Documentação do Código do Robô Tartaruga Linhas

Competição CoRA 2025 - UFMG

Lucas Lemos Ricaldoni

Setembro
2025

Sumário

1 Introdução

Este documento apresenta a documentação do código desenvolvido para o robô seguidor de linha "Tartaruga Linhas", projetado para a competição CoRA 2025 da UFMG. O sistema é baseado em um microcontrolador e utiliza sensores infravermelhos para a detecção da pista e um giroscópio para auxiliar na navegação.

2 Dependências

O código foi desenvolvido em C++ para a plataforma Arduino. As seguintes bibliotecas são necessárias para a compilação e funcionamento do projeto:

- `MPU6050.h`: Para comunicação com o giroscópio e acelerômetro MPU6050.
- `Wire.h`: Para comunicação I2C com o sensor MPU6050.
- `SPI.h` e `SD.h`: Para comunicação com o módulo de cartão SD.

3 Estrutura do Código Principal: `CoRA2025.ino`

O arquivo `CoRA2025.ino` é o coração do sistema, responsável por orquestrar todas as operações do robô, desde a leitura dos sensores até o controle dos motores e a execução das lógicas de desafios da competição.

3.1 Função `setup()`

Esta função é executada uma única vez quando o robô é ligado ou resetado. Sua principal finalidade é inicializar todos os componentes de hardware e software necessários para o funcionamento.

- Inicializa a comunicação serial e com o cartão SD (se os respectivos modos de depuração estiverem ativos).

- Configura os pinos dos sensores de linha e do LED como entradas e saídas.
- Inicializa a comunicação I2C (Wire) para o giroscópio.
- Inicia e calibra o giroscópio (MPU6050) para obter o valor de bias.
- Configura os motores para o controle de movimento.
- Acende o LED frontal para indicar que a inicialização foi concluída.

```
void setup() {
  // Initialize serial communication
  if (debugMode) Serial.begin(9600);
  if (debugSD) setup_sd();
  // Sensor initialization
  pinMode(sensor_esquerda, INPUT);
  pinMode(sensor_esquerda_central, INPUT);
  pinMode(sensor_central, INPUT);
  pinMode(sensor_direita_central, INPUT);
  pinMode(sensor_direita, INPUT);
  pinMode(sensor_curva_esquerda, INPUT);
  pinMode(sensor_curva_direita, INPUT);
  pinMode(LEDS, OUTPUT);
  // Gyroscope initialization
  Wire.begin();
  mpu.begin();

  // liga os motores
  setup_motor();

  // Calibrate the gyroscope
  gyro_bias_z = calibrate_gyro();
  // liga os leds da cara para indicar que o robo iniciou
  digitalWrite(LEDS, HIGH);
  tempoLedLigou = millis();
  ledLigado = true;
}
```

Listing 1: Função de inicialização do robô.

3.2 Função `setup_sd()`

Esta função inicializa o cartão SD e cria um novo arquivo de log. Para evitar a sobrescrita de dados, a função gera um nome de arquivo único a

cada inicialização, usando o valor da constante `Kp` e um índice sequencial (ex: `L180_0.TXT`, `L180_1.TXT`). Se o arquivo for criado com sucesso, um cabeçalho detalhado é escrito para facilitar a análise posterior dos dados.

```
void setup_sd() {
  if (!SD.begin(chipSelect)) {
    if (debugMode) Serial.println("SD Card initialization failed!");
    return;
  }

  char newFileName[16];

  // Procura o primeiro indice disponivel
  for (int i = 0; i < 100; i++) {
    // Formata o nome do arquivo de forma segura usando snprintf
    snprintf(newFileName, sizeof(newFileName), "L%d_%d.TXT", (int)Kp, i);
    // Verifica se o arquivo Nao existe
    if (!SD.exists(newFileName)) {
      break;
      // Encontrou um nome disponivel
    }
  }

  if (debugMode) {
    Serial.print("Creating new log file: ");
    Serial.println(newFileName);
  }

  // Abre o novo arquivo para escrita
  logFile = SD.open(newFileName, FILE_WRITE);
  if (logFile) {
    logFile.println("Time,Error,Challenge,MarcacaoDireita,MarcacaoEsquerda,T_Dir,T_Esq,Velocidade Direita,Velocidade Esquerda,sc0,s0,s1,s2,s3,s4,sc1");
    logFile.flush();
    if(debugMode) Serial.println("Log file created successfully.");
  } else {
    if(debugMode) Serial.println("Failed to create the log file.");
  }
}
```

Listing 2: Inicialização do cartão SD e criação do arquivo de log.

3.3 Função `write_sd()`

Responsável por escrever uma linha de dados no arquivo de log. A cada chamada, a função registra um conjunto completo de dados de telemetria, incluindo o tempo (‘millis()‘), o erro, um marcador de desafio, contagem e tempo das marcações, velocidades dos motores e o estado de todos os sete sensores. O uso de ‘logFile.flush()‘ garante que os dados sejam gravados imediatamente no cartão.

```
void write_sd(int challenge_marker = 0) {
  if (logFile) {
    logFile.print(millis());
    logFile.print(",");
    logFile.print(erro);
    logFile.print(",");
    logFile.print(challenge_marker);
    logFile.print(",");
    logFile.print(marcacoesDireita);
    logFile.print(",");
    logFile.print(marcacoesEsquerda);
    logFile.print(",");
    logFile.print(tempoMarcacaoDireita);
    logFile.print(",");
    logFile.print(tempoMarcacaoEsquerda);
    logFile.print(",");
    logFile.print(velocidadeDireita);
    logFile.print(",");
    logFile.print(velocidadeEsquerda);
    for (int i = 0; i < 5; i++) {
      logFile.print(",");
      logFile.print(SENSOR[i]);
    }
    for (int i = 0; i < 2; i++) {
      logFile.print(",");
      logFile.print(SENSOR_CURVA[i]);
    }
    logFile.println();
    logFile.flush();

    if (debugMode) {
      if (challenge_marker != 0) Serial.println("Challenge
   event logged to SD.");
    }
  }
}
```

Listing 3: Escreve dados de telemetria no cartão SD.

3.4 Função `ler_sensores()`

Responsável por ler o estado digital dos sensores de linha e dos sensores de curva.

- Atualiza os arrays globais `SENSOR` e `SENSOR_CURVA` com os valores lidos.
- Um valor `PRETO` (1) indica que o sensor detectou a linha, enquanto `BRANCO` (0) indica o contrário.

```
void ler_sensores() {
  SENSOR[0] = digitalRead(sensor_esquerda);
  SENSOR[1] = digitalRead(sensor_esquerda_central);
  SENSOR[2] = digitalRead(sensor_central);
  SENSOR[3] = digitalRead(sensor_direita_central);
  SENSOR[4] = digitalRead(sensor_direita);
  SENSOR_CURVA[0] = digitalRead(sensor_curva_esquerda);
  SENSOR_CURVA[1] = digitalRead(sensor_curva_direita);
}
```

Listing 4: Leitura dos sensores de linha e curva.

3.5 Função `ajusta_movimento()`

Ajusta a velocidade dos motores com base no valor de saída do controlador PID.

- Calcula a velocidade para o motor direito e esquerdo, subtraindo e somando o valor do PID à velocidade base, respectivamente.
- Utiliza a função `constrain` para garantir que os valores de velocidade permaneçam dentro do intervalo válido (0 a 255).
- Aciona os motores com as novas velocidades calculadas.

```
void ajusta_movimento() {
  // Change the speed value
  velocidadeDireita = constrain(velocidadeBaseDireita - PID,
   0, 255);
  velocidadeEsquerda = constrain(velocidadeBaseEsquerda + PID
   , 0, 255);
  // Send the new speed to the run function
```

```
  run(velocidadeDireita, velocidadeEsquerda);
}
```

Listing 5: Ajuste da velocidade dos motores.

3.6 Função `calcula_erro()`

Calcula o erro de posição do robô em relação à linha.

- O erro é calculado utilizando uma média ponderada das leituras dos 5 sensores de linha.
- O resultado indica a distância e a direção do desvio do robô em relação ao centro da linha.
- Se todos os sensores estiverem sobre a cor preta, a função considera que a linha foi perdida.
- Antes do cálculo, a função chama `verifica_inversao()` e, se uma inversão for concluída, ativa a flag `faixa_de_pedestre`.

```
void calcula_erro() {
  // Update sensor values
  ler_sensores();
  // Check for inversion, signaling a pedestrian crossing
  verifica_inversao(SENSOR, SENSOR_CURVA);

  if (inversao_finalizada) {
    faixa_de_pedestre = true;
  }

  // Initialize variables for error calculation
  int pesos[5] = {-2, -1, 0, 1, 2};
  int somatorioErro = 0;
  int sensoresAtivos = 0;

  // Perform a summation with the sensor values and weights
  for (int i = 0; i < 5; i++) {
    somatorioErro += SENSOR[i] * pesos[i];
    sensoresAtivos += SENSOR[i];
  }

  // Determine the car's error
  if (sensoresAtivos == QUANTIDADE_TOTAL_SENSORES) {
```

```
24      erro = LINHA_NAO_DETECTADA;
25    } else {
26      int sensoresInativos = QUANTIDADE_TOTAL_SENSORES -
       sensoresAtivos;
27      erro = somatorioErro / sensoresInativos;
28    }
29  }
```

Listing 6: Cálculo do erro de posição.

3.7 Função `calcula_PID()`

Implementa o controlador PID (Proporcional-Integral-Derivativo).

- **Proporcional (P)**: Reage proporcionalmente ao erro atual.
- **Integral (I)**: Acumula o erro ao longo do tempo para corrigir desvios persistentes.
- **Derivativo (D)**: Responde à taxa de variação do erro para amortecer oscilações.
- O valor final do PID é a soma ponderada dessas três componentes e é usado para ajustar a velocidade dos motores.

```
1  void calcula_PID() {
2    // Initialize variables for calculation
3    PID = 0;
4    P = erro;
5    I = constrain(I + P, -255, 255);
6    D = erro - erroAnterior;
7    // Calculate PID
8    PID = (Kp * P) + (Ki * I) + (Kd * D) + OFFSET;
9    // Update the previous error value
10   erroAnterior = erro;
11 }
```

Listing 7: Cálculo do controle PID.

3.8 Função `imprime_serial()`

Envia informações de depuração para o monitor serial.

- Imprime o estado dos sensores de curva e de linha.
- Invoca as funções de cálculo de erro e PID e imprime seus resultados, além das velocidades resultantes dos motores.

```
void imprime_serial() {
  // Print sensor values
  Serial.print(SENSOR_CURVA[0]);
  Serial.print(" | ");

  for (int i = 0; i < 5; i++) {
    Serial.print(SENSOR[i]);
    Serial.print(" | ");
  }

  Serial.print(SENSOR_CURVA[1]);
  Serial.print(" | ");

  // Print Error, PID, and speed variables
  Serial.print("\tErro: ");
  calcula_erro();
  Serial.print(erro);
  Serial.print(" PID: ");
  calcula_PID();
  Serial.print(PID);
  Serial.print(" Velocidade Direita: ");
  Serial.print(velocidadeDireita);
  Serial.print(" Velocidade Esquerda: ");
  Serial.println(velocidadeEsquerda);
}
```

Listing 8: Impressão de dados de depuração.

3.9 Função Principal `loop()`

Este é o ciclo principal do programa, executado continuamente. A nova lógica é uma máquina de estados mais sofisticada para lidar com os diversos cenários da competição.

3.9.1 Lógica de Competição

- **Modo Arrancada**: Se `arrancadaMode` for verdadeiro, o robô executa um simples seguidor de linha sem a lógica de desafios.

- **Modo Competição**:

    1. **Leitura e Detecção**: A cada ciclo, os sensores são lidos e é verificado se há uma curva de 90 graus.
    2. **Faixa de Pedestres**: É detectada após uma inversão de pista. O robô aguarda uma confirmação (contando 3 leituras de "todos os sensores no preto") antes de executar a manobra de parada.
    3. **Seguir Linha**: Se nenhuma curva for detectada e a linha estiver visível, o robô segue a pista normalmente usando o controle PID. As tentativas de recuperação de linha são resetadas após um tempo se o robô permanecer estável.
    4. **Detecção de Curva/Desafio**: Se uma curva de 90 graus é detectada, o robô analisa as marcações laterais para decidir a manobra.
        - **Curva Simples**: Se uma ou mais marcações são vistas de apenas um lado, executa uma curva de 90 graus para esse lado.
        - **Marcha à Ré vs. Cruzamento**: Se marcações são vistas em ambos os lados, o robô mede a diferença de tempo entre a detecção delas. Uma diferença de tempo significativa indica uma manobra de marcha à ré. Se o tempo for muito próximo, indica um cruzamento em dúvida.

3.9.2 Tratamento de Exceções (Perda de Linha)

- **Tolerância**: Se a linha é perdida, um contador é iniciado. A ação só é tomada após um número de ciclos (`LIMITE_TOLERANCIA_LINHA_PERDIDA`) para ignorar falhas momentâneas.
- **Recuperação**: O robô executa a função `tenta_recuperar_linha()`. Se for bem-sucedido, incrementa um contador de tentativas.
- **Área de Parada**: Se a recuperação falhar, ou se o número de recuperações em um curto período de tempo exceder o limite (`LIMITE_TENTATIVAS_RECUPERACAO`), o robô para completamente na área de parada, considerando o percurso finalizado.

3.9.3 Modo de Depuração

Se a flag `debugMode` estiver ativa, a lógica principal é ignorada e o robô imprime continuamente os dados dos sensores no monitor serial para calibração e análise.

```
56 void loop() {
57   // verifica o estado do led
58   verifica_estado_led();
59   if (!debugMode) {
60     if (arrancadaMode) {
61       ler_sensores();
62       calcula_erro();
63       calcula_PID();
64       ajusta_movimento();
65       if (debugSD) write_sd(0);
66     } else {
67       ler_sensores();
68       int posicao = calcula_posicao(SENSOR);
69       if (posicao != 0) {
70         ultima_posicao_linha = posicao;
71       }
72       // verifica se tem uma curva de 90
73       int saidaCurva = verifica_curva_90(SENSOR, SENSOR_CURVA
    );
74       if (!inversaoAtiva){
75         if (saidaCurva != CURVA_NAO_ENCONTRADA && (millis() -
     tempoUltimaCurva < DEBOUNCE_TEMPO_CURVA)) {
76           saidaCurva = CURVA_NAO_ENCONTRADA;
77         }
78       }
79
80       if (faixa_de_pedestre) {
81         // filtro de ruido
82         if (calcula_sensores_ativos(SENSOR) ==
    QUANTIDADE_TOTAL_SENSORES) {
83           qnt_fim_inversao += 1;
84         }
85
86         if (qnt_fim_inversao >= 3) {
87           if (debugSD) write_sd(2);
88           realiza_faixa_de_pedestre();
89           faixa_de_pedestre = false;
90         }
91       }
92
93       if (saidaCurva == CURVA_NAO_ENCONTRADA) {
94         calcula_erro();
```

```
95        if (erro != LINHA_NAO_DETECTADA) {
96          // segue normalmente
97          calcula_PID();
98          if ((SENSOR_CURVA[0] == PRETO && SENSOR_CURVA[1] ==
     PRETO) || inversaoAtiva) {
99            ajusta_movimento();
100         }
101
102         if (millis() - tempoUltimaRecuperacao >
    TEMPO_RESET_TENTATIVAS) {
103           tentativasRecuperacao = 0;
104         }
105
106         if (debugSD) write_sd(0);
107         contadorLinhaPerdida = 0;
108       } else if (erro == LINHA_NAO_DETECTADA) {
109         // perdeu a linha
110         if (contadorLinhaPerdida == 0) {
111           tempoSemLinha = millis();
112         }
113         contadorLinhaPerdida++;
114         if (contadorLinhaPerdida >
    LIMITE_TOLERANCIA_LINHA_PERDIDA || millis() -
    tempoSemLinha > 1000) {
115
116           stop_motors();
117           if (tenta_recuperar_linha()) {
118             contadorLinhaPerdida = 0;
119             tentativasRecuperacao++;
120             tempoUltimaRecuperacao = millis();
121
122             if (tentativasRecuperacao >=
    LIMITE_TENTATIVAS_RECUPERACAO) {
123               erro = erroAnterior;
124               if (debugSD) write_sd(3);
125               area_de_parada();
126             }
127           } else {
128             erro = erroAnterior;
129             if (debugSD) write_sd(3); // Log challenge 3:
    Stop area
130             area_de_parada();
131           }
132         }
133       }
134     } else if (saidaCurva != CURVA_NAO_ENCONTRADA) {
135       // evita ligar no cruzamento
136       if (calcula_sensores_ativos(SENSOR) > 1) {
137         if (!inversaoAtiva) {
```

```
138             marcacoesDireita = 0;
139             marcacoesEsquerda = 0;
140
141             analisa_marcacoes();
142
143             if (marcacoesDireita == 1 && marcacoesEsquerda ==
    1 ) {
144               unsigned long deltaTempo;
145               if (tempoMarcacaoDireita >
  tempoMarcacaoEsquerda) {
146                 deltaTempo = tempoMarcacaoDireita -
  tempoMarcacaoEsquerda;
147               } else {
148                 deltaTempo = tempoMarcacaoEsquerda -
  tempoMarcacaoDireita;
149               }
150
151               if (deltaTempo >= TOLERANCIA_TEMPO_SIMULTANEO)
  {
152                 saidaCurva = (tempoMarcacaoDireita <
  tempoMarcacaoEsquerda) ?
153                 CURVA_DIREITA : CURVA_ESQUERDA;
154                 realiza_marcha_re(saidaCurva);
155               } else {
156                 turn_90(CURVA_EM_DUVIDA);
157                 if (debugSD) write_sd(6);
158               }
159             } else if (marcacoesDireita >= 1 ||
  marcacoesEsquerda >= 1) {
160               turn_90(saidaCurva);
161             }
162           }
163
164           stop_motors();
165         }
166       }
167     }
168   } else {
169     if (debugMotor) {
170       test_motors();
171     } else {
172       // Get the output of the car's data
173       ler_sensores();
174       imprime_serial();
175     }
176   }
177
178   delay(5);
```

```
179 }
```

Listing 9: Loop principal de execução do robô.

4 Arquivo de Desafios: `challenges.cpp`

Este arquivo contém a lógica para lidar com os desafios específicos da competição. Ele foi reestruturado para incluir uma máquina de estados mais robusta para a detecção de inversões e lógicas mais refinadas para cada manobra.

4.1 Função `verifica_estado_led()`

Uma função simples de gerenciamento que verifica se o tempo de inicialização do LED (`TEMPO_MAX_LED_LIGADO`) já passou e, em caso afirmativo, desliga o LED.

```
void verifica_estado_led() {
  if (ledLigado) {
    if (millis() - tempoLedLigou >= TEMPO_MAX_LED_LIGADO) {
      digitalWrite(LEDS, LOW);
      ledLigado = false;
    }
  }
}
```

Listing 10: Verifica e desliga o LED de status após o tempo definido.

4.2 Função `calcula_sensores_ativos()`

Esta função auxiliar conta quantos dos cinco sensores principais de linha estão atualmente sobre a cor preta.

```
int calcula_sensores_ativos(int SENSOR[]) {
  int sensoresAtivos = 0;
  for(int i = 0; i < 5; i++) {
    sensoresAtivos += SENSOR[i];
  }
  return sensoresAtivos;
```

```
}
```

Listing 11: Calcula o número de sensores ativos.

4.3 Função `verifica_curva_90()`

Analisa a combinação de leituras dos sensores para determinar se o robô encontrou uma curva de 90 graus. A lógica identifica uma curva quando pelo menos 2 sensores centrais estão ativos e um dos sensores de curva laterais também está ativo.

```
int verifica_curva_90(int SENSOR[], int SENSOR_CURVA[]) {
  int sensores_pretos = calcula_sensores_ativos(SENSOR);

  if (SENSOR_CURVA[0] == PRETO && sensores_pretos >= 2 &&
   SENSOR_CURVA[1] == BRANCO) {
    return CURVA_ESQUERDA;
  }
  if (SENSOR_CURVA[0] == BRANCO && sensores_pretos >= 2 &&
   SENSOR_CURVA[1] == PRETO) {
    return CURVA_DIREITA;
  }
  if (SENSOR[1] == PRETO && SENSOR[3] == PRETO &&
   SENSOR_CURVA[0] == BRANCO && SENSOR_CURVA[1] == BRANCO) {
    return CURVA_EM_DUVIDA;
  }

  return CURVA_NAO_ENCONTRADA;
}
```

Listing 12: Verifica a existência de uma curva de 90 graus.

4.4 Função `calcula_posicao()`

Calcula uma posição média ponderada da linha com base nos sensores ativos, retornando um valor entre -2 e 2. É usada para fazer pequenos ajustes de ângulo antes de uma curva de 90 graus.

```
int calcula_posicao(int SENSOR[]) {
  int pesos[5] = {-2, -1, 0, 1, 2};
  int somaPesos = 0, somaAtivos = 0;

  for (int i = 0; i < 5; i++) {
```

```
    if (SENSOR[i] == PRETO) {
      somaPesos += pesos[i];
      somaAtivos++;
    }
  }

  if (somaAtivos == 0) return 0;
  return somaPesos / somaAtivos;
}
```

Listing 13: Calcula a posição ponderada da linha.

4.5 Função `turn_90()`

Executa a manobra de curva de 90 graus. A função avança até passar da marcação inicial, para, calcula um ângulo de giro ajustado com base na posição da linha e, após a rotação, avança novamente até se realinhar na pista para evitar detecções duplicadas.

```
void turn_90(int curvaEncontrada) {
  if (!inversaoAtiva) {
    unsigned long startTime = millis();
    while (calcula_sensores_ativos(SENSOR) <= 1) {
      run(velocidadeBaseDireita, velocidadeBaseEsquerda);
      ler_sensores();
    }
    delay(50);

    stop_motors();
    delay(100);
    int posicao = calcula_posicao(SENSOR);
    int ajuste = posicao * 0;
    int anguloFinal = 80 + ajuste;

    stop_motors();
    delay(200);

    erro = erroAnterior;
    if (curvaEncontrada == CURVA_ESQUERDA) {
      turn_right(velocidadeBaseDireita,
   velocidadeBaseEsquerda);
      turn_until_angle(anguloFinal);
    } else if (curvaEncontrada == CURVA_DIREITA) {
      turn_right(velocidadeBaseDireita,
   velocidadeBaseEsquerda);
      turn_until_angle(anguloFinal);
```

```
26      }
27
28      stop_motors();
29      delay(100);
30      while (true) {
31        calcula_erro();
32        calcula_PID();
33        ajusta_movimento();
34        if (SENSOR_CURVA[0] == PRETO && SENSOR_CURVA[1] ==
      PRETO && SENSOR[2] == BRANCO) {
35          break;
36        }
37      }
38      tempoUltimaCurva = millis();
39      marcacoesDireita = 0; marcacoesDireita = 0;
40    }
41  }
```

Listing 14: Executa uma curva de 90 graus aprimorada.

4.6 Função `calibrate_gyro()`

Calibra o giroscópio medindo seu desvio (bias) enquanto o robô está parado. A média de múltiplas leituras é calculada e armazenada para corrigir as medições futuras.

```
1  float calibrate_gyro(int samples = 200) {
2      if (debugMode) Serial.println("Calibrating gyroscope...
       Keep the robot stationary.");
3
4      float sum_gz = 0;
5      for (int i = 0; i < samples; i++) {
6          mpu.update();
7          sum_gz += mpu.getGyroZ();
8          delay(10);
9      }
10
11     float bias_gz = sum_gz / samples;
12
13     if (debugMode) {
14       Serial.print("Calibration complete. Gyroscope Bias (Gz)
       = ");
15       Serial.println(bias_gz, 4);
16     }
17
18     return bias_gz;
```

```
}
```

Listing 15: Calibra o giroscópio.

4.7 Função `turn_until_angle()`

Realiza uma rotação precisa do robô até que um ângulo alvo seja atingido, utilizando o giroscópio.

```
void turn_until_angle(int target_angle = 90) {
  unsigned long previous_time = millis();
  float angle_z = 0;

  while (abs(angle_z) < target_angle) {
    mpu.update();
    float angular_velocity_z = mpu.getGyroZ() - gyro_bias_z;

    unsigned long current_time = millis();
    float delta_time = (current_time - previous_time) /
   1000.0;
    previous_time = current_time;

    angle_z += angular_velocity_z * delta_time;
    delay(10);
  }
  stop_motors();
}
```

Listing 16: Gira o robô até um ângulo específico.

4.8 Função `inverte_sensor()`

Inverte a lógica de um sensor (1 vira 0, 0 vira 1), permitindo que o robô siga uma linha de cor invertida.

```
int inverte_sensor(int sensor){
  if (sensor == 1){
    return 0;
  }
  return 1;
}
```

Listing 17: Inverte o valor de um sensor.

4.9 Função `verifica_inversao()`

Implementa uma máquina de estados para detectar e gerenciar a travessia de uma seção de pista com cores invertidas.

- **Entrada**: Ativa o modo de inversão (`inversaoAtiva`) quando apenas 1 sensor de linha está ativo.
- **Durante a Inversão**: Inverte a leitura de todos os sensores para que a lógica do seguidor de linha continue funcionando.
- **Saída**: Desativa o modo de inversão quando os sensores de curva e de linha indicam o fim da seção. Ao sair, ativa a flag `inversao_finalizada` para sinalizar uma possível faixa de pedestres.

```
bool verifica_inversao(int SENSOR[], int SENSOR_CURVA[]) {
  int ativos = calcula_sensores_ativos(SENSOR);

  if (ativos == 1 && !inversaoAtiva && SENSOR_CURVA[0] ==
   BRANCO && SENSOR_CURVA[1] == BRANCO) {
    inversaoAtiva = true;
    tempoInversaoAtivada = millis();
  }

  if (inversaoAtiva && millis() - tempoInversaoAtivada >
   1000) {
    if (ativos >= 3 && SENSOR_CURVA[0] == PRETO &&
   SENSOR_CURVA[1] == PRETO) {
      inversaoAtiva = false;
      inversao_finalizada = true;
      return false;
    }
  }

  if (inversaoAtiva) {
    for (int i = 0; i < 5; i++) {
      SENSOR[i] = inverte_sensor(SENSOR[i]);
    }
    return true;
  }
  return false;
}
```

Listing 18: Verifica e trata a inversão de cores da pista.

4.10 Função `realiza_faixa_de_pedestre()`

Executa uma rotina robusta para a faixa de pedestres. O robô para pelo tempo definido, dá uma pequena ré para encontrar a linha novamente, se realinha sobre ela e então avança em alta velocidade para garantir a travessia completa.

```
void realiza_faixa_de_pedestre() {
  stop_motors();
  delay(TIMEOUT_FAIXA_PEDESTRE);

  unsigned long start = millis();
  while (erro == LINHA_NAO_DETECTADA || millis() - start <
   700) {
    ler_sensores();
    calcula_erro();
    run_backward(150, 150);
  }
  stop_motors();

  int pos = calcula_posicao(SENSOR);
  if (pos < -1) {
    run(-velocidadeBaseDireita, velocidadeBaseEsquerda);
  } else if (pos > 1) {
    run(velocidadeBaseDireita, -velocidadeBaseEsquerda);
  }
  while (SENSOR[1] != PRETO && SENSOR[2] != BRANCO && SENSOR
   [3] != PRETO) {
    ler_sensores();
  }
  stop_motors();
  delay(500);

  run(255, 255);
  delay(TIMEOUT_PERIODO_FAIXA);

  // ... (L gica de finaliza  o da travessia)

  inversao_finalizada = false;
}
```

Listing 19: Executa o desafio da faixa de pedestres com realinhamento.

4.11 Função `realiza_marcha_re()`

Executa o desafio da marcha à ré. O robô para, move-se para trás em alta velocidade, para novamente, avança um pouco para se posicionar e então realiza a curva de 90 graus para o lado determinado.

```
void realiza_marcha_re(int lado_da_curva) {
  if (debugSD) write_sd(7);
  stop_motors();
  delay(1000);

  run_backward(255, 255);
  delay(1200);

  stop_motors();
  delay(500);

  run(velocidadeBaseDireita, velocidadeBaseEsquerda);
  delay(200);

  turn_90(lado_da_curva);

  stop_motors();
  delay(500);
}
```

Listing 20: Executa o desafio de marcha à ré.

4.12 Função `realiza_rotatoria()`

Controla a navegação na rotatória. Após entrar, o robô segue a linha e utiliza uma lógica com flag (`aguardando_realinhar`) e um contador de detecção para contar as saídas de forma confiável, evitando que a mesma saída seja contada múltiplas vezes. Ao atingir a saída desejada, executa a manobra para sair da rotatória.

```
void realiza_rotatoria(int saidaCurva, int saidaDesejada) {
  bool aguardando_realinhar = false;
  static int contador_deteccao_saida = 0;
  const int TOLERANCIA_SAIDA = 2;
  int saidaAtual = 0;

  if (saidaCurva == SAIDA_ESQUERDA) {
    turn_left(velocidadeBaseDireita, velocidadeBaseEsquerda);
    turn_until_angle(90);
```

```
10    } else if (saidaCurva == SAIDA_DIREITA) {
11      turn_right(velocidadeBaseDireita, velocidadeBaseEsquerda)
       ;
12      turn_until_angle(90);
13    }
14
15    while (saidaAtual < saidaDesejada) {
16      calcula_erro();
17      ajusta_movimento();
18
19      bool saida_detectada = (SENSOR_CURVA[0] == PRETO &&
       SENSOR_CURVA[1] == BRANCO);
20      bool robo_realinhado = (SENSOR_CURVA[0] == PRETO &&
       SENSOR_CURVA[1] == PRETO);
21
22      if (!aguardando_realinhar) {
23        if (saida_detectada) {
24          contador_deteccao_saida++;
25        } else {
26          contador_deteccao_saida = 0;
27        }
28        if (contador_deteccao_saida >= TOLERANCIA_SAIDA) {
29          saidaAtual++;
30          aguardando_realinhar = true;
31          contador_deteccao_saida = 0;
32        }
33      } else {
34        if (robo_realinhado) {
35          aguardando_realinhar = false;
36        }
37      }
38    }
39
40    if (saidaCurva == SAIDA_ESQUERDA) {
41        turn_right(velocidadeBaseDireita,
       velocidadeBaseEsquerda);
42        turn_until_angle(90);
43    } else {
44        turn_left(velocidadeBaseDireita, velocidadeBaseEsquerda
       );
45        turn_until_angle(90);
46    }
47  }
```

Listing 21: Executa o desafio da rotatória com contagem de saídas aprimorada.

4.13 Função `tenta_recuperar_linha()`

Rotina de recuperação executada quando a linha é perdida. O robô move-se para trás por um tempo limite (`TIME_WITHOUT_LINE`), procurando pela linha. Para confirmar que a linha foi reencontrada, exige 3 leituras consecutivas válidas.

```
bool tenta_recuperar_linha() {
  unsigned long tempoPerdido = millis();
  int leiturasValidas = 0;

  while (millis() - tempoPerdido < TIME_WITHOUT_LINE) {
    run_backward(velocidadeBaseDireita,
   velocidadeBaseEsquerda);
    delay(50);
    ler_sensores();

    if (SENSOR[1] == PRETO || SENSOR[2] == PRETO || SENSOR[3]
    == PRETO) {
      leiturasValidas++;
      if (leiturasValidas >= 3) {
        stop_motors();
        return true;
      }
    } else {
      leiturasValidas = 0;
    }
  }
  return false;
}
```

Listing 22: Tenta recuperar a linha movendo-se para trás.

4.14 Função `conta_marcacao()`

Realiza a contagem de marcadores (quadrados pretos). A lógica foi aprimorada para contar na transição para o branco e exigir um número mínimo de leituras (`TOLERANCIA_MARCACAO`) para evitar ruídos, garantindo que cada marcador seja contado apenas uma vez.

```
int conta_marcacao(int estadoSensor, int contagemAtual, bool
   &jaContou, int &contadorBranco, unsigned long &
   tempoMarcacao) {
  if (calcula_sensores_ativos(SENSOR) <= 1) return
   contagemAtual;
```

```
  if (marcacoesDireita >= 1 && marcacoesEsquerda >= 1) return
    contagemAtual;

  if (estadoSensor == BRANCO) {
    contadorBranco++;
    if (contadorBranco >= TOLERANCIA_MARCACAO && !jaContou) {
      jaContou = true;
      tempoMarcacao = millis();
      return contagemAtual + 1;
    }
  } else {
    contadorBranco = 0;
    jaContou = false;
  }
  return contagemAtual;
}
```

Listing 23: Contagem de marcadores de pista com filtro de ruído.

4.15 Função `analisa_marcacoes()`

Orquestra a detecção de desafios. Ela entra em um loop por um tempo definido (`TIMEOUT_MARCACAO`) após a detecção de uma curva, chamando `conta_marcacao` para registrar os marcadores de cada lado e seus respectivos tempos de detecção.

```
void analisa_marcacoes() {
  if (!inversaoAtiva) {
    unsigned long tempoUltimaDeteccao = millis();
    jaContouEsquerda = false;
    jaContouDireita = false;

    while((millis() - tempoUltimaDeteccao < TIMEOUT_MARCACAO)
   ) {
      ler_sensores();

      static int contadorBrancoEsq = 0;
      static int contadorBrancoDir = 0;

      int marcacoesAntesEsq = marcacoesEsquerda;
      int marcacoesAntesDir = marcacoesDireita;

      marcacoesEsquerda = conta_marcacao(SENSOR_CURVA[0],
   marcacoesEsquerda, jaContouEsquerda, contadorBrancoEsq,
   tempoMarcacaoEsquerda);
```

```
17        marcacoesDireita  = conta_marcacao(SENSOR_CURVA[1],
    marcacoesDireita, jaContouDireita, contadorBrancoDir,
    tempoMarcacaoDireita);
18
19        if (marcacoesEsquerda > marcacoesAntesEsq ||
    marcacoesDireita > marcacoesAntesDir) {
20          tempoUltimaDeteccao = millis();
21        }
22
23        if (marcacoesDireita >= 1 && marcacoesEsquerda >= 1) {
24          marcacoesDireita = 1;
25          marcacoesEsquerda = 1;
26        }
27        calcula_erro();
28        calcula_PID();
29        ajusta_movimento();
30      }
31    }
32  }
```

Listing 24: Analisa e conta marcadores de pista por um tempo determinado.

4.16 Função `area_de_parada()`

Função final que é chamada quando o robô não consegue se recuperar de uma perda de linha ou atinge o limite de tentativas. Ele avança um pouco, para os motores, acende o LED e entra em um loop infinito, finalizando o percurso.

```
1  void area_de_parada() {
2    run(velocidadeBaseDireita - 5, velocidadeBaseEsquerda);
3    delay(1000);
4    stop_motors();
5    digitalWrite(LEDS, HIGH);
6    tempoLedLigou = millis();
7    ledLigado = true;
8    if (debugSD) write_sd(3); // Log challenge 3: Stop area
9    while(true);
10 }
```

Listing 25: Finaliza o percurso na área de parada.

5 Arquivo de Controle dos Motores: `motors.cpp`

Este arquivo representa a camada de abstração de hardware para o controle dos motores do robô. Ele é responsável por inicializar os pinos do microcontrolador e fornecer uma interface simples para executar movimentos básicos como andar para frente, para trás, girar e parar.

5.1 Função `setup_motor()`

Inicializa os pinos de controle dos motores como saídas.

```
void setup_motor() {
  pinMode(MOTOR_LEFT_CLKWISE, OUTPUT);
  pinMode(MOTOR_LEFT_ANTI, OUTPUT);
  pinMode(MOTOR_RIGHT_CLKWISE, OUTPUT);
  pinMode(MOTOR_RIGHT_ANTI, OUTPUT);
  stop_motors();
}
```

Listing 26: Configuração inicial dos pinos dos motores.

5.2 Função `set_state_motor()`

Função de baixo nível que aplica diretamente os valores de PWM (0-255) aos pinos de controle dos motores.

```
void set_state_motor(int leftCw, int leftCcw, int rightCw,
    int rightCcw) {
  analogWrite(MOTOR_LEFT_CLKWISE, leftCw);
  analogWrite(MOTOR_LEFT_ANTI, leftCcw);
  analogWrite(MOTOR_RIGHT_CLKWISE, rightCw);
  analogWrite(MOTOR_RIGHT_ANTI, rightCcw);
}
```

Listing 27: Define o estado e a velocidade de cada motor.

6 Arquivo de Constantes e Variáveis Globais: `constants.cpp`

Este arquivo centraliza todas as constantes e variáveis globais. Essa abordagem facilita o ajuste de parâmetros e o gerenciamento de estados do robô sem alterar a lógica principal.

6.1 Pinos e Hardware

Define as conexões físicas dos sensores, LEDs e o pino Chip Select para o cartão SD.

6.2 Variáveis de Estado dos Sensores

Arrays que armazenam as leituras dos sensores de linha (`SENSOR`) e de curva (`SENSOR_CURVA`).

6.3 Configurações de Velocidade e Motores

- `velocidadeBase`: A velocidade padrão, que é mais alta (255) no modo de arrancada e menor (210) no modo de competição.
- `OFFSET_MOTORS`: Um valor de ajuste para compensar diferenças mecânicas entre os motores, garantindo um movimento reto.
- `velocidadeDireita`, `velocidadeEsquerda`: Variáveis que armazenam a velocidade ajustada pelo PID.

6.4 Controle PID

- **Ganhos**: `Kp`, `Ki`, `Kd`. Notavelmente, no modo de arrancada (`arrancadaMode`), os ganhos `Ki` e `Kd` são zerados, transformando o controle em um sistema puramente Proporcional, mais simples e agressivo.
- **Variáveis de Cálculo**: `erro`, `P`, `I`, `D`, `PID`, etc., usadas para o cálculo em tempo de execução.

6.5 Variáveis de Estado dos Desafios

Flags e contadores que gerenciam o estado do robô durante a competição.

- `inversaoAtiva`: Flag que indica se o robô está em uma seção de pista invertida.
- `inversao_finalizada`: Flag ativada ao sair de uma inversão, sinalizando uma possível faixa de pedestres.
- `marcacoesDireita`, `marcacoesEsquerda`: Contadores para os marcadores de desafio.
- `tempoMarcacaoDireita`, `tempoMarcacaoEsquerda`: Armazenam o `millis()` no momento da detecção de um marcador, usado para diferenciar desafios.
- `ultima_posicao_linha`: Guarda a última posição válida da linha para auxiliar na recuperação.

6.6 Controle de Depuração (Debug)

Flags booleanas para ativar diferentes modos de teste.

- `debugMode`: Ativa a impressão de dados no monitor serial.
- `debugMotor`: Ativa um teste de motores no `loop()`.
- `debugSD`: Habilita a gravação de logs no cartão SD.
- `arrancadaMode`: Ativa um modo de performance simplificado, com maior velocidade e controle apenas Proporcional.

6.7 Parâmetros de Tolerância e Tempo

Constantes que definem o comportamento temporal do robô.

- `TIMEOUT_FAIXA_PEDESTRE`: Tempo de parada na faixa.

- `TIMEOUT_MARCACAO`: Duração da janela de tempo para analisar os marcadores de um desafio.
- `DEBOUNCE_TEMPO_CURVA`: Tempo de espera para evitar a detecção dupla de uma mesma curva.
- `TOLERANCIA_TEMPO_SIMULTANEO`: Limiar de tempo para diferenciar uma marcha à ré de um cruzamento.
- `LIMITE_TOLERANCIA_LINHA_PERDIDA`: Número de ciclos de "linha perdida" antes de iniciar a recuperação.

7 Script de Análise e Visualização: `plot_log.py`

Para auxiliar no processo de calibração do controlador PID, foi desenvolvido um script em Python, o `plot_log.py`. Esta ferramenta permite analisar os arquivos de log `.TXT` gerados pelo robô, visualizar o desempenho em gráficos detalhados e, mais importante, simular o comportamento do robô com diferentes constantes PID sem a necessidade de realizar um novo percurso físico. O script recalcula a saída do PID com base no histórico de erros e nos novos ganhos fornecidos pelo usuário, oferecendo também sugestões automáticas para otimização.

7.1 Função `calculate_pid_values()`

Esta é a função central da simulação. Ela itera sobre cada registro de erro do arquivo de log e recalcula, linha por linha, quais teriam sido os termos Proporcional, Integral e Derivativo e a saída final do PID se as novas constantes `Kp`, `Ki` e `Kd` tivessem sido usadas. A lógica, incluindo a restrição do termo integral, espelha exatamente o comportamento do microcontrolador.

```
1 def calculate_pid_values(df, Kp, Ki, Kd):
2     pid_values = []
3     I = 0
4     erroAnterior = 0
5     OFFSET = 0
6 
7     for erro in df['Error']:
8         P = erro
9         I = I + P
```

```
        # Constrain the integral term, similar to the Arduino code
        I = max(-255, min(255, I))
        D = erro - erroAnterior

        p_term_val = Kp * P
        i_term_val = Ki * I
        d_term_val = Kd * D

        PID = p_term_val + i_term_val + d_term_val + OFFSET
        pid_values.append(PID)

        erroAnterior = erro

    return pd.DataFrame({'PID': pid_values})
```

Listing 28: Recalcula a saída do PID com novas constantes.

7.2 Função `plot_data()`

Esta função é responsável por gerar a visualização gráfica dos dados. Ela cria uma janela com quatro gráficos sobrepostos:

- **Erro vs. Tempo**: Mostra o desvio do robô em relação à linha, incluindo marcadores para os desafios da competição. Uma curva de erro suavizada (usando um filtro Savitzky-Golay) é sobreposta para mostrar a tendência geral do percurso.
- **Erro Suavizado vs. Tempo**: Apresenta a curva de erro suavizada para uma visualização mais clara da trajetória.
- **Saída do PID vs. Tempo**: Exibe o valor de correção calculado pelo PID.
- **Velocidade dos Motores vs. Tempo**: Mostra como a saída do PID foi traduzida para a velocidade de cada motor.

```
def plot_data(df, Kp, Ki, Kd):
    fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(14, 12), sharex=True)
    fig.suptitle(f"Analysis of PID Control with Kp={Kp}, Ki={Ki}, Kd={Kd}", fontsize=16)

```

```
    try:
        window_length = min(51, len(df['Error']))
        if window_length % 2 == 0:
            window_length -= 1

        if window_length > 3:
             df['Smoothed_Error'] = savgol_filter(df['Error'], window_length, 3)
        else:
             df['Smoothed_Error'] = df['Error']
    except Exception:
        df['Smoothed_Error'] = df['Error']

    ax1.plot(df['Time'], df['Error'],  marker='o', label='Error (Actual Path)', color='blue', alpha=0.4)
    ax1.plot(df['Time'], df['Smoothed_Error'], label='Smoothed Trend', color='green', linewidth=2.5)
    ax1.axhline(0, color='k', linestyle='--', linewidth=1, label='Ideal Path (Error=0)')

    if 'Challenge' in df.columns and not df[df['Challenge'] != 0].empty:
        challenges = df[df['Challenge'] != 0]
        plotted_labels = set()

        for _, row in challenges.iterrows():
            challenge_code = int(row['Challenge'])
            if challenge_code in challenge_map:
                props = challenge_map[challenge_code]
                label = props['label']

                current_label = label if label not in plotted_labels else ""

                ax1.scatter(row['Time'], row['Error'],
                            label=current_label,
                            color=props['color'],
                            marker=props['marker'],
                            s=props['size'],
                            zorder=5)
                plotted_labels.add(label)

    ax1.set_title('Error vs Time')
    ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))

    ax2.plot(df['Time'], df['Smoothed_Error'], label='Smoothed_Error', color='green', linewidth=2.5)
    ax2.set_title('Smoothed_Error vs Time')

```

```
    ax3.plot(df['Time'], df['PID'], label='Total PID Output',
    color='red')
    ax3.set_title('PID Output vs Time')

    ax4.plot(df['Time'], df['Velocidade Direita'], label='
   Velocidade Direita', color='purple')
    ax4.plot(df['Time'], df['Velocidade Esquerda'], label='
   Velocidade Esquerda', color='orange')
    ax4.set_title('Velocidade dos Motores vs. Tempo')

    plt.tight_layout(rect=[0, 0.03, 0.85, 0.95])
    plt.show()
```

Listing 29: Gera os gráficos de análise de desempenho.

7.3 Função `analyze_performance_and_suggest_pid()`

Após a visualização dos gráficos, esta função realiza uma análise quantitativa do desempenho do robô. Ela calcula métricas como o erro médio absoluto (MAE), o erro máximo (overshoot) e o número de oscilações (usando detecção de picos). Com base nessas métricas, ela imprime no terminal um diagnóstico do comportamento do robô e sugere ações claras para o ajuste dos ganhos PID, automatizando o processo de calibração.

```
def analyze_performance_and_suggest_pid(df, Kp, Ki, Kd):
    print("\n--- An lise de Desempenho & Sugest es de
   Ajuste ---")

    error = df['Error'].to_numpy()
    mae = np.mean(np.abs(error))
    max_abs_error = np.max(np.abs(error))

    peaks, _ = find_peaks(error, prominence=0.5)
    troughs, _ = find_peaks(-error, prominence=0.5)
    num_oscillations = len(peaks) + len(troughs)

    print(f"Erro M dio Absoluto (MAE): {mae:.4f}")
    print(f"M ximo Erro Absoluto (Overshoot/Undershoot): {
   max_abs_error:.4f}")
    print(f"Detectadas {num_oscillations} oscila  es
   significativas.")

    print("\nSugest es:")
    suggestion_made = False

    if num_oscillations > 8:
```

```
20          print("- Comportamento Detectado: O rob  oscila
    muito...")
21          print(f"  - A   o Recomendada: Diminua o valor de Kp
     (atual: {Kp}).")
22          suggestion_made = True
23
24      # ... (l gica para outras sugest es) ...
25
26      if not suggestion_made:
27          print("- O desempenho parece razo vel.")
```

Listing 30: Analisa métricas e sugere ajustes para o PID.

7.4 Exemplo de Saída Gráfica

A Figura 1 ilustra a janela de análise gerada pelo script `plot_log.py`. É possível observar o comportamento do erro (gráfico superior), a saída do controlador PID (gráfico intermediário) e a velocidade resultante dos motores (gráfico inferior). Os marcadores coloridos no primeiro gráfico indicam os exatos momentos em que eventos da competição ocorreram, como a detecção de uma $`linha_invertida`(quadradoazul), amanobrada`FaixadePedestre`(quadradolaranja)ea`reade$

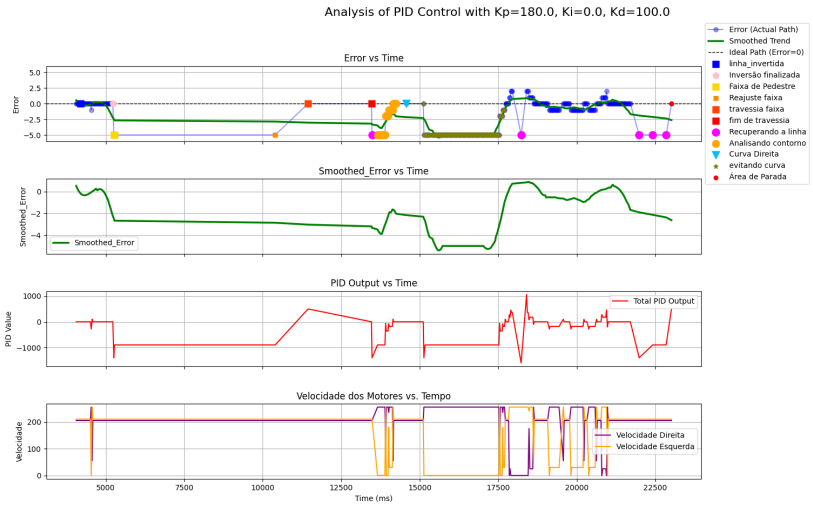

Figure 1: Exemplo da janela de análise gerada pelo script `plot_log.py`.

8 Referências Bibliográficas

8.1 Controle PID

O Controle PID (Proporcional Integral Derivativo) é usado para o controle de processos. O seu objetivo é apartir de equações matematicas que envolvem integrais e derivadas, minimizar o erro do sistema. O controle proporcional ajusta a variável de controle de forma proporcional ao erro. O controle integral ajusta a variável de controle baseando-se no tempo em que o erro acontece. O controle derivativo ajusta a variável de controle tendo como base a taxa de variação do erro.

Equação do controle PID:

$$PID(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{d}{dt} e(t) \tag{1}$$

Nessa equação, temos os seguintes parâmetros:

- K_p: ganho proporcional;
- K_i: ganho integral;
- K_d: ganho derivativo;
- e: erro instantâneo (diferença entre o valor desejado e o valor medido);
- τ: variável de integração;
- t: tempo atual.

Abaixo há uma tabela explicando como se deve ajustar os parametros do PID conforme o erro que o robô apresenta.

Comportamento do Robô	Parâmetro a Ajustar	Ação Recomendada
Oscila muito em torno da linha (zig-zag)	Proporcional (Kp)	Diminua o valor de Kp. Um Kp muito alto causa reações exageradas.
Corrige devagar e não consegue seguir bem a linha	Proporcional (Kp)	Aumente o valor de Kp. Um Kp muito baixo gera correções fracas.
Erro pequeno persiste por muito tempo	Integral (Ki)	Aumente Ki para corrigir erros acumulados lentamente.
Começa a oscilar após um tempo seguindo bem	Integral (Ki)	Diminua Ki. Um Ki alto pode acumular erro demais e causar instabilidade.
Reação muito abrupta a mudanças rápidas na linha	Derivativo (Kd)	Aumente Kd para suavizar a resposta. Ele ajuda a "frear" mudanças bruscas.
Resposta lenta a mudanças rápidas	Derivativo (Kd)	Diminua Kd se o robô demorar demais para reagir a curvas ou desvios.

Table 1: Ajuste dos parâmetros do PID com base no comportamento do robô

9 Conclusão

Este documento apresenta a estrutura e as funcionalidades do código, facilitando o entendimento do sistema e promovendo a democratização do mesmo. Ele proporciona liberdade e autonomia para a realização de testes e a utilização do código em diferentes contextos, permitindo uma maior flexibilidade no desenvolvimento e aprimoramento do projeto.