Documentação do Código do Robô Tartaruga Linhas

Competição CoRA 2025 - UFMG

Lucas Lemos Ricaldoni

Agosto
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.

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.

- Configura os pinos dos sensores de linha e dos LEDs como entradas e saídas.
- Inicializa a comunicação serial (se o modo de depuração estiver ativo) e 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.
- Inicializa o cartão SD para registro de logs (se o modo de log SD estiver ativo).
- Acende os LEDs frontais para indicar que a inicialização foi concluída.

```
void setup() {
  // Initialize serial communication
  if (debugMode) Serial.begin(9600);

  // 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(LED_LEFT, OUTPUT);
  pinMode(LED_RIGHT, OUTPUT);

  // Gyroscope initialization
  Wire.begin();
  mpu.begin();
  // liga os motores
  setup_motor();

  // Calibrate the gyroscope
  gyro_bias_z = calibrate_gyro();

  if (debugSD) setup_sd();
  // liga os leds da cara para indicar que o robo iniciou
  digitalWrite(LED_LEFT, HIGH);
  digitalWrite(LED_RIGHT, 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 de testes anteriores, a função gera um nome de

arquivo único a cada inicialização, baseado no valor da constante `Kp` e em um índice sequencial (ex: `L95_0.TXT`, `L95_1.TXT`). Se o arquivo for criado com sucesso, um cabeçalho ("Time,Error,Challenge") é escrito.

```
void setup_sd() {
  if (!SD.begin(chipSelect)) {
    if (debugMode) Serial.println("SD Card initialization
   failed!");
    digitalWrite(LED_LEFT, HIGH);
    ledLigado = true;
    tempoLedLigou = millis();
    return;
  }

  String baseName = "L" + String((int)Kp);
  String newFileName;

  for (int i = 0; i < 100; i++) {
    newFileName = baseName + "_" + String(i) + ".TXT";
    if (!SD.exists(newFileName)) {
      break;
    }
  }

  logFile = SD.open(newFileName, FILE_WRITE);
  if (logFile) {
    logFile.println("Time,Error");
    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 aberto. A cada chamada, a função registra o tempo atual da placa (‘millis()‘), o valor do erro calculado e um marcador de desafio opcional. O uso de ‘logFile.flush()‘ garante que os dados sejam gravados imediatamente no cartão, evitando perdas em caso de desligamento súbito.

```
void write_sd(int challenge_marker = 0) {
  if (logFile) {
    logFile.print(millis());
    logFile.print(",");
    logFile.print(erro);
    logFile.print(",");
    logFile.println(challenge_marker);
    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 verifica a ocorrência de uma inversão de pista (faixa de pedestres).

```
void calcula_erro() {
  // Update sensor values
  ler_sensores();
  // Check for inversion, signaling a pedestrian crossing
  if (verifica_inversao(SENSOR, SENSOR_CURVA)) {
    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) {
    erro = LINHA_NAO_DETECTADA;
  } else {
    int sensoresInativos = QUANTIDADE_TOTAL_SENSORES -
   sensoresAtivos;
    erro = somatorioErro / sensoresInativos;
  }
}
```

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.

```
void calcula_PID() {
  // Initialize variables for calculation
  PID = 0;
  P = erro;
```

```
  I = constrain(I + P, -255, 255);
  D = erro - erroAnterior;
  // Calculate PID
  PID = (Kp * P) + (Ki * I) + (Kd * D) + OFFSET;
  // Update the previous error value
  erroAnterior = erro;
}
```

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 `contaMarcacao()`

Realiza a contagem de marcadores (quadrados pretos) na pista.

- Incrementa um contador na transição de BRANCO para PRETO (borda de subida), garantindo que cada marcador seja contado apenas uma vez.
- Uma flag de controle (`jaContou`) evita múltiplas contagens do mesmo marcador.

```
int contaMarcacao(int estadoSensor, int contagemAtual, bool &
    jaContou) {
  if (estadoSensor == PRETO && !jaContou) {
    // Activate the lock to prevent recounting
    jaContou = true;
    return contagemAtual + 1;
  } else if (estadoSensor == BRANCO) {
    // Reset the lock when white is seen
    jaContou = false;
  }
  // Return the count unchanged
  return contagemAtual;
}
```

Listing 9: Contagem de marcadores de pista.

3.10 Função Principal `loop()`

Este é o ciclo principal do programa, que é executado continuamente. A lógica principal do robô reside aqui, tratando os diferentes cenários da competição.

3.10.1 Lógica de Competição

- **Controle do LED**: Gerencia o tempo que os LEDs de inicialização permanecem acesos.
- **Log de Dados**: Se habilitado, grava os dados de telemetria no cartão SD a cada ciclo.

- **Detecção de Curvas**: O robô primeiro verifica se há uma curva de 90 graus. Se houver, ele entra em um sub-loop para contar os marcadores de pista e decidir qual manobra executar.
  - **1 Marcador**: Executa uma curva simples de 90 graus.
  - **2 Marcadores**: Executa uma manobra de marcha à ré.
  - **3 ou mais Marcadores**: Entra em uma rotatória, usando o número de marcadores para determinar a saída correta.
- **Reset de Contadores**: Após cada manobra especial, os contadores de marcação são resetados para o estado inicial.

3.10.2 Tratamento de Exceções

- **Perda de Linha**: Se o robô perde a linha (`erro == LINHA_NAO_DETECTADA`), um contador ('contadorLinhaPerdida') é incrementado. Apenas se esse contador atingir um limite de tolerância ('LIMITE$_TOLERANCIA_LINHA_PERDIDA$'), ar
$Umavezativada, arotinaprimeiroverificaseacausafoiumafaixadepedestreseaexecuta.Ca$

- **Seguir Linha Padrão**: Se a linha é detectada normalmente, o contador de linha perdida é zerado, e o robô segue a linha utilizando o controle PID.

3.10.3 Modo de Depuração

Se a flag `debugMode` estiver ativa, a lógica principal é ignorada. O robô pode executar um teste simples de motores ou imprimir continuamente os dados dos sensores no monitor serial para fins de calibração e análise.

```
65 void loop() {
66   if (ledLigado) {
67     if (millis() - tempoLedLigou >= TEMPO_MAX_LED_L GADO) {
68       digitalWrite(LED_LEFT, LOW);
69       digitalWrite(LED_RIGHT, LOW);
70       ledLigado = false;
71     }
72   }
73
74   if (!debugMode) {
75     calcula_erro();
76     if (debugSD) write_sd();
77
78     int saidaCurva = verifica_curva_90(SENSOR, SENSOR_CURVA);
```

```
79      if (saidaCurva != CURVA_NAO_ENCONTRADA) {
80        while(erro != LINHA_NAO_DETECTADA) {
81          ler_sensores();
82          marcacoesEsquerda = contaMarcacao(SENSOR_CURVA[0],
    marcacoesEsquerda, jaContouEsquerda);
83          marcacoesDireita = contaMarcacao(SENSOR_CURVA[1],
    marcacoesDireita, jaContouDireita);
84          calcula_erro();
85          calcula_PID();
86          ajusta_movimento();
87        }
88
89        if (marcacoesEsquerda == 1 || marcacoesDireita == 1) {
90          turn_90(saidaCurva);
91          marcacoesEsquerda = 0; jaContouEsquerda = false;
92          marcacoesDireita = 0; jaContouDireita = false;
93        } else if ((marcacoesEsquerda > 1 && marcacoesEsquerda
    <= 2)
94        || (marcacoesDireita > 1 && marcacoesDireita <= 2)) {
95          saidaCurva = determina_saida_curva(marcacoesEsquerda,
     marcacoesDireita);
96          realiza_marcha_re(saidaCurva);
97          marcacoesEsquerda = 0; jaContouEsquerda = false;
98          marcacoesDireita = 0; jaContouDireita = false;
99        } else{
100         saidaCurva = determina_saida_curva(marcacoesEsquerda,
     marcacoesDireita);
101         int numeroDeMarcas = (saidaCurva == SAIDA_ESQUERDA) ?
     marcacoesEsquerda : marcacoesDireita;
102         realiza_rotatoria(saidaCurva,
    determina_saida_rotatoria(saidaCurva, numeroDeMarcas));
103         marcacoesEsquerda = 0; jaContouEsquerda = false;
104         marcacoesDireita = 0; jaContouDireita = false;
105       }
106       stop_motors();
107     } else {
108       if (erro == LINHA_NAO_DETECTADA) {
109         contadorLinhaPerdida++;
110         if (contadorLinhaPerdida >=
    LIMITE_TOLERANCIA_LINHA_PERDIDA) {
111           PID = 0;
112           stop_motors();
113
114           if (faixa_de_pedestre) {
115             realiza_faixa_de_pedestre();
116             faixa_de_pedestre = false;
117             contadorLinhaPerdida = 0;
118           } else {
119             unsigned long tempoPerdido = millis();
```

```
120             bool linhaEncontradaRe = false;
121
122             run_backward(velocidadeBaseDireita,
    velocidadeBaseEsquerda);
123             while (millis() - tempoPerdido <
    TIME_WITHOUT_LINE) {
124               ler_sensores();
125               if (calcula_sensores_ativos(SENSOR) > 0) {
126                 stop_motors();
127                 linhaEncontradaRe = true;
128                 contadorLinhaPerdida = 0;
129                 break;
130               }
131               delay(5);
132             }
133             if (!linhaEncontradaRe) {
134                 stop_motors();
135                 digitalWrite(LED_LEFT, HIGH);
136                 digitalWrite(LED_RIGHT, HIGH);
137                 if (debugSD) write_sd();
138                 while(true);
139             }
140           }
141         } else {
142           ajusta_movimento();
143         }
144       } else {
145         contadorLinhaPerdida = 0;
146         calcula_PID();
147         ajusta_movimento();
148       }
149     }
150   }
151   else {
152       if (debugMotor) {
153         // ... (c digo de teste de motor)
154       } else {
155         ler_sensores();
156         imprime_serial();
157       }
158   }
159   delay(5);
160 }
```

Listing 10: 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, como curvas, faixas de pedestre, marcha à ré e rotatórias. Ele também inclui funções auxiliares para processamento de dados dos sensores e controle de movimento preciso com o giroscópio.

4.1 Função `calcula_sensores_ativos()`

Esta função auxiliar simplesmente conta quantos dos cinco sensores principais de linha estão atualmente sobre a cor preta (estado ativo).

```
int calcula_sensores_ativos(int SENSOR[]) {
  int sensoresAtivos = 0;
  // Calculates the number of active sensors
  for(int i = 0; i < 5; i++) {
    sensoresAtivos += SENSOR[i];
  }

  // Returns the number of active sensors
  return sensoresAtivos;
}
```

Listing 11: Calcula o número de sensores ativos.

4.2 Função `verifica_curva_90()`

Analisa a combinação de leituras dos sensores para determinar se o robô encontrou uma curva de 90 graus. A nova lógica considera uma curva quando 3 ou mais sensores de linha estão ativos, e então usa os sensores de curva laterais para definir a direção.

- Se 3 ou mais sensores de linha estão ativos e o sensor de curva direito detecta preto, retorna `CURVA_DIREITA`.
- Se 3 ou mais sensores de linha estão ativos e o sensor de curva esquerdo detecta preto, retorna `CURVA_ESQUERDA`.
- Se os sensores de linha indicam uma curva, mas os sensores laterais não, a direção é considerada incerta (`CURVA_EM_DUVIDA`).

```
int verifica_curva_90(int SENSOR[], int SENSOR_CURVA[]) {
  if (calcula_sensores_ativos(SENSOR) >= 3) {
    if (SENSOR_CURVA[0] == BRANCO && SENSOR_CURVA[1] ==
   BRANCO) {
      return CURVA_EM_DUVIDA;
    }
    if (SENSOR_CURVA[0] == BRANCO && SENSOR_CURVA[1] == PRETO
   ) {
      return CURVA_DIREITA;
    }
    if (SENSOR_CURVA[0] == PRETO && SENSOR_CURVA[1] == BRANCO
   ) {
      return CURVA_ESQUERDA;
    }
  }

  return CURVA_NAO_ENCONTRADA;
}
```

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

4.3 Função `turn_90()`

Executa a manobra de curva de 90 graus. A função primeiro avança em linha reta enquanto os sensores ainda detectam a marcação de curva (ou até um tempo limite ser atingido), para melhor se posicionar. Em seguida, para os motores para garantir estabilidade e utiliza a função `turn_until_angle` para realizar a rotação precisa.

```
void turn_90(int curvaEncontrada) {

  unsigned long startTime = millis();
  // This timeout prevents the robot from getting stuck
  while (calcula_sensores_ativos(SENSOR) >= 3 && (millis() -
   startTime < TIMEOUT_90_CURVE)) {
    run(velocidadeBaseDireita, velocidadeBaseEsquerda);
    ler_sensores();
  }

  // Stop the car for greater stability
  stop_motors();
  delay(200);

  if (curvaEncontrada == CURVA_ESQUERDA) {
    digitalWrite(LED_LEFT, HIGH);
    turn_left(velocidadeBaseDireita, velocidadeBaseEsquerda);
```

```
    turn_until_angle(ANGLE_CURVE);
    digitalWrite(LED_LEFT, LOW);
  } else if (curvaEncontrada == CURVA_DIREITA) {
    digitalWrite(LED_RIGHT, HIGH);
    turn_right(velocidadeBaseDireita, velocidadeBaseEsquerda)
    ;
    turn_until_angle(ANGLE_CURVE);
    digitalWrite(LED_RIGHT, LOW);
  }
}
```

Listing 13: Executa uma curva de 90 graus.

4.4 Função `calibrate_gyro()`

Esta função é essencial e deve ser chamada durante a inicialização (`setup`). Ela calibra o giroscópio medindo o seu desvio (bias) enquanto o robô está parado. A média de múltiplas leituras é calculada e armazenada para corrigir as medições futuras da velocidade angular.

```
float calibrate_gyro(int samples = 200) {
    if (debugMode) Serial.println("Calibrating gyroscope...
   Keep the robot stationary.");

    float sum_gz = 0;
    for (int i = 0; i < samples; i++) {
        mpu.update();
        sum_gz += mpu.getGyroZ();
        delay(10);
    }

    float bias_gz = sum_gz / samples;

    if (debugMode) {
      Serial.print("Calibration complete. Gyroscope Bias (Gz)
    = ");
      Serial.println(bias_gz, 4);
    }

    return bias_gz;
}
```

Listing 14: Calibra o giroscópio.

4.5 Função `turn_until_angle()`

Realiza uma rotação precisa do robô até que um ângulo alvo seja atingido. A função integra a velocidade angular (já corrigida pelo bias) ao longo do tempo para calcular o ângulo atual. Os motores são parados assim que o ângulo desejado é alcançado.

```
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;

    if (debugMode) Serial.println(angular_velocity_z);
    delay(10);
  }
  stop_motors();

  if (debugMode) Serial.println("Rotation finished");
}
```

Listing 15: Gira o robô até um ângulo específico.

4.6 Funções de Inversão de Pista

4.6.1 Função `inverte_sensor()`

Uma função simples que inverte a lógica de um sensor. Se a entrada for 1 (preto), retorna 0 (branco), e vice-versa. É a base para permitir que o robô siga uma linha invertida.

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

```
}
```

Listing 16: Inverte o valor de um sensor.

4.6.2 Função `verifica_inversao()`

Esta função detecta se ocorreu uma inversão de cores na pista (ex: de pista branca com linha preta para pista preta com linha branca). Na implementação atual, ela considera que uma inversão ocorreu se apenas um sensor principal estiver ativo. Se detectada, ela modifica os valores do array `SENSOR` para que o robô possa continuar seguindo a linha.

```
bool verifica_inversao(int SENSOR[], int SENSOR_CURVA[]) {
  if (calcula_sensores_ativos(SENSOR) == 1) {
    // Invert the state of the sensors
    for (int i = 0; i < 5; i++) {
      SENSOR[i] = inverte_sensor(SENSOR[i]);
    }
    // Return true to indicate an inversion occurred
    return true;
  }
  // Return false as no inversion was detected
  return false;
}
```

Listing 17: Verifica e trata a inversão de cores da pista.

4.7 Funções de Desafios Específicos

4.7.1 Função `realiza_faixa_de_pedestre()`

Implementa a lógica para o desafio da faixa de pedestres. O robô para, aguarda por um tempo determinado (6 segundos) e então avança para cruzar a faixa.

```
void realiza_faixa_de_pedestre() {
  // Wait for the minimum time of 6 seconds to cross
  delay(6000);
  // Move forward to cross the track
  run(velocidadeBaseDireita, velocidadeBaseEsquerda);
  delay(2000);
}
```

Listing 18: Executa o desafio da faixa de pedestres.

4.7.2 Função `realiza_marcha_re()`

Executa a sequência de ações para o desafio da marcha à ré. O robô para, move-se para trás por um segundo, para novamente e então realiza uma curva de 90 graus para o lado que foi determinado pela contagem de marcadores.

```
void realiza_marcha_re(int lado_da_curva) {
  stop_motors();
  delay(500);

  // 2. Execute a re por um tempo fixo
  run_backward(velocidadeBaseDireita, velocidadeBaseEsquerda);
  delay(1000);

  // 3. Pare novamente
  stop_motors();
  delay(500);

  // 4. Turn to the side of the second marker, as per the rules
  if (lado_da_curva == SAIDA_DIREITA) {
    turn_right(velocidadeBaseDireita, velocidadeBaseEsquerda);
    turn_until_angle(90);
  } else {
    turn_left(velocidadeBaseDireita, velocidadeBaseEsquerda);
    turn_until_angle(90);
  }

  stop_motors();
}
```

Listing 19: Executa o desafio de marcha à ré.

4.7.3 Função `determina_saida_curva()`

Usada nos desafios de marcha à ré e rotatória, esta função compara a contagem de marcadores da esquerda e da direita. A regra é que o robô deve sair pelo lado que apresentou menos marcadores.

```
int determina_saida_curva(int marcacoesEsquerda, int marcacoesDireita) {
  if (marcacoesDireita < marcacoesEsquerda) {
    return SAIDA_DIREITA;
  } else {
```

```
    return SAIDA_ESQUERDA;
  }
}
```

Listing 20: Determina a saída baseada na contagem de marcadores.

4.7.4 Função `determina_saida_rotatoria()`

Define qual saída da rotatória o robô deve tomar com base no número total de marcas contadas antes de entrar no desafio.

- 2 marcas: 1ª saída.
- 3 marcas: 2ª saída.
- 4 ou mais marcas: 3ª saída.

```
int determina_saida_rotatoria(int saidaCurva, int
    numeroDeMarcas) {
  if (numeroDeMarcas == 2) {
    saidaDesejada = 1; // 1a Saida
  } else if (numeroDeMarcas == 3) {
    saidaDesejada = 2; // 2a Saida
  } else if (numeroDeMarcas >= 4) {
    saidaDesejada = 3; // 3a Saida
  }

  return saidaDesejada;
}
```

Listing 21: Determina a saída da rotatória.

4.7.5 Função `realiza_rotatoria()`

Controla a navegação do robô dentro da rotatória. Ele entra na rotatória, segue a linha e conta as saídas disponíveis (detectadas pelos sensores de curva) até atingir o número da `saidaDesejada`. Ao encontrar a saída correta, ele executa uma curva para sair da rotatória.

```
void realiza_rotatoria(int saidaCurva, int saidaDesejada){
  // Initialize the current exit count
  int saidaAtual = 1;
```

```
4
5   // Perform a 90-degree turn to enter the roundabout
6   if (saidaCurva == CURVA_ESQUERDA) {
7     turn_left(velocidadeBaseDireita, velocidadeBaseEsquerda);
8     turn_until_angle(90);
9   } else if (saidaCurva == CURVA_DIREITA) {
10    turn_right(velocidadeBaseDireita, velocidadeBaseEsquerda)
      ;
11    turn_until_angle(90);
12  }
13
14  // Loop until the robot reaches the correct exit
15  while(saidaAtual != saidaDesejada) {
16    // Calculate error to stay on the line
17    calcula_erro();
18    ajusta_movimento();
19
20    // Check which side the exit should be on (based on
      global variable)
21    if (saida_rotatoria == SAIDA_ESQUERDA) {
22      // Check for a marker on the right (indicating an exit)
23      if (calcula_sensores_ativos(SENSOR) <= 3 &&
      SENSOR_CURVA[0] == PRETO && SENSOR_CURVA[1] == BRANCO) {
24        delay(200);
25        // Update the current exit count
26        saidaAtual++;
27      }
28    } else if(saida_rotatoria == SAIDA_DIREITA) {
29      // Check for a marker on the left (indicating an exit)
30      if (calcula_sensores_ativos(SENSOR) <= 3 &&
      SENSOR_CURVA[0] == BRANCO && SENSOR_CURVA[1] == PRETO) {
31        delay(200);
32        // Update the current exit count
33        saidaAtual++;
34      }
35    }
36  }
37  // Exit to the correct side of the roundabout
38  if (saida_rotatoria == SAIDA_ESQUERDA) {
39    turn_left(velocidadeBaseDireita, velocidadeBaseEsquerda);
40  } else {
41    turn_right(velocidadeBaseDireita, velocidadeBaseEsquerda)
      ;
42  }
43 }
```

Listing 22: Executa o desafio da rotatória.

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. A lógica de movimento é baseada no controle individual da velocidade de cada motor através de sinais PWM.

5.1 Função `setup_motor()`

Inicializa os pinos de controle dos motores como saídas. Esta função é chamada uma única vez no `setup()` principal do robô para garantir que os motores estejam prontos para receber comandos e que o robô comece em um estado parado.

```
void setup_motor() {
  pinMode(MOTOR_LEFT_CLKWISE, OUTPUT);
  pinMode(MOTOR_LEFT_ANTI, OUTPUT);
  pinMode(MOTOR_RIGHT_CLKWISE, OUTPUT);
  pinMode(MOTOR_RIGHT_ANTI, OUTPUT);
  // ensure the car starts off on 2 sec
  stop_motors();
  //delay(DELAY_TO_START);
}
```

Listing 23: Configuração inicial dos pinos dos motores.

5.2 Função `set_state_motor()`

Esta é a função central de controle dos motores. Ela recebe quatro valores de velocidade (PWM, de 0 a 255) e os aplica diretamente aos pinos de controle dos motores. Cada motor possui um pino para o sentido horário (frente) e um para o anti-horário (ré). Ao definir um valor PWM em um desses pinos, o motor gira na direção e velocidade correspondentes.

```
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);
```

```
5   analogWrite(MOTOR_RIGHT_ANTI, rightCcw);
6 }
```

Listing 24: Define o estado e a velocidade de cada motor.

5.3 Funções de Movimento

As funções a seguir utilizam `set_state_motor()` para realizar ações de movimento específicas.

5.3.1 Função `stop_motors()`

Para completamente o robô, enviando um valor de 0 para todos os pinos de controle dos motores.

```
1 void stop_motors() {
2   set_state_motor(0, 0, 0, 0);
3 }
```

Listing 25: Para todos os motores.

5.3.2 Função `run()`

Move o robô para a frente. A `velocityLeft` é aplicada ao pino do motor esquerdo no sentido horário, e a `velocityRight` ao pino do motor direito no sentido horário.

```
1 void run(int velocityRight, int velocityLeft) {
2   set_state_motor(velocityLeft, 0, velocityRight, 0);
3 }
```

Listing 26: Move o robô para frente.

5.3.3 Função `run_backward()`

Move o robô para trás, aplicando a velocidade aos pinos de sentido anti-horário de cada motor.

```
void run_backward(int velocityRight, int velocityLeft) {
  set_state_motor(0, velocityLeft, 0, velocityRight);
}
```

Listing 27: Move o robô para trás.

5.3.4 Função `turn_right()`

Executa uma curva para a direita no próprio eixo (curva de tanque). O motor esquerdo gira para frente enquanto o motor direito gira para trás.

```
void turn_right(int velocityRight, int velocityLeft) {
  set_state_motor(velocityLeft, 0, 0, velocityRight);
}
```

Listing 28: Gira o robô para a direita.

5.3.5 Função `turn_left()`

Executa uma curva para a esquerda no próprio eixo. O motor esquerdo gira para trás enquanto o motor direito gira para frente.

```
void turn_left(int velocityRight, int velocityLeft) {
  set_state_motor(0, velocityLeft, velocityRight, 0);
}
```

Listing 29: Gira o robô para a esquerda.

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

Este arquivo centraliza todas as constantes e variáveis globais utilizadas no projeto. A principal vantagem dessa abordagem é facilitar o ajuste de parâmetros (como ganhos do PID e velocidades) e o gerenciamento de estados do robô, sem a necessidade de alterar a lógica principal nos outros arquivos.

6.1 Pinos de Sensores e LEDs

Define as conexões físicas entre os componentes eletrônicos e os pinos do microcontrolador.

- **Sensores**: Mapeia cada um dos 7 sensores infravermelhos (5 para a linha e 2 para as curvas) para um pino digital ou analógico específico.
- **LEDs**: Mapeia os dois LEDs de indicação para seus respectivos pinos.

6.2 Variáveis de Estado dos Sensores

Arrays globais que armazenam as leituras mais recentes dos sensores.

- `SENSOR[5]`: Armazena o estado (PRETO/BRANCO) dos cinco sensores centrais de seguimento de linha.
- `SENSOR_CURVA[2]`: Armazena o estado dos dois sensores laterais, usados para detectar curvas e marcadores de desafios.

6.3 Configurações de Velocidade

Parâmetros que controlam a movimentação dos motores.

- `velocidadeBaseDireita` e `velocidadeBaseEsquerda`: Definem a velocidade padrão (valor PWM de 0-255) para cada motor. Os valores podem ser ligeiramente diferentes para compensar variações mecânicas e garantir que o robô ande em linha reta.
- `velocidadeDireita` e `velocidadeEsquerda`: Variáveis que armazenam a velocidade ajustada após a aplicação do controle PID.

6.4 Controle PID

Todas as constantes e variáveis relacionadas ao cálculo do controlador Proporcional-Integral-Derivativo.

- **Ganhos**: `Kp`, `Ki`, `Kd` são as constantes de ganho que determinam a intensidade da resposta do robô ao erro. O ajuste fino desses valores é crucial para o desempenho do seguidor de linha.
- **Variáveis de Cálculo**: `erro`, `erroAnterior`, `P`, `I`, `D` e `PID` são as variáveis utilizadas em tempo de execução para calcular a saída do controlador a cada ciclo.

Nota: O código também inclui constantes comentadas para um método de auto-ajuste (Ziegler-Nichols), que podem ser utilizadas como ponto de partida para a calibração.

6.5 Variáveis de Estado dos Desafios

Flags e contadores que ajudam o robô a saber em que parte de um desafio ele se encontra.

- `saida_rotatoria`: Indica a direção da rotatória.
- `faixa_de_pedestre`: Uma flag booleana que se torna verdadeira quando uma faixa de pedestres é detectada.
- `saidaDesejada`: Armazena o número da saída que o robô deve tomar em uma rotatória.

6.6 Controle de Depuração (Debug)

Flags booleanas que podem ser ativadas para auxiliar no desenvolvimento e teste.

- `debugMode`: Quando `true`, ativa a impressão de dados no monitor serial.
- `debugMotor`: Quando `true`, executa um teste simples de motores em vez da lógica principal.
- `debugSD`: Uma flag booleana que, quando `true`, habilita a inicialização e a gravação de dados no cartão SD.

6.7 Parâmetros de Tolerância e Tempo

- `LIMITE_TOLERANCIA_LINHA_PERDIDA`: Define quantas leituras consecutivas de "linha perdida" são toleradas antes de iniciar a rotina de recuperação.
- `TEMPO_MAX_LED_LIGADO`: Controla por quanto tempo os LEDs de inicialização ficam acesos.
- `TIMEOUT_90_CURVE`: Um tempo máximo de segurança para a manobra de avanço antes de uma curva de 90 graus, para evitar que o robô fique preso.
- `TIME_WITHOUT_LINE`: Define o tempo máximo que o robô tentará reencontrar a linha andando de ré antes de parar completamente.

6.8 Variáveis do Giroscópio e LEDs

- `gyro_bias_z`: Armazena o valor de calibração (bias) do giroscópio, essencial para obter medições de rotação precisas.
- **Controle de LED**: As variáveis `tempoLedLigou`, `TEMPO_MAX_LED_LIGADO` e `ledLigado` gerenciam o comportamento dos LEDs indicadores na inicialização do robô.

7 Referências Bibliográficas

7.1 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

7.2 Sensor de Refletância Infravermelho (TCRT5000)

Os "olhos" do robô para seguir a linha são os módulos sensores TCRT5000. Cada módulo contém um par de componentes: um diodo emissor de luz

(LED) infravermelha e um fototransistor, que funciona como um detector de luz infravermelha.

Princípio de Funcionamento:

- O LED emite um feixe constante de luz infravermelha, invisível ao olho humano, em direção à superfície da pista.
- **Superfície Branca:** Uma superfície clara, como a pista branca, reflete a maior parte da luz infravermelha de volta para o sensor. O fototransistor detecta essa alta reflexão, permitindo a passagem de corrente. No módulo TCRT5000, isso geralmente resulta em um nível de tensão **BAIXO (LOW)** na saída digital. No código, este estado é tratado como `BRANCO` (valor 0).
- **Superfície Preta:** Uma superfície escura, como a linha preta, absorve a maior parte da luz. Pouca ou nenhuma luz é refletida de volta. O fototransistor não detecta luz e não conduz corrente, resultando em um nível de tensão **ALTO (HIGH)** na saída. No código, este estado é tratado como `PRETO` (valor 1).

Dessa forma, ao ler o estado de vários desses sensores com a função `digitalRead()`, o robô consegue mapear a posição da linha preta sob ele e calcular o erro para se manter no trajeto.

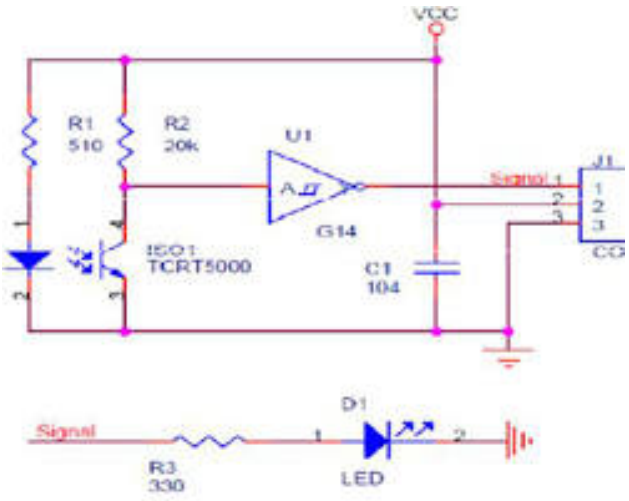

Figure 1: Princípio de funcionamento do sensor de refletância TCRT5000.

(Nota: Para a figura acima funcionar, encontre um diagrama ilustrativo do TCRT5000 e salve-o como `tcrt5000_diagrama.png` *na pasta do seu projeto.)*

7.3 Giroscópio e Unidade de Medição Inercial (IMU MPU6050)

Para realizar curvas com ângulos precisos, o robô utiliza o módulo MPU6050. Este componente é uma Unidade de Medição Inercial (IMU) que contém, entre outros sensores, um giroscópio de 3 eixos. Para este projeto, o eixo Z (eixo de rotação vertical) é o mais importante.

Princípio de Funcionamento:

- **Medição de Velocidade Angular:** Um giroscópio não mede diretamente o ângulo, mas sim a **velocidade angular** (em graus por segundo). Ele detecta o quão rápido o robô está girando em torno de um eixo.
- **Calibração e Bias:** Mesmo quando perfeitamente parado, o sensor do giroscópio apresenta pequenas leituras, um ruído conhecido como "bias" ou "drift". A função `calibrate_gyro()` do nosso código é executada com o robô parado para medir esse desvio médio. O valor do bias é então subtraído de todas as leituras futuras para garantir medições mais precisas.
- **Integração para Obter o Ângulo:** Para descobrir o ângulo total que o robô girou, o código realiza uma operação de integração. De forma simplificada, ele multiplica a velocidade angular (já corrigida pelo bias) pelo pequeno intervalo de tempo entre as leituras: $\Delta \hat{A}ngulo = VelocidadeAngular \times \Delta Tempo$. A função `turn_until_angle()` executa essa conta em um loop rápido, somando os pequenos incrementos de ângulo até que o ângulo total desejado seja atingido.

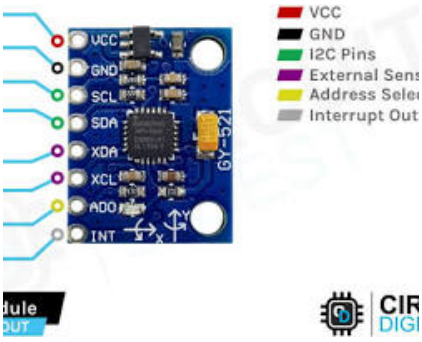

Figure 2: Módulo MPU6050, utilizado para medição de rotação.

7.4 Controle de Motores com Ponte H (Mini L298N)

Os pinos de um microcontrolador como o Arduino não conseguem fornecer a corrente elétrica necessária para alimentar os motores diretamente. Além disso, é preciso uma forma de inverter a direção de rotação dos motores. A solução para ambos os problemas é utilizar um **Driver de Motor**, e neste projeto foi usado o módulo Mini L298N, que contém um circuito do tipo **Ponte H (H-Bridge)**.

Princípio de Funcionamento:

- **Ponte H:** É um circuito eletrônico que permite controlar a direção do fluxo de corrente através de um motor. Imagine-a como quatro interruptores arranjados em formato de "H" com o motor no meio. Ao fechar dois interruptores na diagonal, a corrente flui em uma direção (motor gira para frente). Ao fechar a outra dupla na diagonal, a corrente flui no sentido oposto (motor gira para trás). As funções `run()`, `run_backward()`, etc., no arquivo `motors.cpp` controlam quais "interruptores" da Ponte H são ativados.
- **PWM (Pulse-Width Modulation):** Para controlar a **velocidade** do motor, o L298N utiliza um sinal PWM. Em vez de enviar uma tensão constante, o microcontrolador envia pulsos de energia em alta frequência. A velocidade é determinada pela largura desses pulsos (o "duty cycle"):
  - Pulsos mais largos (maior "duty cycle") significam mais tempo com energia, resultando em maior velocidade.

– Pulsos mais estreitos (menor "duty cycle") significam menos tempo com energia, resultando em menor velocidade.

A função `analogWrite(pino, valor)` do Arduino gera exatamente esse sinal PWM, onde o `valor` (de 0 a 255) define a largura do pulso e, consequentemente, a velocidade do motor.

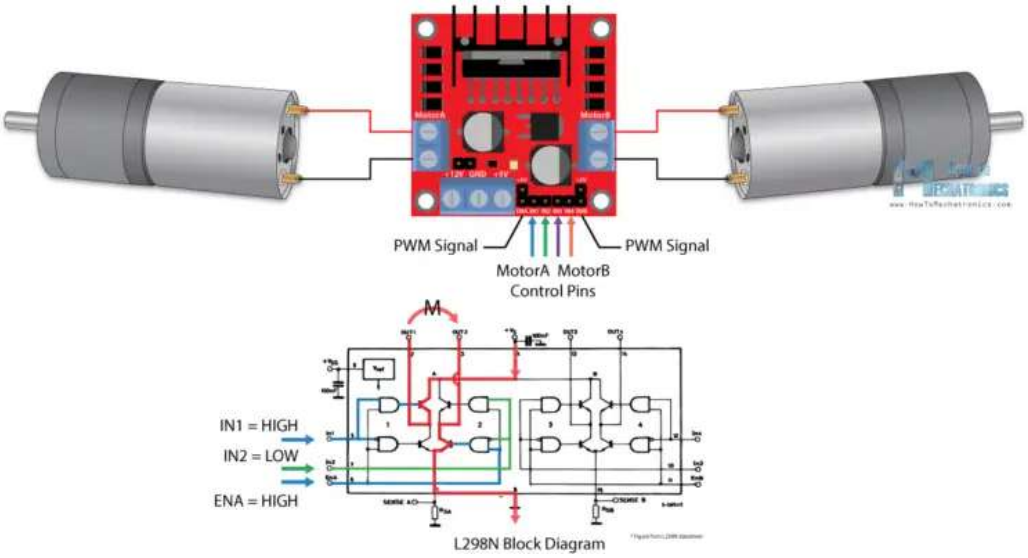

Figure 3: Ilustração do conceito de PWM para controle de velocidade.

8 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.