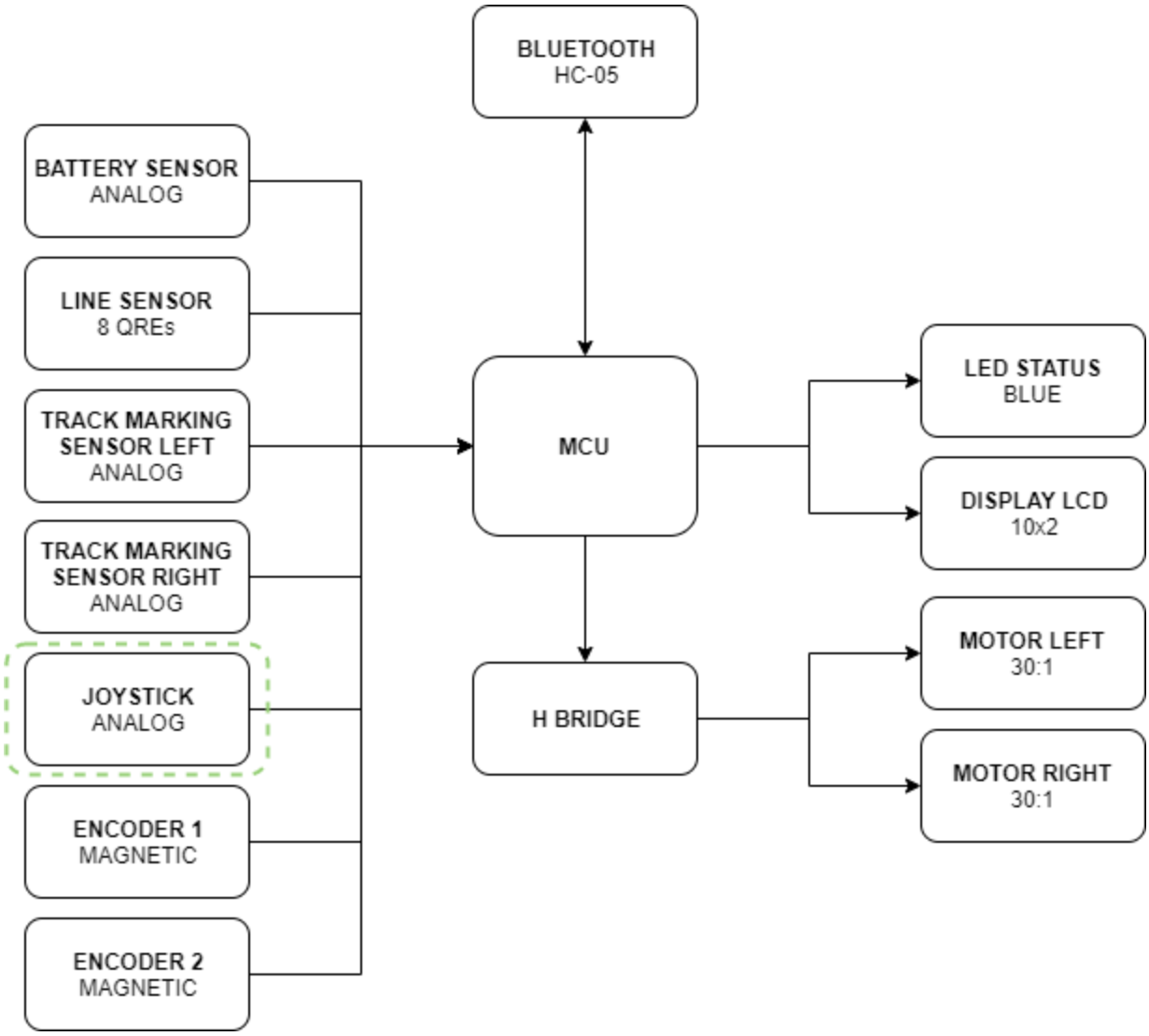


Figura 8 - Arquitetura inicial do robô. Fonte: Autoria própria

Como apresentado na figura 8 acima, na arquitetura inicial do robô um microcontrolador (MCU) fica responsável por receber os sinais lidos pelos sensores e realizar a tomada de decisão enviando os sinais para uma ponte H (*H Bridge)* que controla os motores. Desta forma o robô será composto pelas seguintes entradas e saídas:

Tabela 1 - Entradas e saídas do robô. Fonte: Autoria própria.

Entradas	**Saídas**
Bluetooth	Bluetooth

Sensor de bateria	LED
Sensor de linha	Display LCD
Sensor de marcação de pista esquerdo	Motor esquerdo
Sensor de marcação de pista direito	Motor direito
Encoder esquerdo	Ponte H
Encoder direito	

3.2.2. Etapa 2 - Modelagem dos chassis

Material Utilizado

- Caderno;
- Caneta;
- Régua;
- *Software Autodesk Fusion* 360;
- Balança de precisão;
- Impressora 3D;
- Filamentos para impressão 3D.

Visando atender os requisitos especificados a modelagem dos chassis ocorre a partir da iteração da busca pelo melhor design para o robô. A modelagem ocorreu nas seguintes etapas: esboço em papel do modelo, desenho do modelo no software de modelagem.

Os primeiros esboços do robô foram feitos a mão com base nos robôs já existentes nessa categoria, modelando com dois motores atrás e um apoio na frente. Começando o desenho a partir de uma base retangular, os cortes foram sendo feitos, chegando no modelo inicial conforme a figura abaixo.

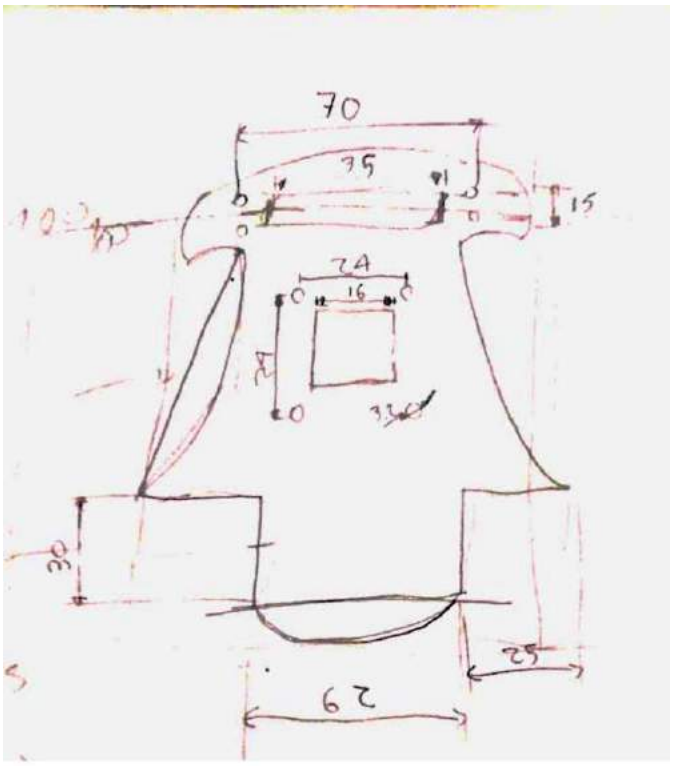


Figura 9 - Primeiro esboço do robô. Fonte: Autoria própria

Posteriormente o esboço foi modelado no software *Fusion 360,* disponibilizado pela *Autodesk*, dessa forma pode-se ter uma ideia mais aprofundada do design do robô conforme a figura abaixo. Percebe-se a necessidade de alguns ajustes, pois nesse primeiro esboço o chassis está com muita área, o que torna o robô pesado.

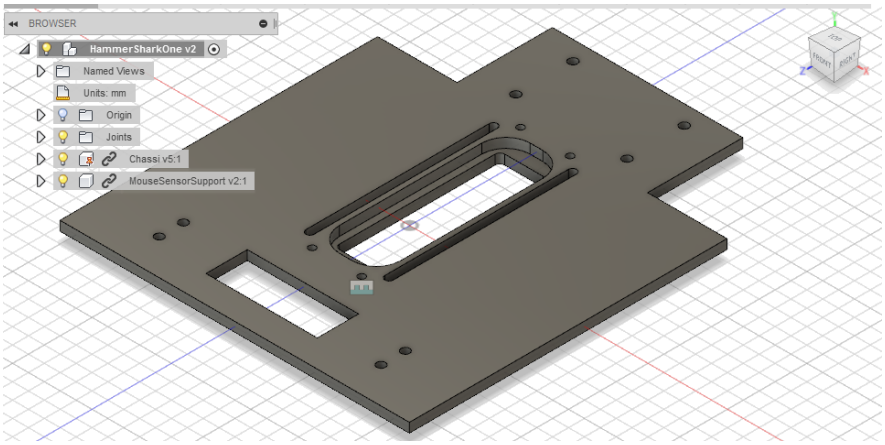


Figura 10 - Primeira modelagem do robô. Fonte: Autoria própria

Como já dito, esta etapa de modelagem requer várias iterações em busca da melhor modelagem do robô. Os requisitos apresentados pedem que o robô seja leve, de forma que seja aplicável o melhor rendimento aos motores, pensando dessa forma o modelo anterior foi alterado removendo o máximo possível de estruturas dos chassis, deixando apenas as principais sustentações e “braços” abertos para posicionar os sensores de marcação de pista. O resultado intermediário das interações pode ser visto na imagem abaixo:

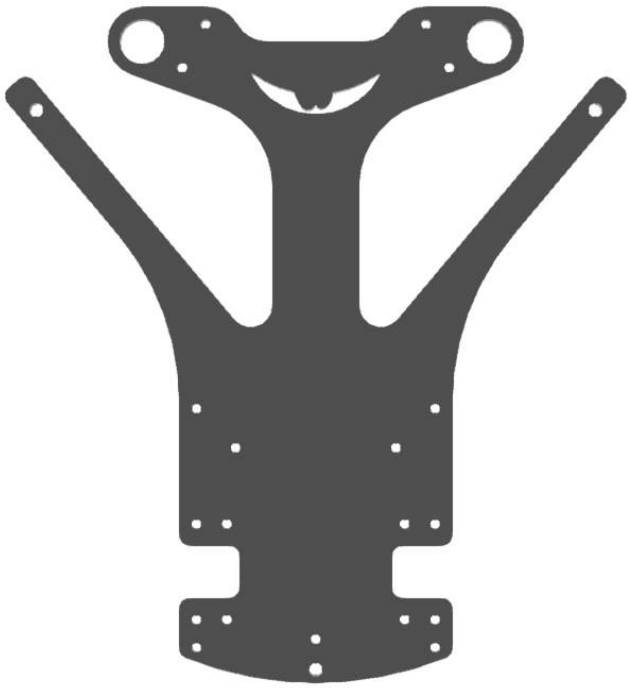


Figura 11 - Chassis do robô. Fonte: Autoria própria

Este modelo consiste em apresentar o chassis do robô na vista superior, contento todos os furos para alocar os sensores, PCB e motores.

Visando ainda uma maior otimização na massa do robô, pode-se perceber que na parte inferior do robô será adicionada a PCB e desta forma, pode-se utilizar a própria PCB como chassis, resultando assim nas interações finais para o modelo do robô apresentado na figura abaixo.

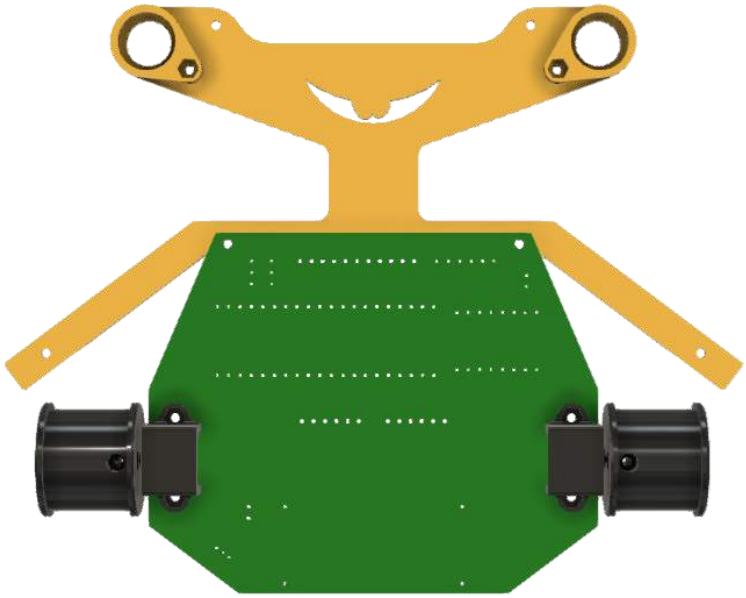


Figura 12 - Modelo 3D do chassis do robô. Fonte: Autoria própria

3.2.3. Etapa 3 - Modelagem do circuito elétrico

Material Utilizado

- *Software Circuit Maker*;
- *Maple Mini ARM STM32 Development Board;*
- *QTR-8A Reflectance Sensor Array;*
- *TB6612FNG Dual Motor Driver Carrier;*
- *Bluetooth Module HC-05;*
- *30:1 Micro Metal Gearmotor HP 6V with Extended Motor Shaft;*
- *Magnetic Encoder Pair Kit for Micro Metal Gearmotors;*
- *Profuse 2S 7.4V 850 Mah LiPo Battery;*
- Estanho para solda liga 60%Sn/40%Pb;
- Placa padrão.

Dados os requisitos da etapa 1, o circuito elétrico é modelado a partir dos conceitos dispositivos eletrônicos e condicionamento de sinais para coleta de dados dos sensores.

Primeiro foi feito um trabalho de análise das massas de cada um dos componentes, visando ter a melhor relação entre massa x funcionalidade. Dessa forma foi obtida a seguinte tabela de componentes e suas respectivas massas:

Tabela 2 - Massa teórica dos componentes do robô. Fonte: Autoria própria

Componente	QTD (UN)	Massa (g)	Massa Total (g)
PCB + Microcontrolador + Driver	1	27	27
QTR-8A Reflectance Sensor Array	1	3	3
Rodas	2	22.5	45
Chassi	1	25	25
Suporte do motor	2	4	8
Ball Caster Holder	2	1	2
Suporte da PCB	1	6	6
Spacer	1	6	6
Motores + Encoder + Cabos	2	13	26
Bateria	1	35	35
Total	**187**	**142,5**	**183**

A partir dos componentes definidos foi montado o circuito elétrico no *software Circuit Maker* para cada uma das principais funcionalidades do robô. Dessa forma os circuitos foram separados em 4 grupos: *Core* (principal), Alimentação, Comunicação e Sensores.

3.2.3.1. Circuito Principal

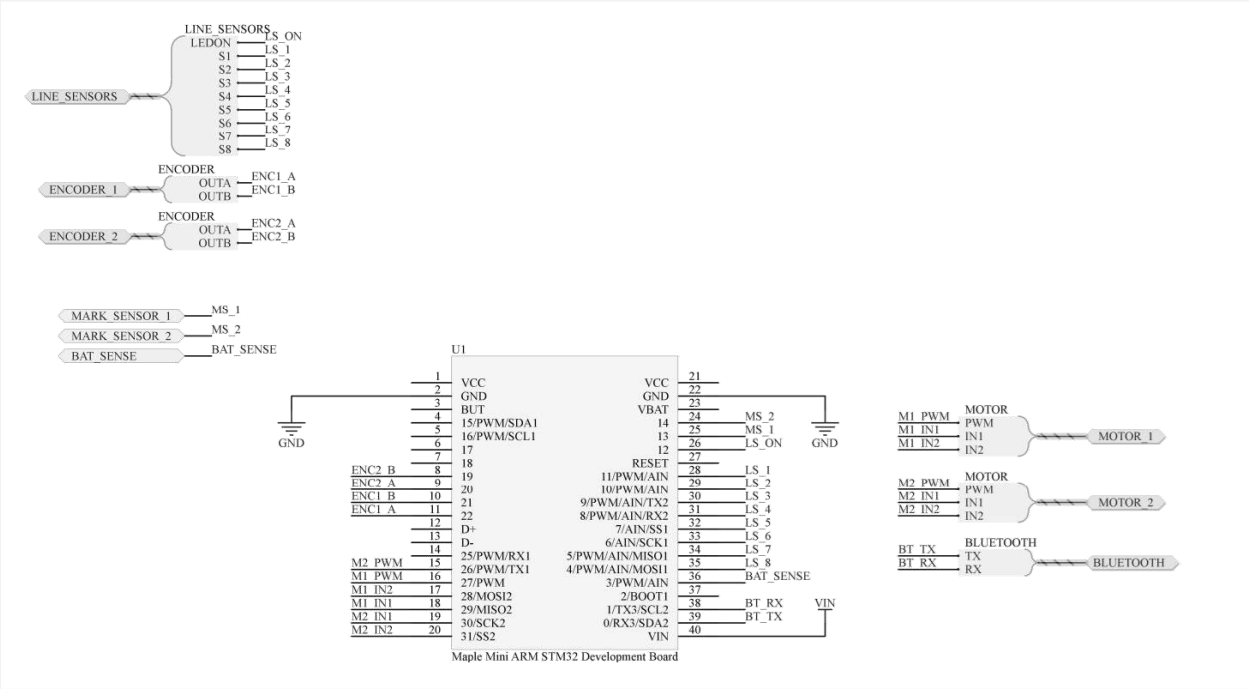


Figura 13 - Circuito principal do robô. Fonte: Autoria própria

O circuito principal é composto pelo processador Arm que se conecta com o restante do circuito, motores, módulo *Bluetooth*, sensores de infravermelhos e *encoder*. Na imagem acima é possível ver todo o esquema de pinagem elaborado.

3.2.3.2. Circuito de alimentação

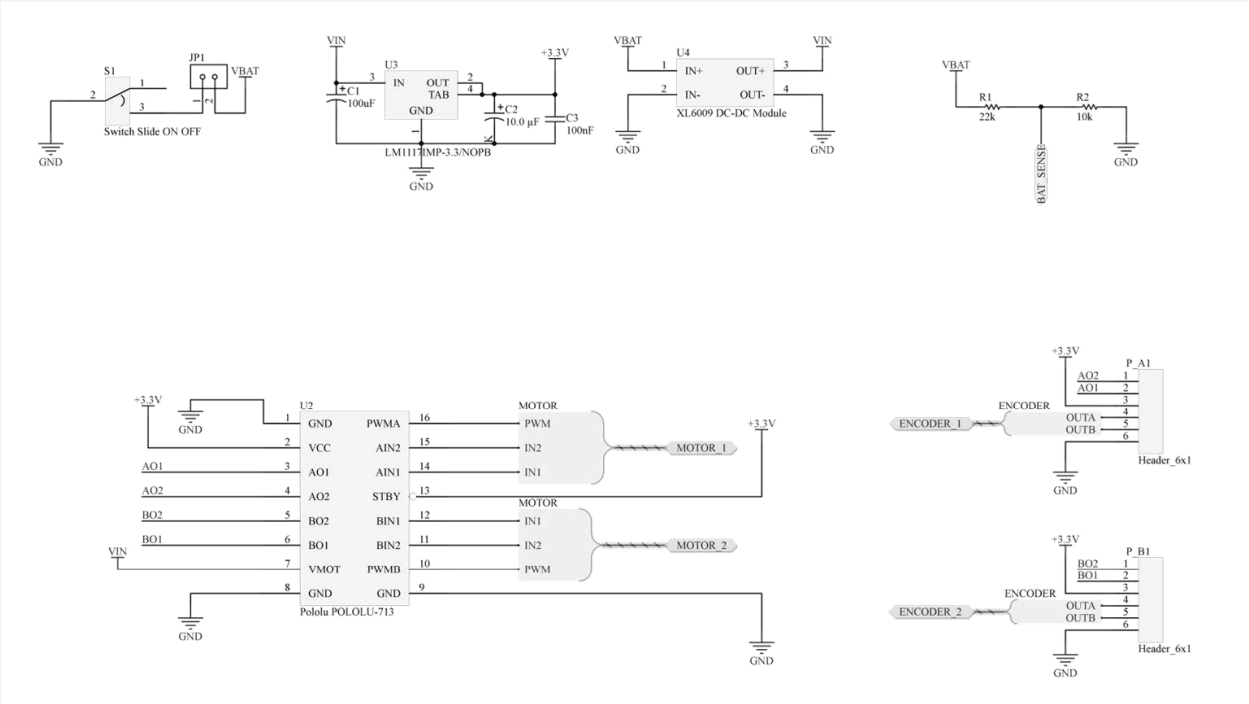


Figura 14 - Circuito de alimentação. Fonte: Autoria própria

O circuito de alimentação do robô é o circuito responsável por fornecer energia elétrica a todo o sistema. Para alimentação dos motores foi utilizado um circuito do tipo Ponte-H por meio do componente *TB6612FNG,* já para alimentação dos componentes foi necessário o uso de um regulador de tensão para 3.3 V, realizado pelo componente LM1117IMP-3.3, nos encoders foi utilizado um header de conexão, visando facilitar as conexões e cabos. Por fim, um *switch* liga e desliga, para acionamento do circuito conectado à bateria que alimenta o restante do robô.

3.2.3.3. Circuito dos sensores

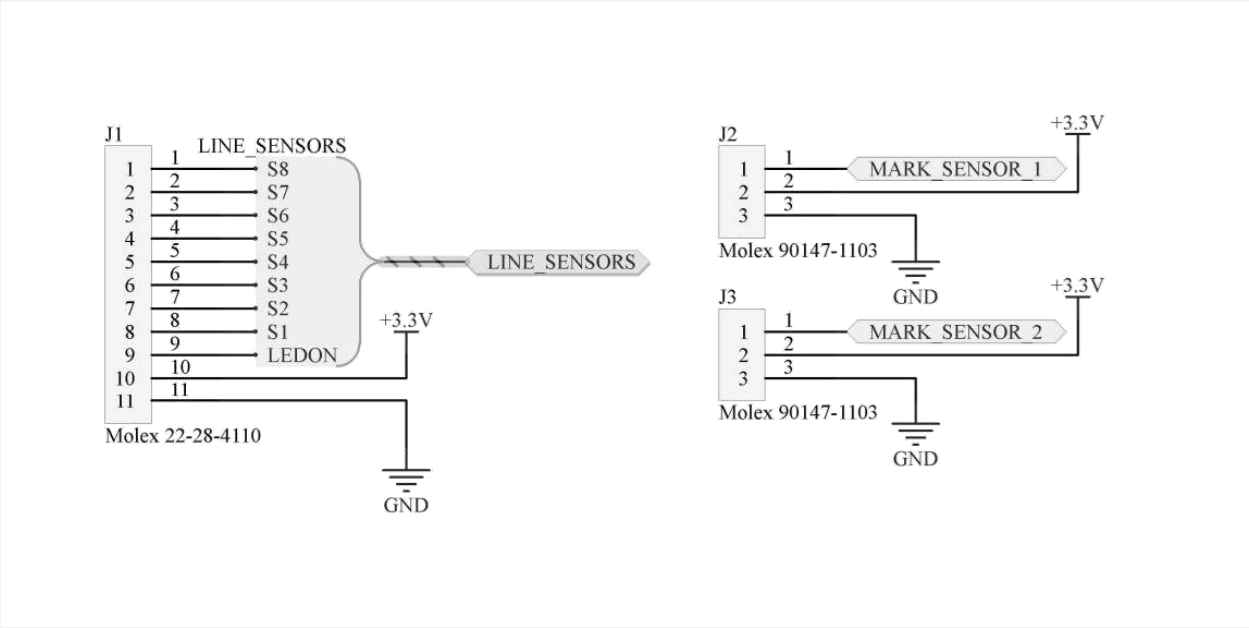


Figura 15 - Circuito dos sensores. Fonte: Autoria própria

Um circuito simples que conecta o sensor de linha *QTR-8A Reflectance Sensor Array* a um multiplexador alimentado por 3.3 V e os dois sensores de marcação de pista conectados a dois multiplexadores, também alimentados por 3.3 V.

3.2.3.4. Circuito de comunicação

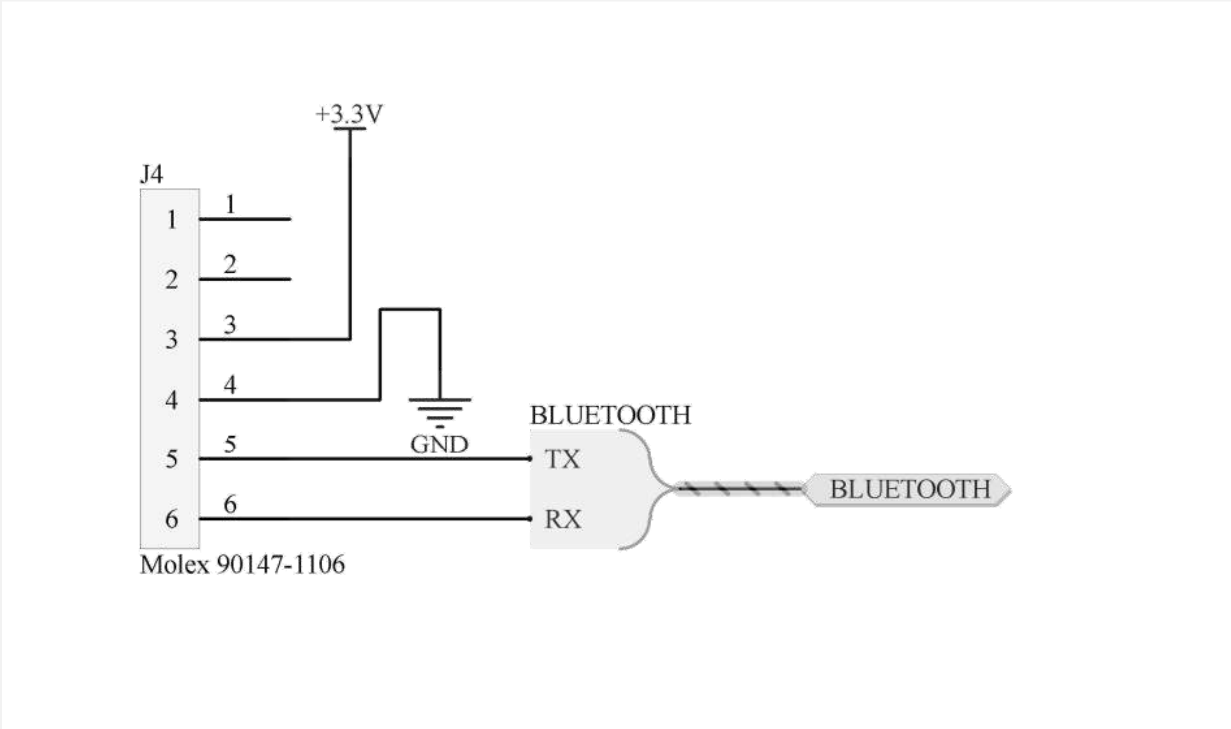


Figura 16 - Circuito de comunicação do robô. Fonte: Autoria própria

Para realizar a comunicação do robô foi utilizado a comunicação via Bluetooth, por meio do componente *HC-05* que permite uma comunicação bidirecional para enviar e receber dados. Dessa forma é possível coletar dados gerados pelo robô e analisar posteriormente. Este circuito também é simples, uma conexão entre um multiplexado e o módulo *Bluetooth HC-05*.

3.2.3.5. PCB

Por fim, foi desenvolvido a PCB que modela os circuitos acima.

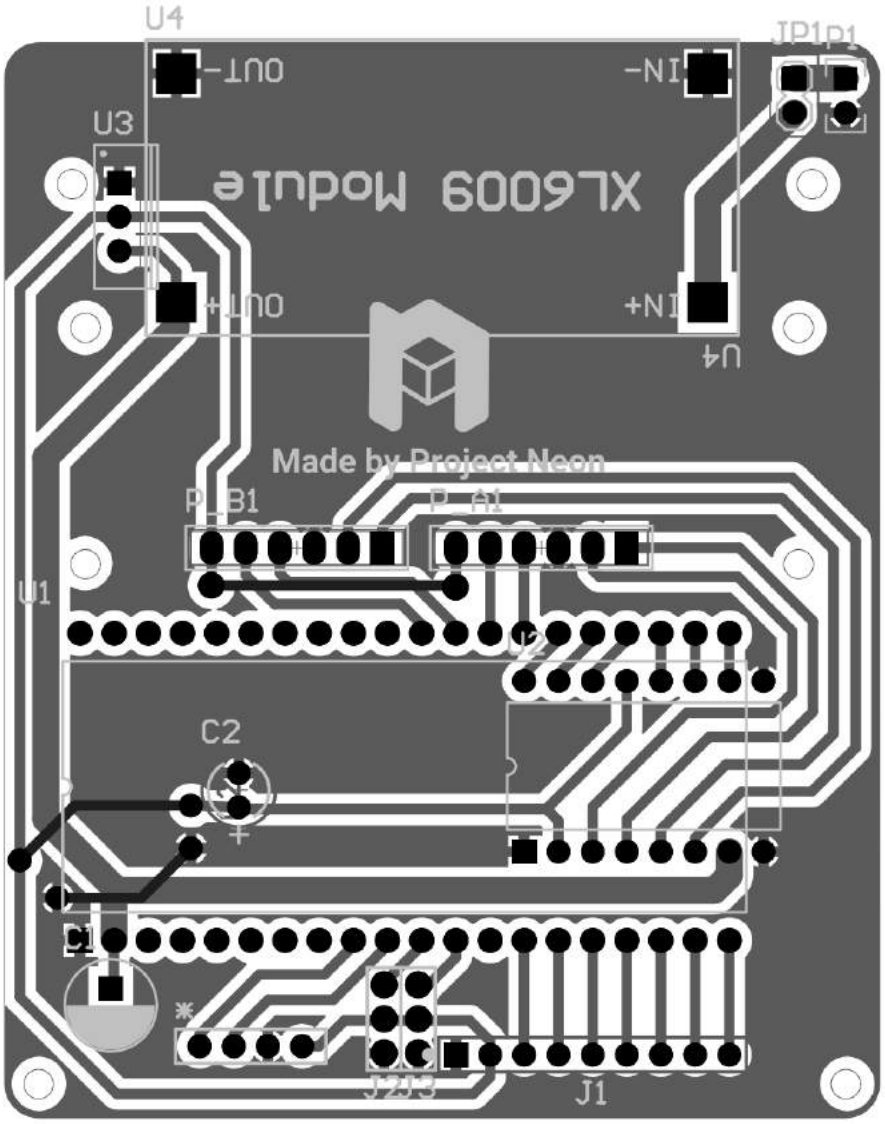


Figura 17 - PCB do robô. Fonte: Autoria própria

3.2.4. Etapa 4 - Definição da linguagem de programação

Material Utilizado

- *Microsoft* Visual Studio Code;
- *IDE PlatformIO* para *VS Code*;

Visando um firmware otimizado, o hardware escolhido foi um processador do tipo *Arm* desta forma a linguagem de programação escolhida foi C++ por permitir realizar o uso de classes e todo o universo de bibliotecas já desenvolvidas disponíveis na *internet*.

Como IDE foi utilizado o *PlatformIO* pois é uma IDE completa capaz de obter as ferramentas necessárias para cada tipo de controlador e fazer o *download* delas automaticamente. Desta forma a configuração para qualquer *MCU* torna-se simples, localizando as bibliotecas e exemplos para o *MCU* gastando menos tempo na configuração do ambiente de desenvolvimento.

3.2.5. Etapa 5 - Implementação do firmware

Material Utilizado

- *Microsoft* Visual Studio Code;
- *IDE PlatformIO* para *VS Code*;
- QEI Library by Aaron Berk, 2010.
- PololuQTRSensors Library by Pololu, 2015;

Seguindo a etapa anterior à implementação do firmware foi desenvolvida por meio da *IDE PlatformIO* instalada no *Microsoft Visual Studio Code*. Inicialmente realizou o teste do ambiente de desenvolvimento com os exemplos do fabricante, definição da arquitetura, definição das bibliotecas e testes com os sensores e por fim integração de todos os programas desenvolvidos.

A figura 18 a seguir apresenta a arquitetura do robô, onde o *robot.h* contém as configurações do robô como parâmetros de tamanho do eixo, roda, tipo de comunicação, pista atual, constantes de controle e pinos dos sensores / atuadores. A partir disso o robô possui bibliotecas que fazem a declaração dos sensores assim como a leitura dos dados, e transmitem esses dados para outras bibliotecas até o sistema de controle PID que realiza o ajuste na velocidade do motor, finalizando com a tomada de decisão dada pelo *line follower*, o programa principal.

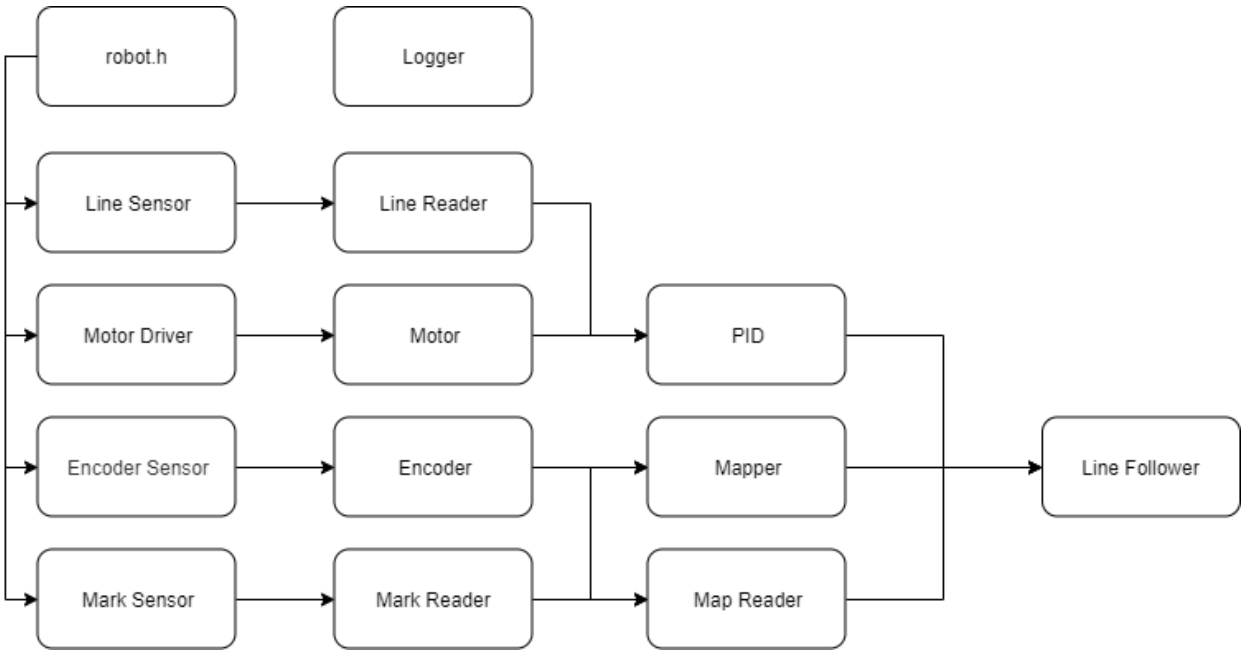


Figura 18 - Arquitetura do firmware do robô. Fonte: Autoria própria.

Posteriormente depois da definição da arquitetura do robô foi realizado o teste com os sensores para validar que o sistema proposto era capaz de realizar a leitura dos sensores, assim como processar os dados. A figura 19 ilustra o teste realizado com o sensor de linha *QTR-8A* e a biblioteca *PololuQTRSensors*.

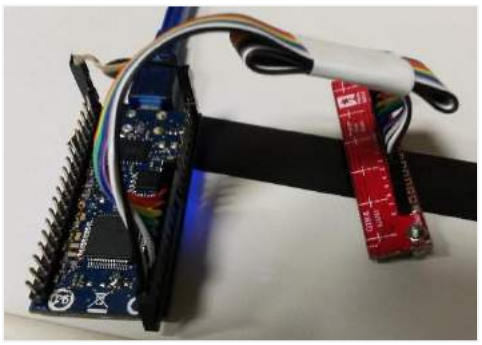


Figura 19 - Teste com o sensor de linha QTR-8A e biblioteca PololuQTRSensors. Fonte: Autoria própria.

Para entender o comportamento dos motores de acordo com a tensão aplicada, realizou-se o ensaio da curva de velocidade dos motores e confirmou-se com o referencial teórico esperado que a velocidade cresce de forma linear. A figura abaixo apresenta a curva em RPS por tensão aplicada nos motores:

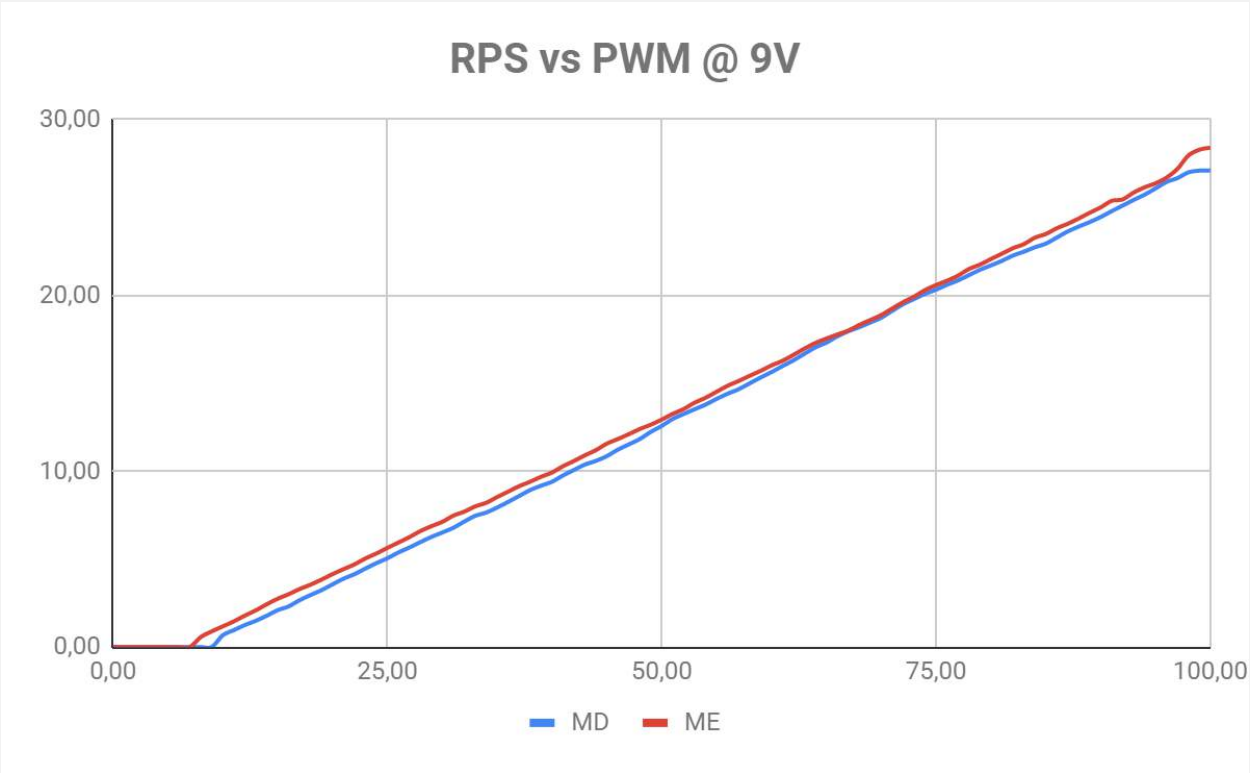


Figura 20 - Curva de velocidade dos motores elétricos. Fonte: Autoria própria

Com o sucesso dos testes para cada um dos componentes, a implementação da arquitetura do software foi desenvolvida.

3.2.5.1. Motores

A biblioteca para implementação dos motores foi desenvolvida por completo neste escopo do projeto, para que fosse capaz de configurar os motores, alterar velocidades, parar e desligar os motores. As configurações dos motores podem ser definidas por:

```
Motor::Motor(uint8_t pinPwm, uint8_t in1, uint8_t
in2)
{
    _in1 = in1;
    _in2 = in2;
    _pinPwm = pinPwm;
    // Setup the PWM frequency to ~95kHz (Timer
One)
    timer.setPrescaleFactor(1);
    timer.setOverflow(760);

    pinMode(_pinPwm, PWM);
    pinMode(_in1, OUTPUT);
    pinMode(_in2, OUTPUT);

    //Start in zero
    pwmWrite(_pinPwm, 0);
    digitalWrite(_in1, 0);
    digitalWrite(_in2, 0);
}
```

Código Fonte 1 - Configuração dos motores. Fonte: Autoria própria

Para o controle de velocidades dos motores, foi implementado uma lógica para receber um valor de velocidade e converter para a porta analógica de saída PWM por meio da função map.

```
void Motor::speed(float speed) {
    float power = speed > 0 ? speed : -speed;
    pwmWrite(_pinPwm, map(power, 0, 100, 0, 761));
    digitalWrite(_in1, speed >=  0 ? HIGH : LOW);
  digitalWrite(_in2, speed >   0 ? LOW : HIGH);
}
```

Código Fonte 2 - Função de implementação de velocidade nos motores. Fonte: Autoria própria

De forma a realizar a parada dos motores, foram implementadas duas funções: Uma função para desligar os motores e dessa forma o robô irá parar devido ao atrito com o solo e outra função para realizar um curto-circuito nos motores, fazendo com que o robô pare imediatamente.

```
void Motor::coast() {
    digitalWrite(_in1, 0);
    digitalWrite(_in2, 0);
}

void Motor::brake() {
    digitalWrite(_in1, 1);
    digitalWrite(_in2, 1);
}
```

Código Fonte 3 - Funções de parada dos motores. Fonte: Autoria própria

Por fim a classe implementada para os motores que realiza as funções acima, é descrita pelo seguinte código:

```
#include <Arduino.h>
#ifndef Motor_h
#define Motor_h


class Motor {
    public:
        Motor(uint8_t pinPwm, uint8_t in1, uint8_t in2);
        void speed(float speed);
        void coast();
        void brake();
    protected:
        uint8_t _pinPwm;
        uint8_t _in1;
        uint8_t _in2;
};

#endif
```

Código Fonte 4 - Classe para implementação dos motores. Fonte: Autoria própria

3.2.5.2. Encoder

Para implementação da comunicação com o *encoder* foi desenvolvido uma biblioteca com o objetivo de facilitar a leitura dos dados dos sensores. A biblioteca é composta por 5 principais funções: configuração, leitura de pulsos, leitura de rotações, leitura de distância e redefinição.

A configuração dos *encoders* foi definida pelo código abaixo, sendo necessário o uso de interrupções pois a leitura dos dados deve ser em tempo real.

```
Encoder::Encoder(uint8_t pinA, uint8_t pinB, int pulsesPerRev, float wheelRadius)
{
  _pulsesPerRev = pulsesPerRev > 0 ? pulsesPerRev : 0;
  _wheelRadius = wheelRadius > 0 ? wheelRadius : 0;
  _pinA = pinA;
  _pinB = pinB;
  _pulses=0;
  _revolutions=0;
  _distance=0;

  attachInterrupt(digitalPinToInterrupt(_pinA), encoderChannelA, RISING);
  attachInterrupt(digitalPinToInterrupt(_pinB), encoderChannelB, RISING);

}
```

Código Fonte 5 - Configuração dos *encoders*. Fonte: Autoria própria

De forma interrupta, uma variável controla os pulsos lidos pelo encoder, da seguinte forma:

```
void encoderChannelA(){
 if (digitalRead(_pinA)== HIGH) {
     _pulses++;
 }
 else{
   _pulses--;
 }
}
void encoderChannelB(){
 if (digitalRead(_pinB)== HIGH) {
     _pulses--;
 }
 else{
   _pulses++;
 }
}
```

Código Fonte 6 - Leitura de pulsos do encoder. Fonte: Autoria própria

As demais funções da biblioteca utilizam os dados da variável *_pulses* para realizar as contas necessárias e retornar os resultados por meio das funções. A imagem abaixo ilustra as funções.

```
int Encoder::getPulses(){
 return _pulses;
}
float Encoder::getRevolutions(){
 _revolutions = _pulsesPerRev > 0 ?
  _pulses/_pulsesPerRev : 0;
 return _revolutions;
}
float Encoder::getDistance(){
 return 2*PI*_wheelRadius*_revolutions;
}

void Encoder::reset(){
 _revolutions=0;
 _pulses=0;
}
```

Código Fonte 7 - Funções do encoder. Fonte: Autoria própria

Por fim a classe implementada para os *encoders* que realiza as funções acima, é descrita pelo seguinte código:

```
#include <Arduino.h>
#ifndef Encoder_h
#define Encoder_h



class Encoder {
   public:
       Encoder(uint8_t pinA, uint8_t pinB, int pulserPerRev, float wheelRadius);
       float getDistance(); //return distance in meters
       int getPulses();
       float getRevolutions();
       void reset();

   protected:
       int _pulsesPerRev;
       float _revolutions;
       float _distance;
       float _wheelRadius;
};

#endif

```

Código Fonte 8 - Classe dos encoders. Fonte: Autoria própria

3.2.5.3. Sensores

Para implementação da comunicação com os sensores foi utilizado a biblioteca *PololuQTRSensors* desenvolvida pela Polulu (Pololu, 2019). Nessa biblioteca foi necessário utilizar 3 funções principais: calibrar, ler sensores e ler linha.

A calibração dos sensores é realizada pela função *calibrate( )*, definida pelo seguinte trecho de código:

```
1  void QTRSensors::calibrate(unsigned char readMode)
2  {
3      if(readMode == QTR_EMITTERS_ON_AND_OFF ||
   readMode == QTR_EMITTERS_ON)
4      {
5          calibrateOnOrOff(&calibratedMinimumOn,
6                           &calibratedMaximumOn,
7                           QTR_EMITTERS_ON);
8      }
9
10
11     if(readMode == QTR_EMITTERS_ON_AND_OFF ||
   readMode == QTR_EMITTERS_OFF)
12     {
13         calibrateOnOrOff(&calibratedMinimumOff,
14                          &calibratedMaximumOff,
15                          QTR_EMITTERS_OFF);
16     }
17 }
```

Código Fonte 9 - Função calibrar sensores de linha. Fonte: (Pololu, 2019)

Já a função para realizar a leitura dos dados dos sensores é implementada da seguinte forma:

```
void QTRSensors::readCalibrated(unsigned int *sensor_values, unsigned char readMode)
{
    int i;

    // if not calibrated, do nothing
    if(readMode == QTR_EMITTERS_ON_AND_OFF || readMode == QTR_EMITTERS_OFF)
        if(!calibratedMinimumOff || !calibratedMaximumOff)
            return;
    if(readMode == QTR_EMITTERS_ON_AND_OFF || readMode == QTR_EMITTERS_ON)
        if(!calibratedMinimumOn || !calibratedMaximumOn)
            return;

    // read the needed values
    read(sensor_values,readMode);

    for(i=0;i<_numSensors;i++)
    {
        unsigned int calmin,calmax;
        unsigned int denominator;

        // find the correct calibration
        if(readMode == QTR_EMITTERS_ON)
        {
            calmax = calibratedMaximumOn[i];
            calmin = calibratedMinimumOn[i];
        }
        else if(readMode == QTR_EMITTERS_OFF)
        {
            calmax = calibratedMaximumOff[i];
            calmin = calibratedMinimumOff[i];
        }
        else // QTR_EMITTERS_ON_AND_OFF
        {

            if(calibratedMinimumOff[i] < calibratedMinimumOn[i]) // no meaningful signal
                calmin = _maxValue;
            else
                calmin = calibratedMinimumOn[i] + _maxValue - calibratedMinimumOff[i]; // this won't go past
_maxValue

            if(calibratedMaximumOff[i] < calibratedMaximumOn[i]) // no meaningful signal
                calmax = _maxValue;
            else
                calmax = calibratedMaximumOn[i] + _maxValue - calibratedMaximumOff[i]; // this won't go past
_maxValue
        }

        denominator = calmax - calmin;

        signed int x = 0;
        if(denominator != 0)
            x = (((signed long)sensor_values[i]) - calmin)
                * 1000 / denominator;
        if(x < 0)
            x = 0;
        else if(x > 1000)
            x = 1000;
        sensor_values[i] = x;
    }

}
```

Código Fonte 10 - Função ler dados dos sensores de linha calibrados. Fonte: (Pololu, 2019)

Por fim a leitura da linha completa é dada por uma média ponderada entre os sensores, resultando em um valor 0 na extrema esquerda e máximo nas extremidades (5000):

```
int QTRSensors::readLine(unsigned int *sensor_values,
    unsigned char readMode, unsigned char white_line)
{
    unsigned char i, on_line = 0;
    unsigned long avg; // this is for the weighted total, which is long
                       // before division
    unsigned int sum; // this is for the denominator which is <= 64000

    readCalibrated(sensor_values, readMode);

    avg = 0;
    sum = 0;

    for(i=0;i<_numSensors;i++) {
        int value = sensor_values[i];
        if(white_line)
            value = 1000-value;

        // keep track of whether we see the line at all
        if(value > 200) {
            on_line = 1;
        }

        // only average in values that are above a noise threshold
        if(value > 50) {
            avg += (long)(value) * (i * 1000);
            sum += value;
        }
    }

    if(!on_line)
    {
        // If it last read to the left of center, return 0.
        if(_lastValue < (_numSensors-1)*1000/2)
            return 0;

        // If it last read to the right of center, return the max.
        else
            return (_numSensors-1)*1000;

    }

    _lastValue = avg/sum;

    return _lastValue;
}
```

Código Fonte 11 - Função ler linha completa dos sensores de linha. Fonte: (Pololu, 2019)

Já para os sensores de marcação de pista, a leitura é feita por meio de interrupção na porta digital.

```
void setupMarkSensors()
{
  attachInterrupt(digitalPinToInterrupt(PIN_TRACK_
  MARKING_RIGHT), checkpointSensorRightCallback,
  FALLING);

  attachInterrupt(digitalPinToInterrupt(PIN_TRACK_
  MARKING_LEFT), checkpointSensorLeftCallback,
  FALLING);

}
```

Código Fonte 12 - Leitura sensores de marcação. Fonte: Autoria própria

3.2.5.4. Controle PID

Para implementação do sistema de controle PID, foi desenvolvido uma classe que fica responsável por realizar os cálculos a partir das constantes recebidas. As constantes no primeiro momento foram obtidas de forma empírica, visualmente de acordo com o comportamento do robô mediante a alteração das constantes.

A classe possui como funcionalidade principal a função *compute( )* que retorna a somatória do fator proporcional, integral e derivativo.

```
float compute(){

        error = setPoint - sample;
        timer = millis();
        float deltaTime =  timer - lastTimer;
        lastTimer = timer;
        P = error * kP;
        I += (error * kI) * deltaTime;
        D = ((lastSample - sample) * kD) / deltaTime;

        lastSample = sample;

        pid = P + I + D;

        return pid;
    }
```

Código Fonte 13 - Função de controle do PID. Fonte: Autoria própria

No código fonte acima pode-se observar que o algoritmo do PID foi implementado de acordo com o referencial teórico.

O fator P (proporcional) é definido por uma constante kP multiplicada pelo erro.

O fator I (integral) é definido pela somatória do produto entre o erro, a constante kI e o intervalo de tempo.

O fator D (derivativo) é definido pela diferença entre as últimas amostras vezes a constante kD, dividido pelo intervalo de tempo. O código completo da classe pode ser visto no trecho abaixo.

```
#ifndef PID_H
#define PID_H

#include "Arduino.h"

class PID {
public:

    PID(float kP, float kI, float kD){
        this->kP = kP;
        this->kI = kI;
        this->kD = kD;
        lastTimer = 0;
    }

    void setProcessValue(float sample){
        this->sample = sample;
    }

    void setSetPoint(float setPoint){
        this->setPoint = setPoint;
    }

    void setTunings(float kP, float kI, float kD){
        this->kP = kP;
        this->kI = kI;
        this->kD = kD;
    }

    float compute(){

        error = setPoint - sample;
        timer = millis();
        float deltaTime =  timer - lastTimer;
        lastTimer = timer;
        P = error * kP;
        I += (error * kI) * deltaTime;
        D = ((lastSample - sample) * kD) / deltaTime;

        lastSample = sample;

        pid = P + I + D;

        return pid;
     }

     protected:
        float timer;
        float lastTimer;
        float error;
        float sample;
        float lastSample;
        float kP, kI, kD;
        float P, I, D;
        float pid;
        float setPoint;
};

#endif
```

Código Fonte 14 - Classe PID. Fonte: Autoria própria

3.2.5.5. Mapeamento de pista

O comportamento normal de um robô na pista com sistema de controle PID é manter uma velocidade constante que seja aplicável tanto para curvas quanto para retas, mas nessa forma de implementação não temos uma otimização do rendimento do robô. Pois o robô não consegue utilizar sua velocidade máxima nas retas. A fim de otimizar o rendimento do robô na pista foi implementado um mecanismo de mapeamento da pista, nessa implementação o robô é capaz de identificar retas e curvas com diferentes raios, possibilitando alternar sua velocidade de

setpoint de acordo com as condições da pista. Na imagem abaixo segue um exemplo de pista que foi utilizada nessa implementação.

Figura 21 - Pista da competição Smiles. Fonte: Autoria própria

Para implementar esse recurso o robô realiza uma volta completa na pista com uma velocidade de *setpoint* conservadora e parâmetros de PID otimizados previamente com testes, com a comunicação via *Bluetooth* é possível receber os dados de distância percorrida e marcação de pista, por meio dos encoders e sensores de marcação. Dessa forma pode-se gerar o seguinte gráfico:

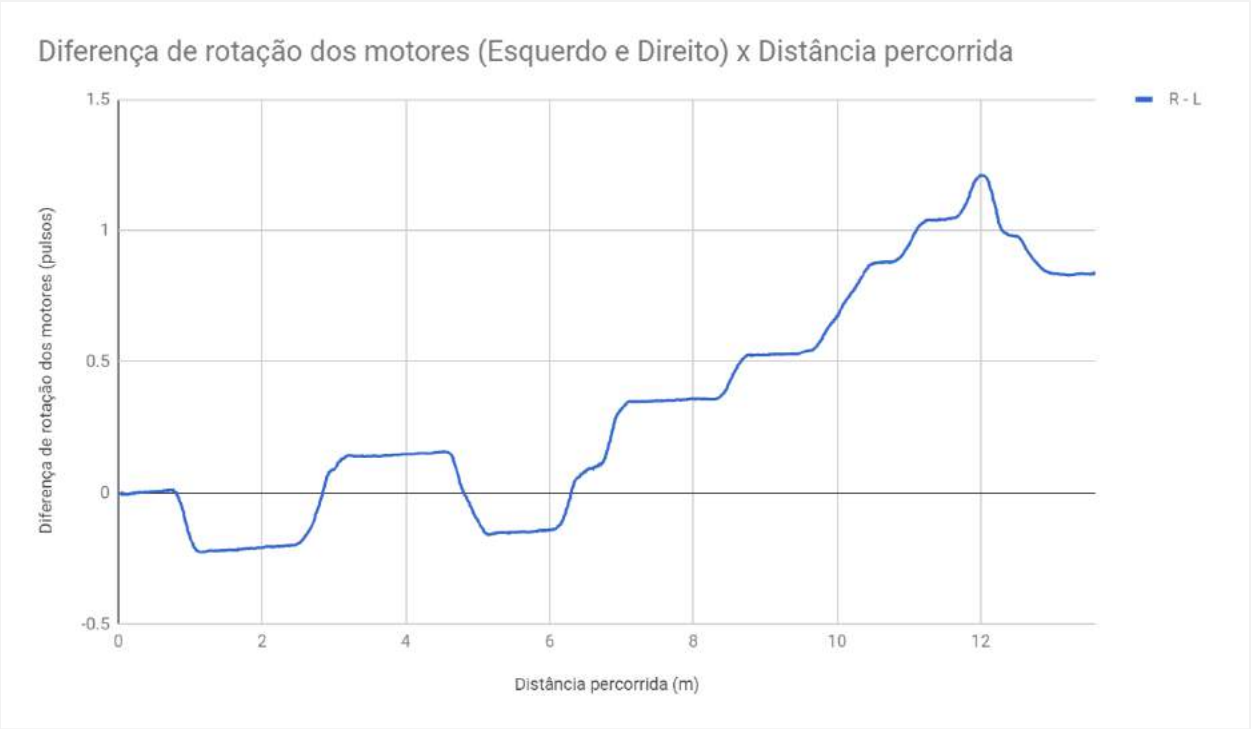


Figura 22 - Gráfico para mapeamento de pista. Fonte: Autoria própria

Na figura 22 pode-se analisar a diferença de rotação dos motores pela distância percorrida, dessa forma quando a diferença de rotação dos motores é zero, indica que o robô está em uma linha reta. Quando a diferença é positiva indica que o lado direito está mais rápido que o esquerdo, logo o robô está fazendo uma curva para esquerda (acionamento diferencial).

Com este gráfico detalhado podemos entender melhor o comportamento do robô na pista, inclusive, avaliar o controle PID que quanto menor a diferença de rotação entre as rodas, mais linear o robô está se movendo. Abaixo a imagem representa os pontos considerados relevantes na pista, para que haja alteração de velocidade.

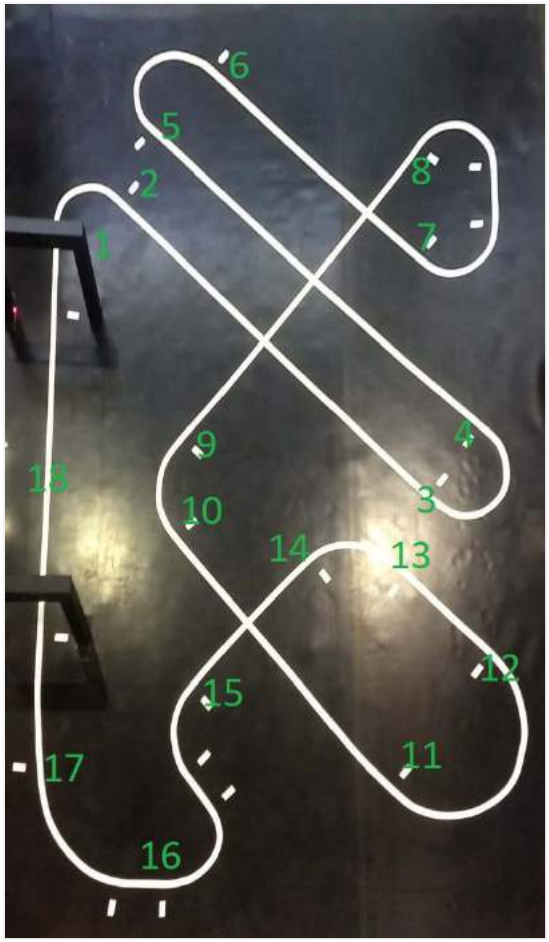


Figura 23 - Pista da competição Smiles com pontos críticos. Fonte: Autoria própria

Com a pista mapeada e seus pontos críticos, a implementação foi realizada da seguinte forma:

1. Criar uma estrutura para armazenar as constantes de PID
2. Criar uma estrutura para armazenar os dados da pista (posição, aceleração e constantes de controle)
3. Implementar a estrutura de velocidades e seus respectivos controles de velocidade para cada tipo de curva:
   3.1. Curva fechada;
   3.2. Curva;
   3.3. Curva aberta;
   3.4. Reta.
4. Implementar a estrutura da pista:

4.1. Distância percorrida;

4.2. Aceleração (positiva ou negativa);

4.3. Estrutura de velocidade.

Definição das estruturas de dados

Em um arquivo de classe *structures.h* foram definidas as estruturas para armazenar as configurações de velocidade do robô e configurações da pista. Conforme código fonte 15:

```
#ifndef STRUCTURES_H
#define STRUCTURES_H

struct Setup {
  float speed;
  float kp;
  float ki;
  float kd;
};
struct Mark {
  float position; // distance in meters
  float acceleration; // mark acceleration
  Setup setup; // robot setup
};

#endif
```

Código Fonte 15 - Definição das estruturas. Fonte: Autoria própria

Estrutura de velocidades

Com as estruturas definidas, a instância de velocidades do robô para cada tipo de curva foi definida por meio do arquivo *settingsSpeed.h*. Dessa forma pode-se perceber que nas retas o robô consegue atingir velocidade máxima de 100% nos motores, isso implica em tensão máxima nos motores (9V), conforme código fonte 16:

```
Setup PD0[] {
  // Target Speed, kp, ki, kd
  {30, 27.0, 0.0, 2.1}, // Slow Curve
  {40, 20.0, 0.0, 3.5}, // Curve
  {70, 16.5, 0.0, 3.5}, // FastCurve
  {100, 15.0, 0.0, 4.1}, // Straight
};
```

Código Fonte 16 - Estruturas de velocidades. Fonte: Autoria própria

Estrutura de pista

Conforme figura 23 a instância da pista foi definida de acordo com os pontos críticos apresentados. Então de acordo com a distância percorrida pelo robô uma nova configuração de velocidades é feita fazendo com que o robô acelere e atinja as velocidades máximas nas retas. O código fonte 17 do arquivo *tracks.h* define a estrutura da pista.

```
Mark SMILE[] = { // Distance in meters, Aceleration, Constants of gain (PID)
  {00.78, +1.5, GAIN_NUMBER[Straight]},        //01
  {01.2, -2, GAIN_NUMBER[Curve]},        //02
  {02.60, +1.0, GAIN_NUMBER[Straight]},        //03
  {03.2, -1.0, GAIN_NUMBER[Curve]},        //04
  {04.6, +1.0, GAIN_NUMBER[Straight]},        //05
  {05.25, -1.0, GAIN_NUMBER[Curve]},        //06
  {06.0, +1.0, GAIN_NUMBER[Straight]},        //07
  {07.25, -1.0, GAIN_NUMBER[FastCurve]},        //08
  {08.5, +1.0, GAIN_NUMBER[Straight]},        //09
  {08.95, -1.0, GAIN_NUMBER[FastCurve]},        //10
  {09.6, +1.0, GAIN_NUMBER[Straight]},        //11
  {10.80, -1.0, GAIN_NUMBER[FastCurve]},        // 12
  {11.0, +1.0, GAIN_NUMBER[Straight]},        //13
  {11.50, -1.0, GAIN_NUMBER[Curve]},        //14
  {11.9, +1.0, GAIN_NUMBER[Straight]},        //15
  {12.60, -1, GAIN_NUMBER[Curve]},        //16
  {13.2, +1.00, GAIN_NUMBER[FastCurve]},        //17
  {FINAL_TARGET_POSITION, +1.5, GAIN_NUMBER[Straight]}
};
```

Código Fonte 17 - Estruturas de pista. Fonte: Autoria própria

Lógica para mudança de velocidades do robô

Para implementação da lógica de mudança de velocidades do robô foi implementado o algoritmo abaixo que a partir da posição atual do robô (dada pelos *encoders*) é feita uma verificação se o robô atingiu a posição de mudança de configurações de acordo com a estrutura de pista e nesse momento é feito a atualização do PID, de acordo com os sensores de linha. Para que não haja mudanças bruscas de velocidade, foi implementado também o algoritmo para fazer uma rampa linear de aceleração e por fim atualizar as velocidades dos motores.

```
if (currentPosition >= TargetMark.position && MAPPING_ENABLED)
    {
      currentMark++;
      // Get current Target Mark
      TargetMark = TRACK_EVENT_NAME[currentMark];
      acceleration = TargetMark.acceleration;
      // Update Robot Setup
      setupPID(TargetMark.setup);
    }

  LineReader.readCalibrated(sensorValues, QTR_EMITTERS_ON);
  linePosition = LineReader.readLine(sensorValues, QTR_EMITTERS_ON, WHITE_LINE) - 2500.0;

    directioncontrol.setProcessValue(linePosition);
    directiongain = directioncontrol.compute();

    // Speed update with acceleration
    if (ACCELERATION_ENABLED)
    {
      nowAccTimer = millis() / 1000.0;
      if (nowAccTimer - startAccTimer > ACCELERATION_INTERVAL)
      {
        // check if the robot accelerates or decelerates
        if ((acceleration > 0 && currentSpeed < targetSpeed) || (acceleration < 0 && currentSpeed > targetSpeed))
        {
          currentSpeed += acceleration; // * ACCELERATION_INTERVAL;
          currentSpeed = currentSpeed > targetSpeed ? targetSpeed : currentSpeed < 0 ? 0 : currentSpeed;
          // LOG.println(currentSpeed);
        }

        startAccTimer = nowAccTimer;
      }
    }
    else
    {
      currentSpeed = targetSpeed;
    }

    // Set the Direction
    leftmotorspeed = currentSpeed + (directiongain > 0 ? -directiongain : 0);
    righmotorspeed = currentSpeed + (directiongain < 0 ? +directiongain : 0);

    // Constrain the speed value to [0, 100] interval
    leftmotorspeed = leftmotorspeed > 100 ? 100 : leftmotorspeed < -REVERSE ? -REVERSE : leftmotorspeed;
    righmotorspeed = righmotorspeed > 100 ? 100 : righmotorspeed < -REVERSE ? -REVERSE : righmotorspeed;

    if (MOTORS_ENABLE)
    {
      LeftMotor.speed(leftmotorspeed);
      RightMotor.speed(righmotorspeed);
    }
  }
```

Código Fonte 18 - Lógica para mudança de configurações do robô. Fonte: Autoria própria

3.2.5.6. Lógica geral

Esta etapa consiste na implementação de todo o fluxo lógico do robô. Foram necessárias algumas implementações cruciais para que o robô possa ter o melhor rendimento na pista, realizar a partida e parada no local correto. Na introdução foi apresentado a pista e as regras da competição, dessa forma a lógica geral deve ser capaz de estar dentro dessas regras.

A lógica geral do robô consiste em seguir o fluxo de implementação representado pela figura 24: instanciamento das classes e bibliotecas, configuração dos encoders, ajuste do PID, sensores de linha, calibração dos sensores de linha, partida, seguidor de linha e chegada. De forma didática o fluxograma da figura 24 abaixo ilustra a lógica implementada no robô para seguir a linha, o código completo consta no Apêndice A.

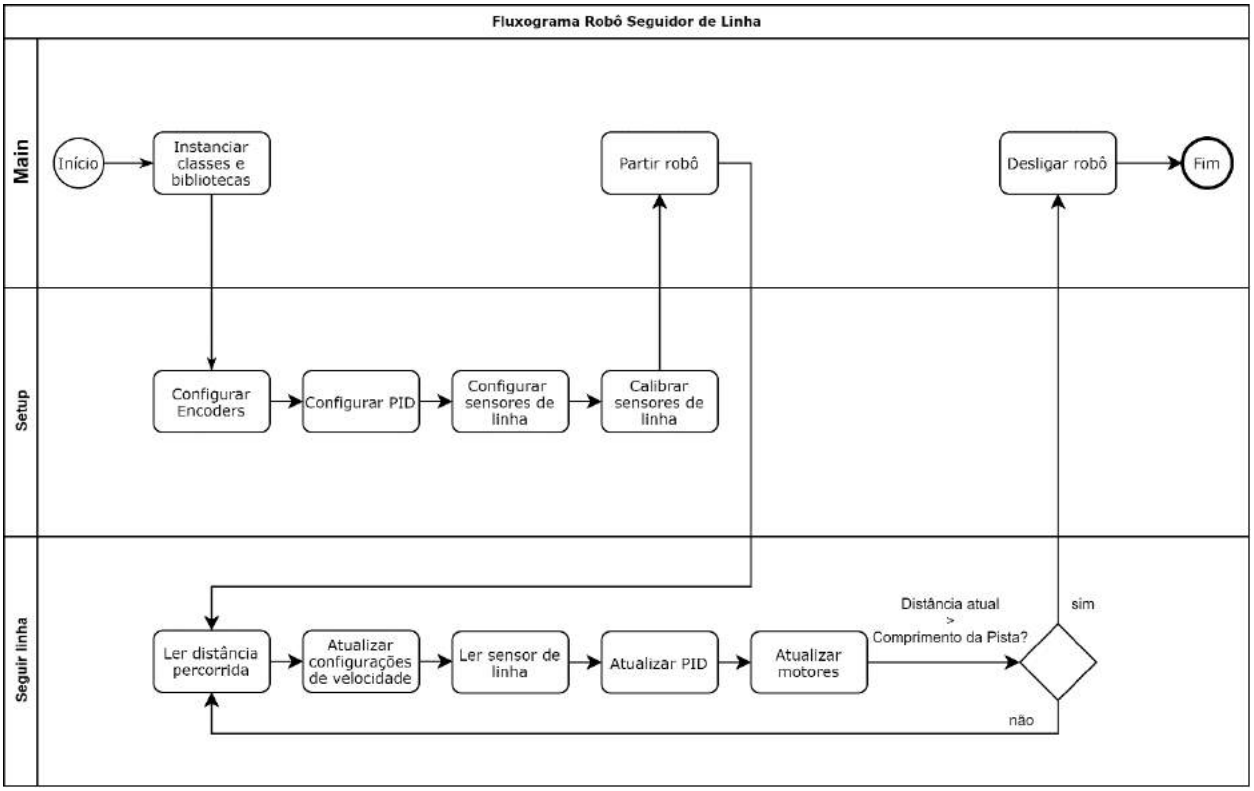


Figura 24 - Fluxograma - Lógica principal do robô. Fonte: Autoria própria

3.2.6. Etapa 6 - Montagem do protótipo

Material Utilizado

- *Microsoft* Visual Studio Code;
- *IDE PlatformIO;*
- *Autodesk Fusion 360;*
- *Circuit Maker;*
- Impressora 3D *Prusa i3*;
- Chaves de montagem.

Com os resultados obtidos em cada uma das etapas anteriores o protótipo foi montado seguindo os modelos especificados na modelagem do circuito elétrico e no modelo dos chassis.

Nesta etapa, primeiro o robô foi modelado integralmente no software *Fusion 360*, ou seja, todos os componentes foram adicionados ao modelo. Podemos visualizar o modelo teórico final na figura abaixo:

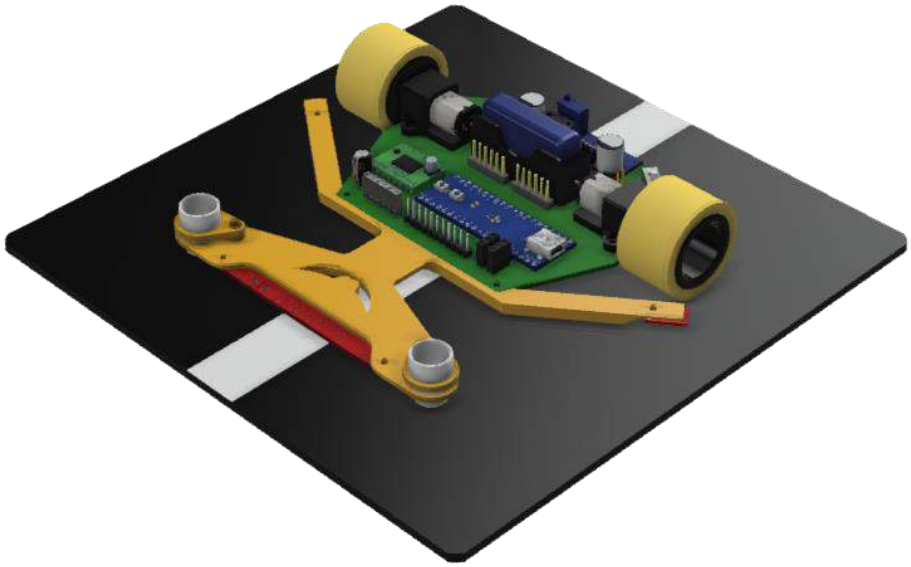


Figura 25 - Modelo 3D completo do robô seguidor de linha. Fonte: Autoria própria

Já o modelo físico, pode ser observado na figura abaixo:

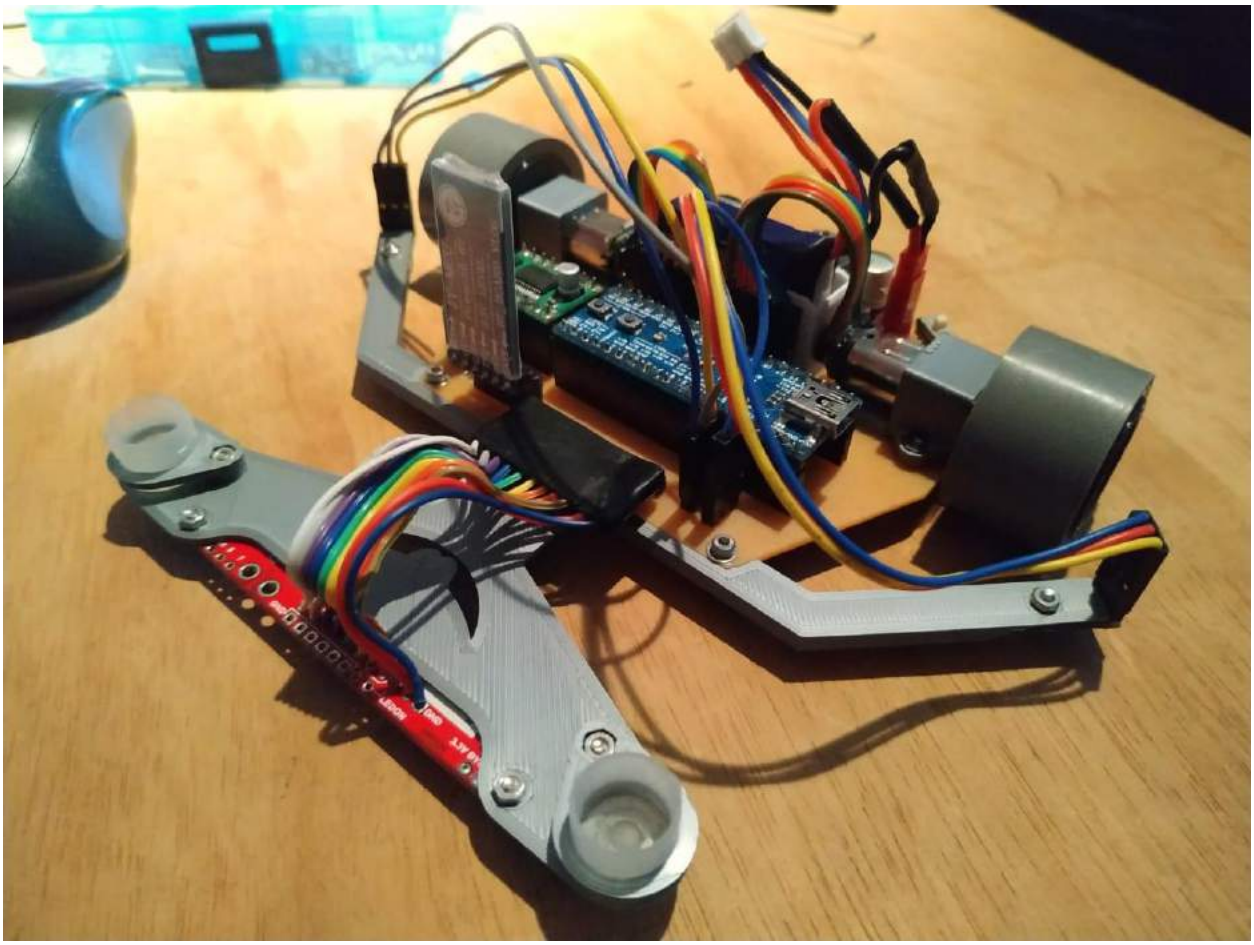


Figura 26 - Robô montado. Fonte: Autoria própria

3.2.7. Etapa 7 - Testes

Material Utilizado

- *Microsoft* Visual Studio Code;
- *IDE PlatformIO;*
- *Arduino IDE;*
- Chaves de montagem.

Através do Arduino IDE na aba do *Serial Monitor*, foi possível realizar os testes de comunicação com o robô via *Bluetooth* e desta forma validar os *logs* gerados pelo robô. Também foi montada uma pista de testes conforme a figura abaixo, para testar o sistema de controle do robô.

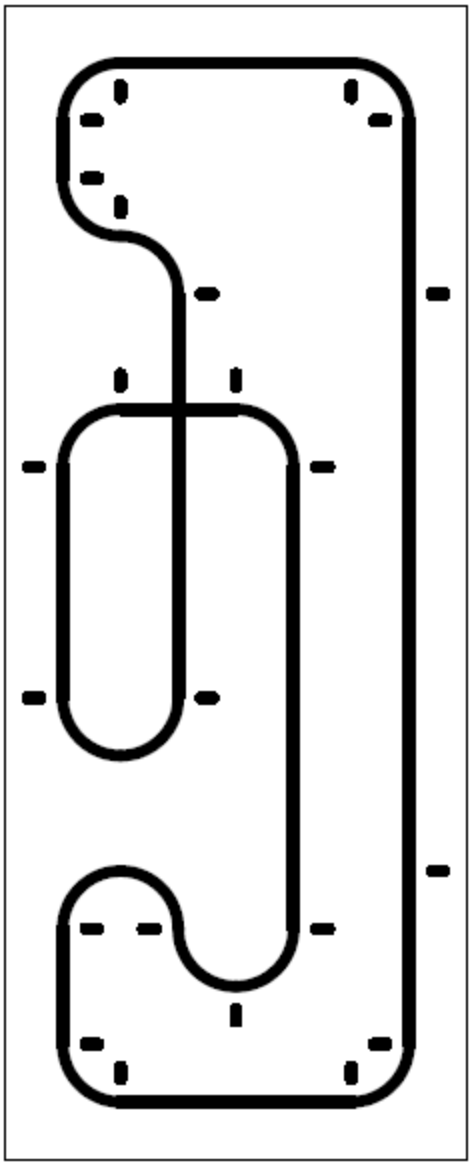


Figura 27 - Pista de testes. Fonte: Autoria própria.

3.3. Dinâmica

3.3.1. Modelagem matemática do robô

Com todos os componentes definidos, pode-se iniciar a modelagem matemática da planta a fim de obter um valor aproximado para as constantes de PID.

O robô é composto por dois motores de corrente contínua 6V (DC Motors) desenvolvidos pela *Pololu* com caixa de redução 30:1, de acordo com o *datasheet*, pode-se obter os parâmetros nominais do motor, representados na tabela 3 abaixo:

Tabela 3 - Parâmetros do motor (Pololu, 2019)

Parâmetros	Valor	Unidade
Velocidade vazio	1000	RPM
Corrente a vazio	0.07	A
Velocidade nominal	830	RPM
Torque vazio	1.0	kg.mm
Potência nominal	0.89	W
Potência máxima	1.5	W
Torque máximo	5.7	kg.mm
Corrente em torque bloqueado	1.6	A

Convertendo os valores para o sistema internacional, obtém-se a seguinte tabela:

Tabela 4 - Parâmetros do motor no SI. Adaptado (Pololu, 2019)

Parâmetros	Valor	Unidade
Velocidade vazio	104.72	rad/s
Velocidade nominal	86.92	rad/s
Torque vazio	9.80665	N.mm
Torque máximo	5.589	N.mm

Pelo projeto, o robô é acionado diretamente, sem reduções mecânicas. Sabe-se que a função de transferência para motores de corrente contínua é dada por:

$$\frac{\theta_m(s)}{E_a(s)} = \frac{K}{s(s+a)} \quad (1)$$

Sabendo a relação entre θ e ω:temos que:

$$\frac{\omega(s)}{E_a(s)} = \frac{K}{s+a} \quad (2)$$

Pelos dados fornecidos na tabela 3 obtido pelo fabricante temos que a velocidade angular do motor é de: 86,92 rad/s, logo a velocidade para o tempo de subida de 63,2% será de:

$$\omega_\tau = 86{,}92 * 0{,}632 = 54{,}93344\ rad/s$$

Para uma entrada degrau de 6V, a função no tempo será:

$$\omega(t) = 6\frac{C}{a}(1 - e^{-at}) \quad (3)$$

Logo, pode-se obter K da seguinte forma:

$$6\frac{C}{a} = K = 86{,}92 \ \therefore \mathrm{K} = 14{,}49$$

Localizando o valor de $\omega_\tau = 54{,}93344\frac{rad}{s}$ na figura abaixo, temos que $\tau = 1.67\ s$.

Logo $a = \frac{1}{1.67} = 0.6\ s^{-1}$

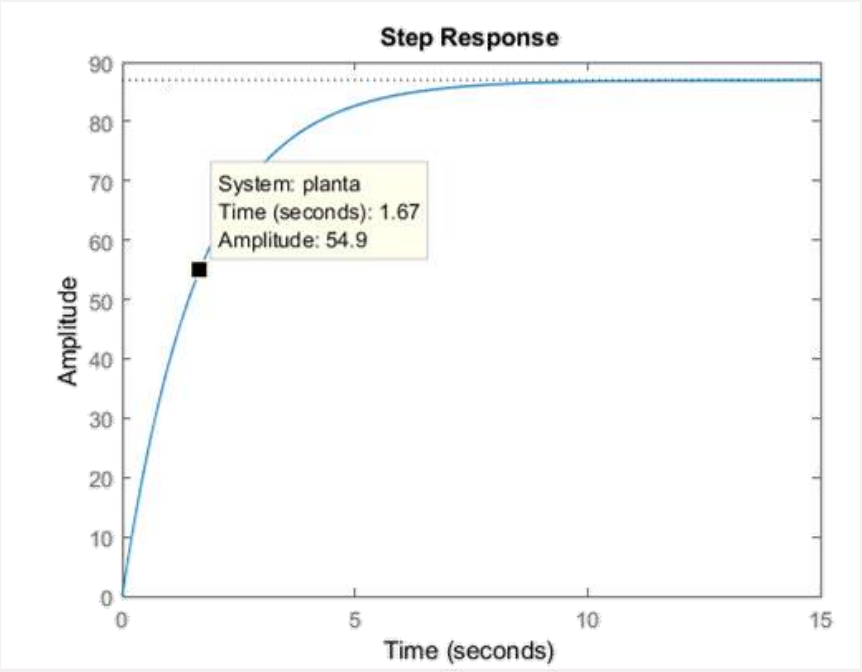


Figura 28 - Constante de tempo do motor. Fonte: Autoria própria

Desta forma com o valor de $a = 0.6\ s^{-1}$, pode-se obter o valor da constante C, da seguinte forma:

$$\frac{C}{a} = \frac{86.92}{6} = 14.49\ \therefore C = 8.675$$

Portanto, função angular no tempo pode ser expressa como:

$$\omega(t)\ =\ 6\frac{8.675}{0.6}(1 - e^{-0.6t}) \tag{4}$$

Já as funções no domínio s, definidas por:

$$\frac{\omega_m(s)}{E_a(s)}\ =\ \frac{8.675}{s+0.6} \tag{5}$$

$$\frac{\theta(s)}{E_a(s)}\ =\ \frac{8.675}{s(s+0.6)} \tag{6}$$

3.3.2. Validação do modelo matemático

Para validar o modelo matemática obtido anteriores, aplica-se uma entrada 6V na função no tempo, ilustrada na figura abaixo:

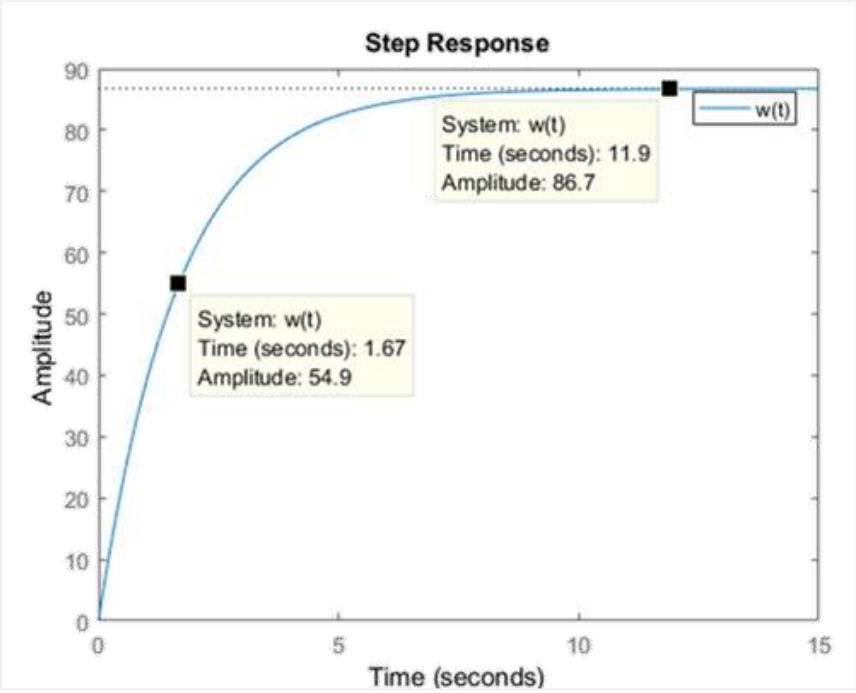


Figura 29 - Resposta do motor no tempo para uma entrada de 6V. Fonte: Autoria própria

Logo como a velocidade de regime permanente está muito próxima com o valor esperado de 86,92 rad/s, da tabela 4, constata-se que o modelo é adequado.

3.3.3. Projeto do controlador PD

A determinação dos pólos podem ser determinadas, se igualando o denominador a zero. Assim o lugar geométrico das raízes da planta em malha aberta é representado pela figura abaixo:

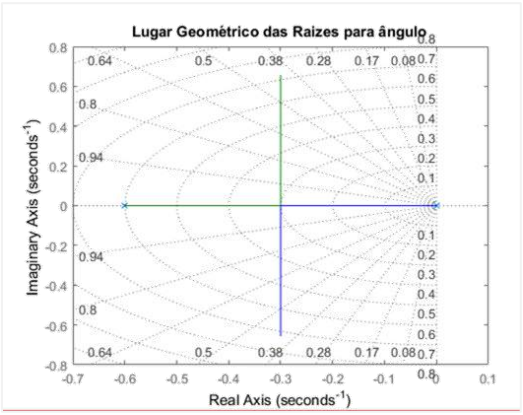


Figura 30 - Lugar geométrico das raízes do modelo. Fonte: Autoria própria

O LGR da função de transferência em malha aberta (Equação 6) implica em uma função que cresce infinitamente, ou seja, é instável. Para evitar essa instabilidade, define-se o sistema em malha fechada com realimentação unitária, portanto a função de transferência obtida em malha fecha é dada por:

$$\frac{\theta(s)}{E_a(s)} = \frac{8.675}{s^2 + 0.6s + 8.675} \quad (7)$$

Empiricamente se percebeu que esta aplicação não necessita de um controle PID, pois a parte integral é acumular erros no robô. Então para se o controle do tipo Proporcional-Derivativo (PD) é o mais adequado ao sistema, pois a parte derivativa faz a correção do erro antecipadamente e tende a aumentar a estabilidade do sistema em um todo, fazendo com que o tempo de resposta diminui, mas sem afetar diretamente o erro estacionário, que pode ser definido como:

$$e(t + T_d) \simeq e(t) + T_d \frac{d\,e(t)}{dt} \quad (8)$$

Onde a saída será dada por: $u(t) = K(e(t) + T_d \frac{d\,e(t)}{dt})$ e pela aproximação acima, pode-se obter: $u(t) \simeq K\,e(t + T_d)$, ou seja, o sinal de controle é proporcional a T_d unidades de tempo à frente. Portanto a função de transferência que define o controlador pode ser escrita como:

$$C(s) = Kp + Kd\,s = Kd(s + \frac{Kp}{Kd}) \quad (9)$$

Ainda pode-se simplificar a equação acima como:

$$C(s) = Kc(s + z) \quad (10)$$

3.3.4. Obtenção dos ganhos do controlador

De acordo com a equação 6 e comparando com a função de transferência em sua forma natural, representada pela equação 11, abaixo:

$$\frac{\theta_m(s)}{E_a(s)} = \frac{{\omega_n}^2}{s^2 + 2\xi\omega_n s + {\omega_n}^2)} \quad (11)$$

Pode-se obter os valores do sistema de frequência natural e amortecimento, que serão dados por:

$$\xi = 0.102 \quad \omega_n = 2.95\, rad/s$$

Pela análise da resposta, o tempo de estabelecimento e tempo de subida podem ser obtidos, respectivamente:

$$t_s = \frac{4.6}{\xi \omega n} = 15.36s \quad t_r = \frac{1.8}{\omega n} = 0.6s \tag{12}$$

Aplicando uma resposta degrau ao modelo acima, pode-se obter o gráfico abaixo que ilustra uma resposta oscilatória e com alto tempo de estabelecimento.

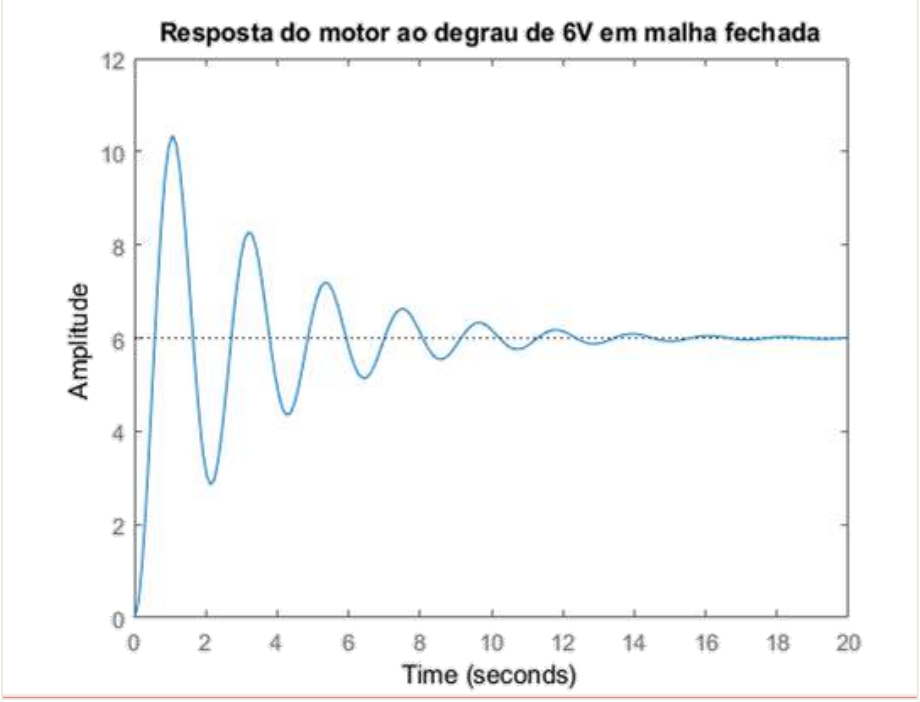


Figura 31 - Resposta do motor em malha fechada. Fonte: Autoria própria

Visando melhorar esta resposta com um sistema de controle PD, considerando, um *overshoot* de 5% e tempo de estabelecimento de 1.5 s, obtém-se os novos valores de frequência natural e tempo de estabelecimento que:

$$\xi = \sqrt{\frac{ln(PO)^2}{ln(PO)^2 + \pi^2}} \therefore \xi = 0.69 \tag{13}$$

$$\omega_n = \frac{4.6}{\xi * ts} \therefore \omega_n = 4.44 \tag{14}$$

Aplicando os dados no algoritmo PD no MATLAB de acordo com a figura abaixo:

```
%% Dados de Entrada
opt = stepDataOptions('InputOffset',0,'StepAmplitude',6);
zetamin=0.69; % fator de amortecimento
wmin=4.44; % frequência natural
num = 8.675;
den = [1 0.6 8.675];
%% Cáculo do polos dominantes em malha fechada
wd=wmin*sqrt(1-(zetamin^2));
p1=-zetamin*wmin+wd*i;
p2=-zetamin*wmin-wd*i;
%% calculo dos parâmetros do controlador
teta=180-atan(wd/(zetamin*wmin))*180/pi;
phi=2*teta-180;
z=-wd/(tan((180-phi)*pi/180))+zetamin*wmin;
kt=(wd^2+1)/(sqrt(wd^2+(z-1)^2));
if (size(num,2)==1)
kc=kt/num;

else
kc=kt;
end
% Visualizando a respota no tempo
[numCL,denCL]=cloop(num,den);
step(numCL,denCL)
hold on
[numCL,denCL]=cloop(conv(num,[kc z*kc]),den);
step(numCL,denCL)
legend('Sistema normal','Sistema controlado por PD')
title('Sistema Controlado por PD modelado aplicado ao degrau de 6V')
```

Figura 32 - Algoritmo para cálculo dos parâmetros de controle. Fonte: Autoria própria

O resultado obtido pelo algoritmo mostra que o sistema responde rapidamente a resposta do degrau e se torna estável, porém com um certo erro estacionário, porém aceitável para o robô.

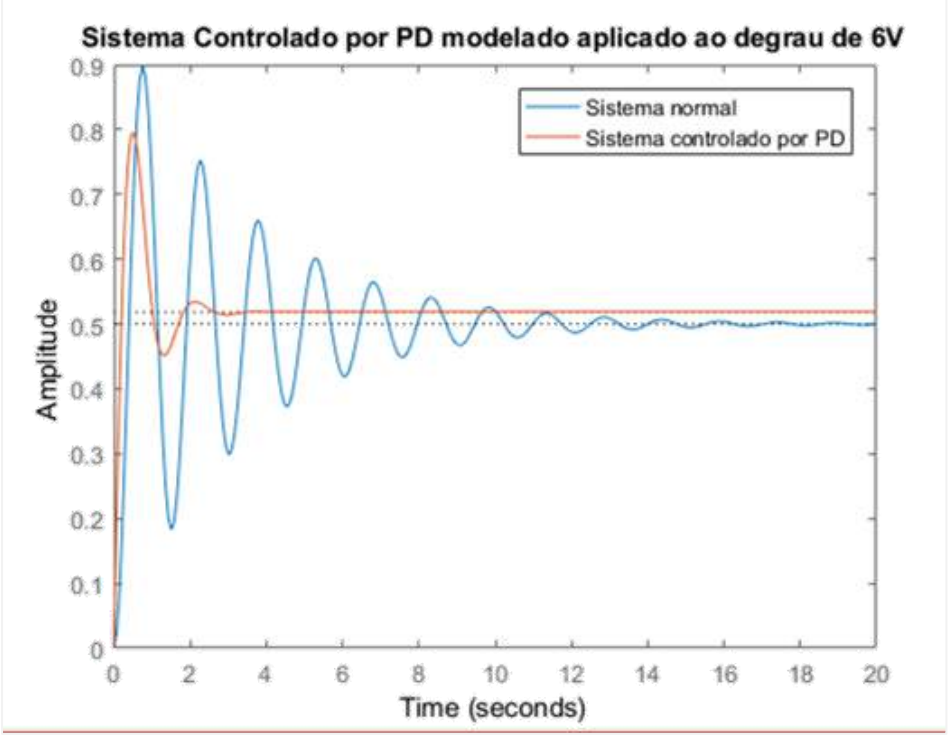


Figura 33 - Resposta do motor com e sem controle. Fonte: Autoria própria

De acordo função de transferência de controle (Equação 9), os parâmetros obtidos foram os seguintes:

$$Kc = 0.03344$$

$$z = 3.2174$$

Logo a função de transferência será dada por:

$$C(s) = 0.03344(s + 3.2174) \quad (15)$$

Para se obter os parâmetros de controle *Kp* e *Kd*, pode-se usar a relação dada pela equação abaixo:

$$C(s) = Kp + Kd * s = Kc\,(s + z) \quad (16)$$

$$Kc = Kd \qquad z = \frac{Kp}{Kd} \quad (17)$$

Implicando que as constantes proporcional e derivativa serão respectivamente:

$$Kp = 0{,}107589 \qquad \text{e} \qquad Kd = 0{,}03344$$

4. RESULTADOS E DISCUSSÕES

4.1. Desenvolvimento proposto

Como resultado do desenvolvimento proposto foi possível desenvolver completamente um robô seguidor de linha 100% autônomo, com sistema de controle PD implementado e uma ótima relação massa x velocidade. Pode-se observar na figura abaixo o robô final em uma balança de precisão medindo 135 gramas, o que é um resultado melhor do que cálculo teórico de 183 gramas.

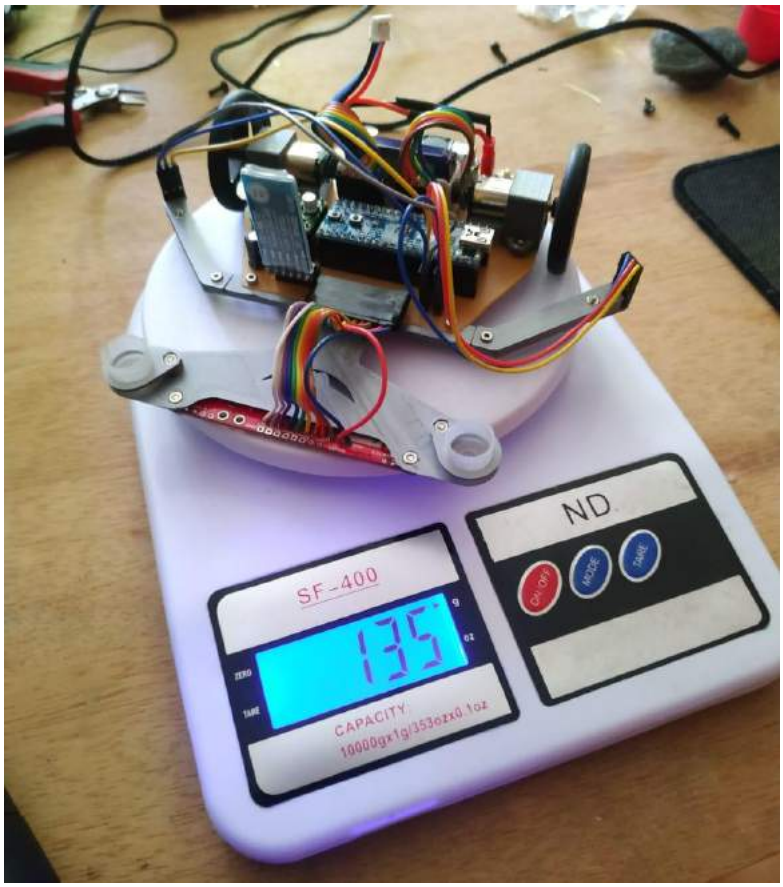


Figura 34 - Massa real do robô. Fonte: Autoria própria

4.2. Modelagem 3D e montagem mecânica

A montagem mecânica do robô foi bem sucedida, sendo muito fiel ao que foi apresentado na metodologia e no modelo 3D. Mostrando a importância de um bom modelo 3D, pois dessa forma o retrabalho com montagem e custo de material foi minimizado.

4.3. Circuito elétrico

Como resultado para o funcionamento do circuito elétrico do robô, foi observado um comportamento coerente com o esperado, sem nenhum superaquecimento e nenhum dano aos componentes elétricos. Na figura abaixo, tem-se o robô ligado.

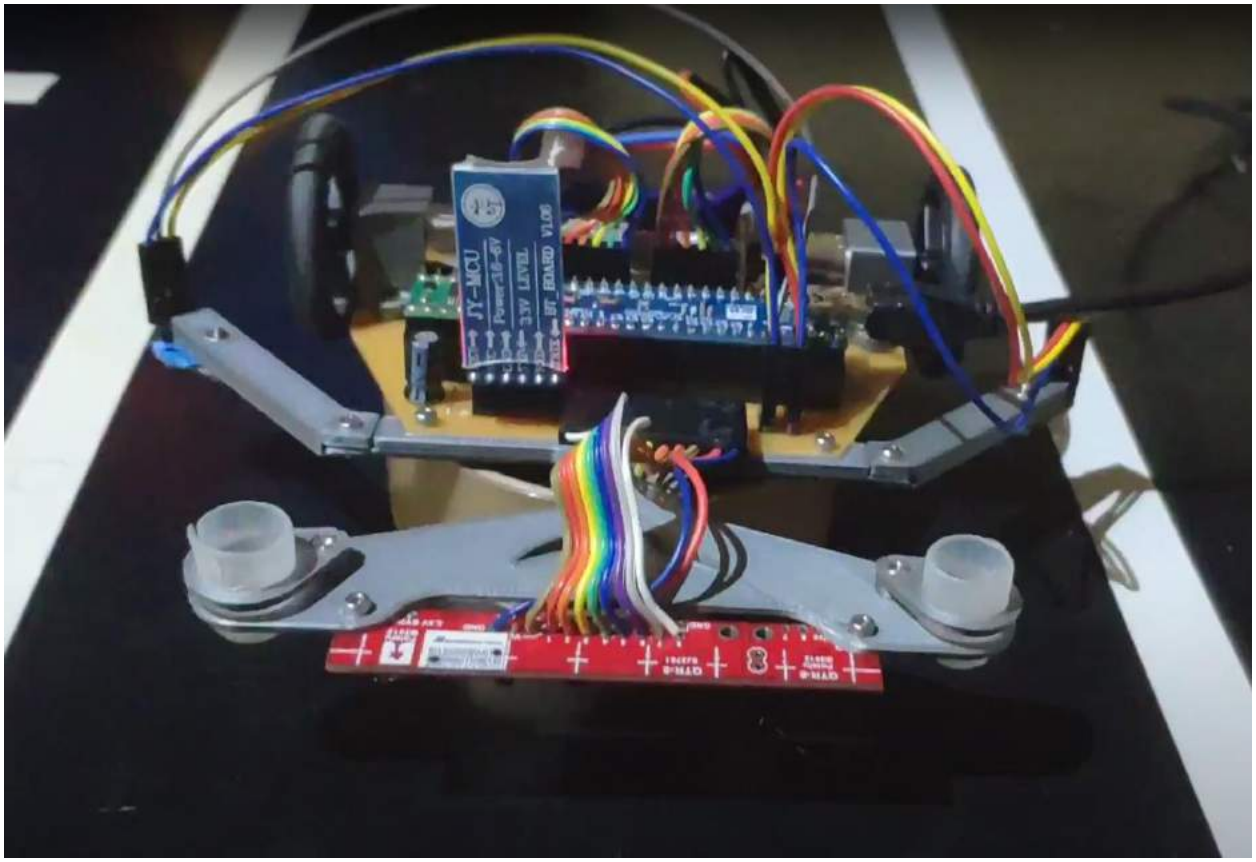


Figura 35 - Robô energizado. Fonte: Autoria própria

4.4. Sistema de controle

Foi possível desenvolver a modelagem dinâmica do motor para um controle PD em busca das constantes e ganhos do controle. Pela simulação no *Matlab* pode-se perceber que o sistema de controle PD implementado funcionava corretamente. Na figura abaixo a curva em azul mostra a resposta do motor sem controle, que é bem oscilatória e com tempo de estabelecimento bem alto. Já a curva em vermelho, mostra o motor com a aplicação da malha fechada de controle, com um tempo de estabelecimento conforme o definido na seção de cálculos do controlador.

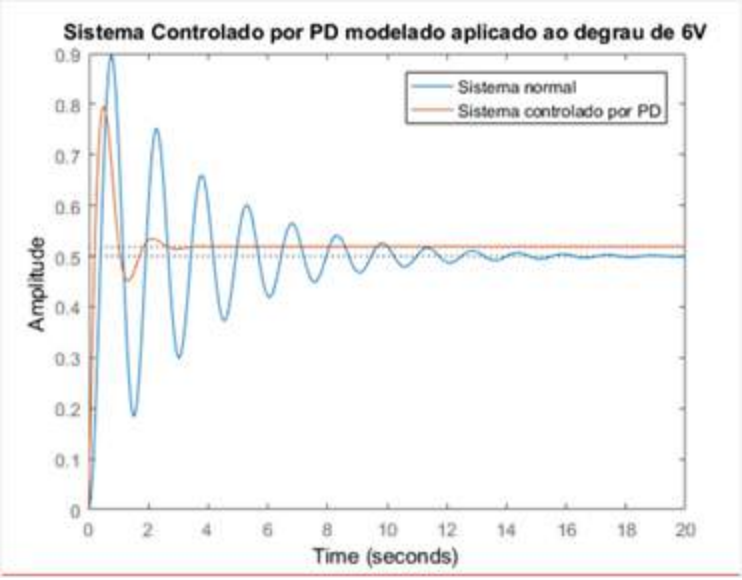


Figura 36 - Resposta do motor com e sem controle. Fonte: Autoria própria

Como característica do controle PD, a resposta controlada ficou com um erro estacionário, porém bem próximo a convergência do sistema e que não impacta no desenvolvimento.

Já a implementação do sistema de controle PID de forma prática, foi a etapa mais crítica do desenvolvimento do robô no qual exigiu muitas interações e testes empíricos na pista de testes. O referencial teórico estudado possibilitou um melhor desempenho no ensaio, pois o ponto de partida para refinamento dos valores para as constantes não era zero, ou seja, tinha-se um direcional para escolha dos ganhos.

Como resultado de muitas interações, foi possível obter uma excelente resposta do robô nas retas, implicando em um baixo tempo de estabelecimento, com um pequeno erro estacionário.

Na figura abaixo, é possível ver o erro do robô que varia de -2500 a 2500, medido pelo valor obtido pelos sensores de linha ao longo do tempo. Nesse ensaio pode-se perceber que na reta inicial o robô estava perfeito, com erro zero e posteriormente foi acumulando erros devido às curvas. No apêndice B, consta um link para o vídeo do robô em funcionamento.

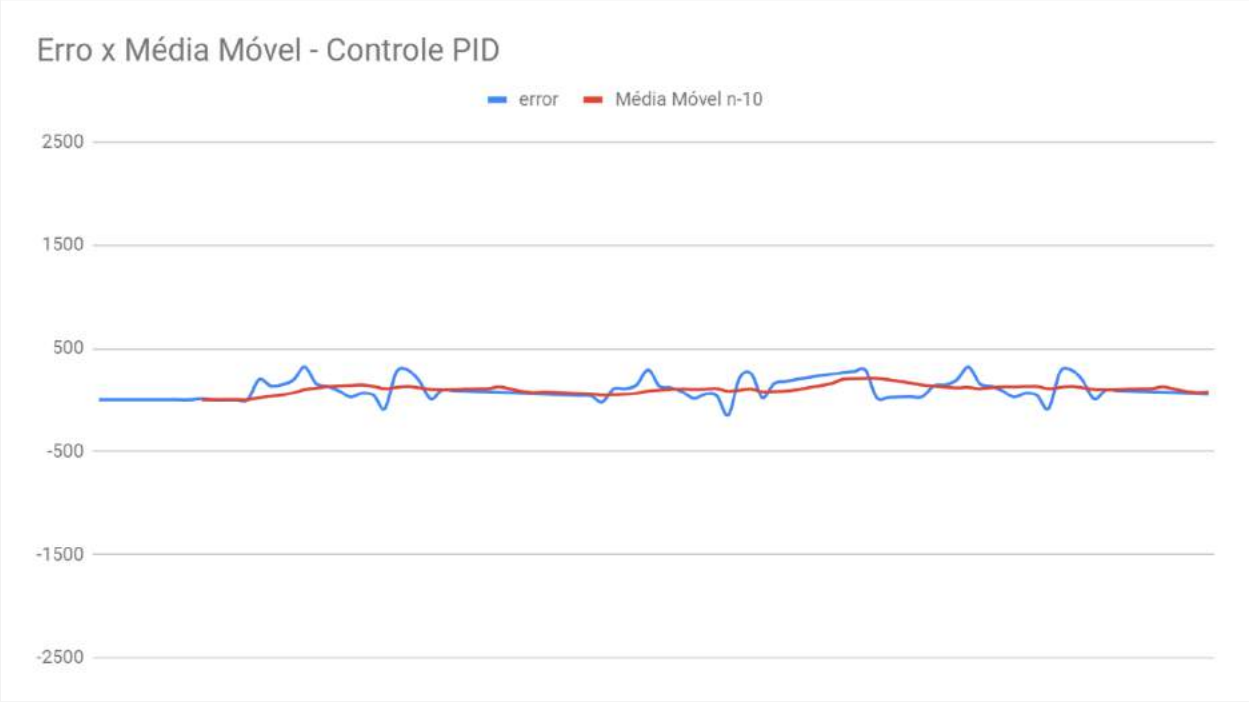


Figura 37 - Gráfico de erro do robô na pista. Fonte: Autoria própria

4.5. Premiações em competições de robótica

Durante o desenvolvimento do projeto, o robô foi nomeado de Sloth, nome sugerido devido à tradução do termo “preguiça” (animal) em português, e participou 4 competições de robótica: 2 *Winter Challenge*, *Smiles* e *Iron Cup*. Nessas participações o robô se provou muito competitivo e acabou conquistando 2 troféus, demonstrados nas duas figuras abaixo.

Figura 38 - Primeiro lugar na competição Smiles. Fonte: Autoria própria

Figura 39 - Terceiro lugar na competição Iron *Cup*. Fonte: Autoria própria

Estas premiações demonstram o valor agregado construído pelo projeto, para conquistar o primeiro lugar na competição paulista (*Smiles*) e o terceiro lugar na competição nacional (*Iron Cup*).

5. CONCLUSÕES E PERSPECTIVAS

5.1. Conclusões finais

Este projeto foi capaz de abordar a interdisciplinaridade de conhecimentos para construção e desenvolvimento de um robô, desde fundamentos de desenho, modelagem 3D, eletrônica, dinâmica, sistemas de controle e programação. Uma abordagem prática de todos os conhecimentos vivenciados durante a graduação com um pouco de desafios técnicos que um ambiente competitivo proporciona.

A partir dos estudos com esse trabalho pode-se perceber o avanço computacional e tecnológico do século XXI, um robô com um *software* capaz de realizar leituras de sensores e processar todos os dados em microssegundos é resultado de todo esse avanço, mostrando o quão poderoso e barato está processar dados hoje em dia.

A solução proposta de um seguidor de linha é de contribuir para outras finalidades, como por exemplo, automatização dos processos de logística. Pois com essa tecnologia de locomoção podemos alocar trabalhadores que transportam materiais de um lado para o outro de um centro logístico para assumir atividades de maior valor agregado e analíticas.

Utilizar toda a metodologia proposta se mostrou eficiente durante o desenvolvimento, sendo possível minimizar o custo e tempo com retrabalhos. Ressalta-se a importância da modelagem 3D e modelagem do circuito elétrico, que permitem realizar simulações para entender o comportamento do robô, sem a necessidade de testes práticos.

Os resultados obtidos foram promissores e este robô foi o primeiro robô nacional a aplicar de forma eficiente o mapeamento de pistas, um diferencial competitivo perante os demais competidores. Foi possível também implementar uma malha de controle fechada, com realimentação do *encoder*, também um diferencial que acabou resultando nas premiações.

Mais do que competir em busca de troféus, o projeto permitiu que toda comunidade de robótica tivesse acesso a esse novo conceito de seguidor de linha, pois toda programação, esquemas elétricos e modelagens estão disponíveis de forma *open-source* por meio do *GitHub*.

5.2. Perspectivas

Dado as dificuldades encontradas para se otimizar o sistema de controle PID, como perspectivas de trabalhos futuros, a intenção é construir um software que seja capaz de estimar os valores das constantes de controle proporcional, integral e derivativo baseado nos parâmetros do robô. Dessa forma espera-se poder atingir resultados melhores e mais rápidos na hora de otimizar o sistema de controle. Na imagem abaixo, um *Wireframe* ilustra como a plataforma funcionaria.

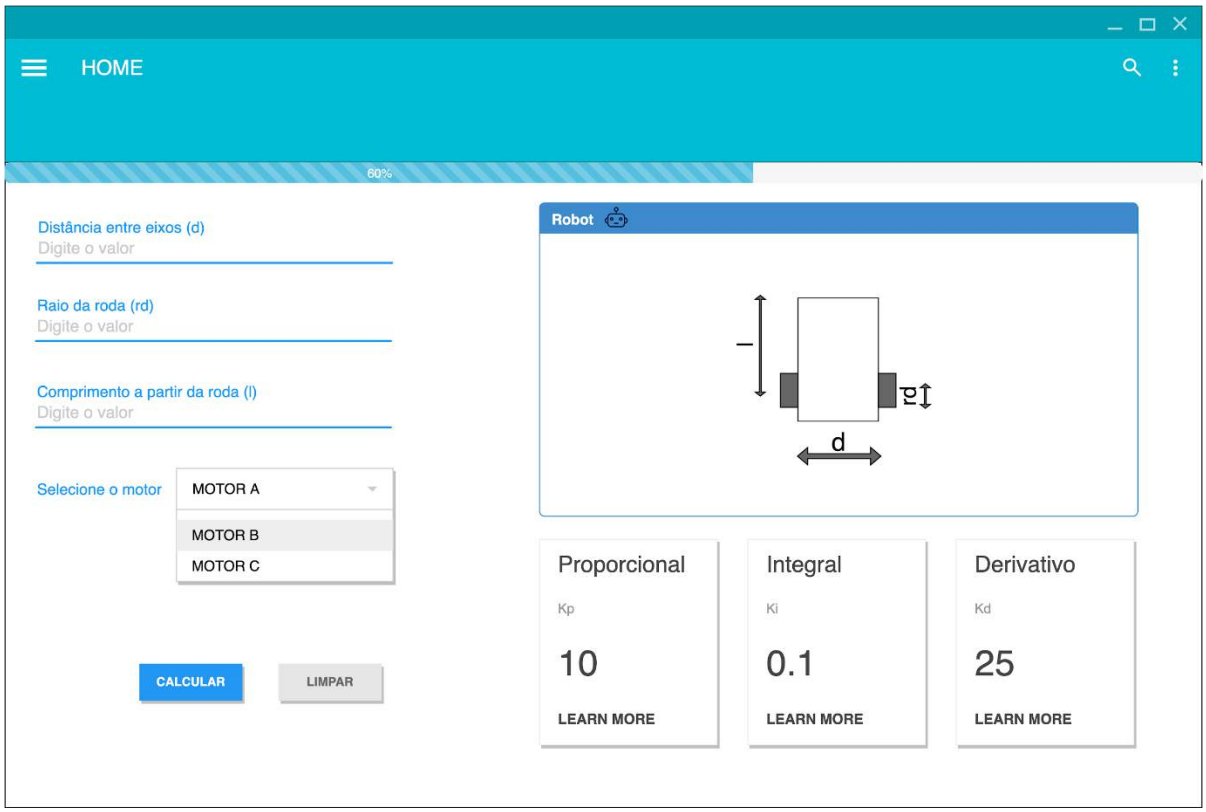


Figura 40 - Wireframe plataforma de controle. Fonte: Autoria própria

Primeiro o usuário insere quais são os parâmetros físicos do robô, ou seja, distância entre eixos, raio da roda e comprimento do robô.

Posteriormente seleciona qual o motor que está sendo utilizado, com base em uma lista de motores pré-existentes e clica no botão de cálculo.

A plataforma realiza os cálculos teóricos e exibe os valores para as constantes: proporcional, integral e derivativo do sistema de controle PID.

Complementando para trabalhos em indústria, pode-se também aprimorar o robô com um módulo de comunicação Wi-Fi, como por exemplo, ESP8266 NodeMCU para aplicar os conceitos de Internet das Coisas e fomentar mais tecnologias para era da Indústria 4.0.

Figura 41 - ESP8266 NodeMCU. (ROBOTDYN)

REFERÊNCIAS BIBLIOGRÁFICAS

BORGES, G. A., DEEP, G. S. e LIMA, A. M. N. 2003. *Controladores cinemáticos de trajetória para robôs móveis com tração diferencial.* s.l. : VI Simpósio Brasileiro de Automação Inteligente, 2003.

CAMPOS, Lorraine Vilela. Campeonatos de robôs. *Brasil Escola.* [Online] [Citado em: 22 de julho de 2019.] https://brasilescola.uol.com.br/curiosidades/campeonatos-robos.htm.

Figueiredo, L. C., & Jota, F. G. 2004. *Introdução ao controle de sistemas não-holonômicos. .* Campinas : Sba Controle & Automação vol.15 no.3, 2004.

KIAT Beng, Ng. 2009. *Workshop in Micromouse.* Taiwan : s.n., 2009.

NISE, N. S. 2012. *Engenharia de Sistemas de Controle.* s.l. : Sixth. [S.l.]: LTC, 2012.

Pololu. 2019. Micro Metal Gearmotors Datasheet. *Pololu.* [Online] Pololu, Dezembro de 2019. https://www.pololu.com/file/0J1487/pololu-micro-metal-gearmotors.pdf.

—. 2019. PololuQTRSensors. *Github.* [Online] Pololu, 2019. https://github.com/pololu/qtr-sensors-arduino.

Robocore. 2016. robocore__regras_seguidor_de_linha. *Robocore.* [Online] Junho de 2016. https://www.robocore.net/upload/attachments/robocore__regras_seguidor_de_linha_108.pdf .

ROBOTDYN. Wifi NodeM ESP8266. *ROBOTDYN.* [Online] https://robotdyn.com/wifi-nodem-esp8266-32m-flash-ch340g.html.

Vieira, Frederico. 2005. *Controle Dinâmico de robôs móveis com acionamento diferencial.* Universidade Federal do Rio Grande do Norte : s.n., 2005.

APÊNDICE A – CÓDIGO PRINCIPAL DO ROBÔ

```
#include "Motor.h"
#include "_config.h"
#include "QEI.h"
#include "PID.h"
#include "QTRSensors.h"
#include "tracks.h"

byte rcvd=0;

Motor LeftMotor(PIN_M1_PWM, PIN_M1_IN1, PIN_M1_IN2);
Motor RightMotor(PIN_M2_PWM, PIN_M2_IN1, PIN_M2_IN2);
float leftmotorspeed = 0;
float righmotorspeed = 0;
float currentSpeed = 0;
float targetSpeed = 0;
float acceleration = 5;
bool MOTORS_ENABLE = true;

QEI LeftEncoder(PIN_ENC1_A, PIN_ENC1_B, PULSES_PER_REV, WHEEL_RADIUS);
QEI RightEncoder(PIN_ENC2_A, PIN_ENC2_B, PULSES_PER_REV, WHEEL_RADIUS);
float currentPosition; // Robot current position in the track
float leftDistance; // left distance by encoder (m)
float rightDistance; // right distance by encoder (m)
bool firstMarkDone = false;

unsigned char pinsLineReader[NUM_SENSORS] = {
  PIN_LR_S8,
  PIN_LR_S7,
  PIN_LR_S6,
  PIN_LR_S5,
  PIN_LR_S4,
  PIN_LR_S3,
  PIN_LR_S2,
  PIN_LR_S1
  };

unsigned int sensorValues[NUM_SENSORS];

QTRSensorsAnalog LineReader(pinsLineReader, NUM_SENSORS, NUM_SAMPLES_PER_SENSOR, EMITTER
_PIN);

int linePosition;

// Counters of left and Right sensors
int checkpoint_left_counter = 0;
int checkpoint_right_counter = 0;
```

```
int last_checkpoint_left_counter = 0;
int last_checkpoint_right_counter = 0;
int crossroad_counter = 0;

bool robotstate = true;
bool robotplay = false;

//PID
int directiongain = 0;
PID directioncontrol(0, 0, 0);

// The target mark
Mark TargetMark;
int currentMark = 0;

// Timers
 float startLogTimer; // Debug the loop
 float startLapTimer; // Time of a lap
 float startAccTimer; // Acceleration interval
 float startCps_left_led_timer; // Acceleration interval
 float startCps_right_led_timer; // Acceleration interval

 float nowLogTimer; // Debug the loop
 float nowLapTimer; // Time of a lap
 float nowAccTimer; // Acceleration interval
 float nowCps_left_led_timer; // Acceleration interval
 float nowCps_right_led_timer; // Acceleration interval

// Interrupt when Mark Left was change
void checkpointSensorLeftCallback() {
   checkpoint_left_counter++;
   // cps_left_led_timer.start();
   digitalWrite(PIN_LED,1);
}

//Interrupt when Mark Right was change
void checkpointSensorRightCallback() {
   checkpoint_right_counter++;
   // cps_right_led_timer.start();
   digitalWrite(PIN_LED,1);
}

void setupPID(Setup setup){
  // Setup PID and Speed
  targetSpeed = setup.speed;
  directioncontrol.setTunings(setup.kp/CONSTANTE, setup.ki/CONSTANTE, setup.kd/CONSTANTE);
  // If line position is 0, the robot is just over the line
  directioncontrol.setSetPoint(0);
```

```
}

void setupMarkSensors(){
  attachInterrupt(digitalPinToInterrupt(PIN_TRACK_MARKING_RIGHT),checkpointSensorRightCallback, F
ALLING);
  attachInterrupt(digitalPinToInterrupt(PIN_TRACK_MARKING_LEFT), checkpointSensorLeftCallback, FALL
ING);
}

void setupRobot(){

  // Release motors for make more easy the calibration
  LeftMotor.coast();
  RightMotor.coast();

  // Calibrate the line sensor
  setupLineReader();

  // Reset Encoders
  LeftEncoder.reset();
  RightEncoder.reset();

  // Clear mark sensors counters
  checkpoint_left_counter = 0;
  checkpoint_right_counter = 0;

  //threads

  // Set the first setup of the Robot
  if(MAPPING_ENABLED){
    // Get first target mark - 0
    TargetMark = TRACK_EVENT_NAME[currentMark];
    acceleration = TargetMark.acceleration;
    // Update Robot Setup
    setupPID(TargetMark.setup);
  }else {
    // Get first target mark - 0
    TargetMark = TRACK_EVENT_NAME[currentMark];
    acceleration = TargetMark.acceleration;
    // Update Robot Setup
    setupPID(Normal);
  }
}

void setup(){
  delay(1000);
  PC.begin(PC_SPEED);
  BT.begin(BT_SPEED);
```

```
  LOG.println(PROJECT_NAME);
  LOG.println(PROJECT_BOARD);
  LOG.println(PROJECT_VERSION);
  pinMode(PIN_LED, OUTPUT);
  setupLeftEncoder();
  setupRightEncoder();
  setupLineReader();
  setupPID(Normal);
  setupRobot();
  setupMarkSensors();
}

void setupLeftEncoder() {
  //X2 encoding uses interrupts on only channel A.
  attachInterrupt(digitalPinToInterrupt(PIN_ENC1_A), leftEncoder, CHANGE);

  //X4 encoding uses interrupts on      channel A,
  //and on channel B.
  attachInterrupt(digitalPinToInterrupt(PIN_ENC1_B), leftEncoder, CHANGE);
}

void rightEncoder() {
  RightEncoder.encode();
}

void setupRightEncoder() {
  //X2 encoding uses interrupts on only channel A.
  attachInterrupt(digitalPinToInterrupt(PIN_ENC2_A), rightEncoder, CHANGE);

  //X4 encoding uses interrupts on      channel A,
  //and on channel B.
  attachInterrupt(digitalPinToInterrupt(PIN_ENC2_B), rightEncoder, CHANGE);
}

//Line Sensor
void setupLineReader() {
  delay(1000);
  LOG.print("Calibrating sensors...");
  digitalWrite(PIN_LED, 1);
  for (int i = 0; i < 1000; i++)
    LineReader.calibrate(true);
  LOG.println("Done.");
  delay(100);
  digitalWrite(PIN_LED, 0);
  delay(2500);
}

void manualTrackMapping(){
```

```
  // Manual Track Mapping
   // LOG.printf("%.2f,", LapTimer.read());
   // LOG.printf("%i,", currentMark);
   // LOG.printf("%i", linePosition);

   // LOG.printf("%.4f,", currentPosition);
   // LOG.printf("%.4f", DIF(leftDistance, rightDistance));
   // manualTrackMapping();
   // START
   // Checkpoint sensors mapping
   // Crossroad
   if (checkpoint_left_counter != last_checkpoint_left_counter && checkpoint_right_counter != last_chec
kpoint_right_counter) {
     crossroad_counter++;
     LOG.print("C,");
     LOG.print(crossroad_counter);
     LOG.print(",");
     LOG.print(leftDistance);
     LOG.print(",");
     LOG.println(rightDistance);
     checkpoint_left_counter--;
     checkpoint_right_counter--;
   }
   // Curve start/end marks
   else if (checkpoint_left_counter != last_checkpoint_left_counter && checkpoint_right_counter == last
_checkpoint_right_counter) {
     LOG.print("L,");
     LOG.print(checkpoint_left_counter);
     LOG.print(",");
     LOG.print(leftDistance);
     LOG.print(",");
     LOG.println(rightDistance);
     last_checkpoint_left_counter = checkpoint_left_counter;
   }
   // Start/Finish marks
   else if (checkpoint_left_counter == last_checkpoint_left_counter && checkpoint_right_counter != last
_checkpoint_right_counter) {
     LOG.print("R,");
     LOG.print(checkpoint_right_counter);
     LOG.print(",");
     LOG.print(leftDistance);
     LOG.print(",");
     LOG.println(rightDistance);
     last_checkpoint_right_counter = checkpoint_right_counter;
   }
   else {
     LOG.print("-,-,");
   }
```

```
}

void btcallback() {
  rcvd = 0;
  if(BT.available()>0){
    rcvd = (char) BT.read();
  }
  // BT.println("TEST BT - CALLBACK");
  switch (rcvd) {
    case 'A':
      kpdir += 0.5/3;
      break;
    case 'B':
      kpdir -= 0.5/3;
      break;
    case 'C':
      kidir += 0.05/3;
      break;
    case 'D':
      kidir -= 0.05/3;
      break;
    case 'E':
      kddir += 0.05/3;
      break;
    case 'F':
      kddir -= 0.05/3;
      break;
    case 'u':
      speedbase += 5.0/3;
      acceleration = acceleration > 0 ? acceleration : -acceleration;
      break;
    case 'd':
      speedbase -= 5.0/3;
      acceleration = acceleration > 0 ? -acceleration : acceleration;
      break;
    case 'G':
      robotstate = true;
      robotplay = true;
      LeftEncoder.reset();
      RightEncoder.reset();
      break;
    case 'S':
      robotstate = false;
      // robotstate = true;
      LOG.println("Robot Paused");
      break;
    case 'M':
      MOTORS_ENABLE = !MOTORS_ENABLE;
```

```
    // robotstate = true;
    LOG.println("Motors state changed");
    break;
  }
  Normal = {speedbase, kpdir, kidir, kddir};
  // BT.print("speed: "); BT.print(speedbase); BT.print("\t ");
  // BT.print("kp: "); BT.print(kpdir); BT.print("\t ");
  // BT.print("kd: "); BT.print(kddir); BT.print("\t ");
  // BT.print("ki: "); BT.print(kidir); BT.print("\t ");
  BT.println();
  setupPID(Normal);
}

void loop(){
  followLine();
}

void followLine(){

  // Timers
  startLogTimer = millis(); // Debug the loop
  startLapTimer = millis()/1000.0; // Time of a lap
  startAccTimer = millis()/1000.0; // Acceleration interval
  startCps_left_led_timer = millis()/1000.0; // Acceleration interval
  startCps_right_led_timer = millis()/1000.0; // Acceleration interval

  while(1){

    if(BT.available()>0){
      btcallback();
    }
    if (robotplay){
      robotplay = false;
      LOG.println("Robot will Start in 2s");
      delay(2000);
    }
    if (checkpoint_right_counter == 0) {
      LeftEncoder.reset();
      RightEncoder.reset();
    }

    leftDistance = PULSES2DISTANCE(LeftEncoder.getPulses());
    rightDistance = PULSES2DISTANCE(RightEncoder.getPulses());
    // currentPosition = AVG(leftDistance, rightDistance) + POSITION_FIX; //get average
    currentPosition = rightDistance + POSITION_FIX; //get average

    // Check if the robot complete the track
    if (currentPosition >= FINAL_TARGET_POSITION && STOP_BY_DISTANCE) {
```

```
  robotstate = false; // Stop the Robot
}

// Checks if medium lap time has been reached
nowLapTimer = millis()/1000.0;
float laptime;
if (nowLapTimer - startLapTimer > LAP_TIME && STOP_BY_TIME) {
  // startLapTimer = nowLapTimer;
  robotstate = false; // Stop the robot
}

if (!robotstate) { // Stop the Robot
  // Stop the robot and release the motors after
  LeftMotor.brake();
  RightMotor.brake();
  delay(500);
  LeftMotor.coast();
  RightMotor.coast();

  // Print some data for statistics
  float laptime = nowLapTimer - startLapTimer;
  float mediumspeed = currentPosition / laptime;
  LOG.print("Time Lap: \t");       LOG.print(laptime);
  LOG.print("s");
  LOG.print("\t");
  LOG.print("Track Length: \t ");  LOG.print(currentPosition);
  LOG.print("m");
  LOG.print("\t");
  LOG.print("Medium Speed: \t ");  LOG.print(mediumspeed);
  LOG.print("m/s");
  LOG.println();
  // Stop the Lap Timer
  startLapTimer = nowLapTimer;
  // Blink the LEDs
  while (!robotstate) {
    digitalWrite(PIN_LED, !digitalRead(PIN_LED));
    delay(500);
    if(BT.available()>0){
      btcallback();
    }
  }
}
else if (robotstate) { // Follow the Line

    //LEDS turn off - About Interrupt
    nowCps_left_led_timer = millis()/1000.0;
    if (nowCps_left_led_timer - startCps_left_led_timer > 0.150) {
      digitalWrite(PIN_LED,0);
```

```
  startCps_left_led_timer = nowCps_left_led_timer;
 }

 if (nowCps_right_led_timer - startCps_right_led_timer > 0.150) {
  digitalWrite(PIN_LED,0);
  startCps_right_led_timer = nowCps_right_led_timer;
 }

 // Check if changed mark
 if (currentPosition >= TargetMark.position && MAPPING_ENABLED) {
  LOG.print(currentPosition);
  LOG.print(": ");
  LOG.print(currentMark);
  LOG.print("--> ");
  currentMark++;
  LOG.print(currentMark);
  LOG.print(" ");
  // Get current Target Mark
  TargetMark = TRACK_EVENT_NAME[currentMark];
  acceleration = TargetMark.acceleration;
  // Update Robot Setup
  LOG.println(TargetMark.setup.speed);
  setupPID(TargetMark.setup);
 }

 // Position of the line: (left)-2500 to 2500(right)
 LineReader.readCalibrated(sensorValues, QTR_EMITTERS_ON);
 linePosition = LineReader.readLine(sensorValues, QTR_EMITTERS_OFF, WHITE_LINE) - 2500.0;

 // Disconsider the next of zero values
 linePosition = linePosition > -
STRAIGHT_FIX ? (linePosition < STRAIGHT_FIX ? 0 : linePosition) : linePosition;

 directioncontrol.setProcessValue(linePosition);
 directiongain = directioncontrol.compute();

 // Speed update with acceleration
 if (ACCELERATION_ENABLED) {
  nowAccTimer = millis()/1000.0;
  if (nowAccTimer - startAccTimer > ACCELERATION_INTERVAL) {
   // check if the robot accelerates or decelerates
   if ((acceleration > 0 && currentSpeed < targetSpeed) || (acceleration < 0 && currentSpeed > targ
etSpeed)) {
    currentSpeed += acceleration;// * ACCELERATION_INTERVAL;
    currentSpeed = currentSpeed > targetSpeed ? targetSpeed : currentSpeed < 0 ? 0 : currentSpeed;
    // LOG.println(currentSpeed);
   }
```

```
          startAccTimer = nowAccTimer;
        }
      } else {
        currentSpeed = targetSpeed;
      }

      // Set the Direction
      leftmotorspeed = currentSpeed + (directiongain > 0 ? -directiongain : 0);
      righmotorspeed = currentSpeed + (directiongain < 0 ? +directiongain : 0);

      // Constrain the speed value to [0, 100] interval
      leftmotorspeed = leftmotorspeed > 100 ? 100 : leftmotorspeed < -REVERSE ? -
REVERSE : leftmotorspeed;
      righmotorspeed = righmotorspeed > 100 ? 100 : righmotorspeed < -REVERSE ? -
REVERSE : righmotorspeed;

      // LOG.printf("PID is working? %f \n", directiongain);
      if(MOTORS_ENABLE){
        LeftMotor.speed(leftmotorspeed);
        RightMotor.speed(righmotorspeed);
      }
    }

    nowLogTimer = millis();

    if (nowLogTimer - startLogTimer > LOG_INTERVAL && LOG_ENABLED) {

      manualTrackMapping();
      startLogTimer = nowLogTimer;
    }
  }
}
```

APÊNDICE B – VÍDEO DO FUNCIONAMENTO DO ROBÔ NA COMPETIÇÃO SMILES

https://gustavosdepaula.github.io/images/portfolio/modals/m-sloth.mp4