Code Documentation for the Turtle Lines Robot

CoRA 2025 Competition - UFMG

Lucas Lemos Ricaldoni

September
2025

Contents

1 Introduction

This document presents the documentation for the code developed for the "Turtle Lines" line-following robot, designed for the CoRA 2025 competition at UFMG. The system is based on a microcontroller and uses infrared sensors for track detection and a gyroscope to assist with navigation.

2 Dependencies

The code was developed in C++ for the Arduino platform. The following libraries are required for the project to compile and run:

- `MPU6050.h`: For communication with the MPU6050 gyroscope and accelerometer.
- `Wire.h`: For I2C communication with the MPU6050 sensor.
- `SPI.h` and `SD.h`: For communication with the SD card module.

3 Main Code Structure: `CoRA2025.ino`

The `CoRA2025.ino` file is the heart of the system, responsible for orchestrating all the robot's operations, from reading sensors to controlling motors and executing the competition's challenge logic.

3.1 Function `setup()`

This function is executed once when the robot is powered on or reset. Its main purpose is to initialize all the necessary hardware and software components for operation.

- Initializes serial communication and the SD card (if the respective debug modes are active).

- Configures the pins for the line sensors and the LED as inputs and outputs.
- Initializes I2C communication (Wire) for the gyroscope.
- Starts and calibrates the gyroscope (MPU6050) to obtain the bias value.
- Configures the motors for motion control.
- Turns on the front LED to indicate that initialization is complete.

```
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();

  // turn on motors
  setup_motor();

  // Calibrate the gyroscope
  gyro_bias_z = calibrate_gyro();
  // turn on the face LEDs to indicate the robot has started
  digitalWrite(LEDS, HIGH);
  tempoLedLigou = millis();
  ledLigado = true;
}
```

Listing 1: Robot initialization function.

3.2 Function `setup_sd()`

This function initializes the SD card and creates a new log file. To avoid overwriting data, the function generates a unique file name at each startup,

using the value of the `Kp` constant and a sequential index (e.g., `L180_0.TXT`, `L180_1.TXT`). If the file is created successfully, a detailed header is written to facilitate later data analysis.

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

  char newFileName[16];

  // Search for the first available index
  for (int i = 0; i < 100; i++) {
    // Format the file name safely using snprintf
    snprintf(newFileName, sizeof(newFileName), "L%d_%d.TXT", (int)Kp, i);
    // Check if the file does NOT exist
    if (!SD.exists(newFileName)) {
      break;
      // Found an available name
    }
  }

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

  // Open the new file for writing
  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: SD card initialization and log file creation.

3.3 Function `write_sd()`

Responsible for writing a line of data to the log file. With each call, the function records a complete set of telemetry data, including the time (‘millis()‘), error, a challenge marker, count and time of markings, motor speeds, and the state of all seven sensors. Using ‘logFile.flush()‘ ensures that the data is written to the card immediately.

```
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: Writes telemetry data to the SD card.

3.4 Function `ler_sensores()`

Responsible for reading the digital state of the line sensors and the curve sensors.

- Updates the global arrays `SENSOR` and `SENSOR_CURVA` with the read values.
- A value of `PRETO` (1) indicates that the sensor has detected the line, while `BRANCO` (0) indicates the opposite.

```
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: Reading the line and curve sensors.

3.5 Function `ajusta_movimento()`

Adjusts the motor speeds based on the output value of the PID controller.

- Calculates the speed for the right and left motors by subtracting and adding the PID value to the base speed, respectively.
- Uses the `constrain` function to ensure that the speed values remain within the valid range (0 to 255).
- Actuates the motors with the new calculated speeds.

```
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: Adjusting the motor speed.

3.6 Function `calcula_erro()`

Calculates the robot's position error relative to the line.

- The error is calculated using a weighted average of the readings from the 5 line sensors.
- The result indicates the distance and direction of the robot's deviation from the center of the line.
- If all sensors are on the black color, the function considers that the line has been lost.
- Before the calculation, the function calls `verifica_inversao()` and, if an inversion is completed, activates the `faixa_de_pedestre` flag.

```
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) {
```

```
    erro = LINHA_NAO_DETECTADA;
  } else {
    int sensoresInativos = QUANTIDADE_TOTAL_SENSORES -
   sensoresAtivos;
    erro = somatorioErro / sensoresInativos;
  }
}
```

Listing 6: Calculation of the position error.

3.7 Function `calcula_PID()`

Implements the PID (Proportional-Integral-Derivative) controller.

- **Proportional (P)**: Reacts proportionally to the current error.
- **Integral (I)**: Accumulates the error over time to correct persistent deviations.
- **Derivative (D)**: Responds to the rate of change of the error to dampen oscillations.
- The final PID value is the weighted sum of these three components and is used to adjust the motor speeds.

```
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: Calculation of the PID control.

3.8 Function `imprime_serial()`

Sends debugging information to the serial monitor.

- Prints the state of the curve and line sensors.
- Invokes the error and PID calculation functions and prints their results, as well as the resulting motor speeds.

```
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: Printing debug data.

3.9 Main Function `loop()`

This is the main loop of the program, executed continuously. The new logic is a more sophisticated state machine to handle the various competition scenarios.

3.9.1 Competition Logic

- **Drag Race Mode**: If `arrancadaMode` is true, the robot performs a simple line following without the challenge logic.

- **Competition Mode**:
  1. **Reading and Detection**: In each cycle, the sensors are read, and it is checked if there is a 90-degree curve.
  2. **Pedestrian Crossing**: It is detected after a track inversion. The robot waits for a confirmation (counting 3 readings of "all sensors on black") before executing the stop maneuver.
  3. **Line Following**: If no curve is detected and the line is visible, the robot follows the track normally using PID control. Line recovery attempts are reset after some time if the robot remains stable.
  4. **Curve/Challenge Detection**: If a 90-degree curve is detected, the robot analyzes the side markings to decide the maneuver.
     - **Simple Curve**: If one or more markings are seen on only one side, it executes a 90-degree turn to that side.
     - **Reverse vs. Intersection**: If markings are seen on both sides, the robot measures the time difference between their detection. A significant time difference indicates a reverse maneuver. If the time is very close, it indicates a doubtful intersection.

3.9.2 Exception Handling (Line Loss)

- **Tolerance**: If the line is lost, a counter is started. Action is only taken after a number of cycles (`LIMITE_TOLERANCIA_LINHA_PERDIDA`) to ignore momentary failures.
- **Recovery**: The robot executes the `tenta_recuperar_linha()` function. If successful, it increments an attempt counter.
- **Stop Area**: If the recovery fails, or if the number of recoveries in a short period exceeds the limit (`LIMITE_TENTATIVAS_RECUPERACAO`), the robot stops completely in the stop area, considering the course finished.

3.9.3 Debug Mode

If the `debugMode` flag is active, the main logic is ignored, and the robot continuously prints sensor data to the serial monitor for calibration and analysis.

```
56 void loop() {
57   // check the state of the 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       // check if there is a 90-degree curve
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         // noise filter
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           // follow normally
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         // lost the line
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       // avoid activating at intersections
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: Main execution loop of the robot.

4 Challenges File: `challenges.cpp`

This file contains the logic for handling the specific challenges of the competition. It has been restructured to include a more robust state machine for detecting inversions and more refined logic for each maneuver.

4.1 Function `verifica_estado_led()`

A simple management function that checks if the LED's initialization time (`TEMPO_MAX_LED_LIGADO`) has passed and, if so, turns off the LED.

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

Listing 10: Checks and turns off the status LED after the defined time.

4.2 Function `calcula_sensores_ativos()`

This helper function counts how many of the five main line sensors are currently on the black color.

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

Listing 11: Calculates the number of active sensors.

4.3 Function `verifica_curva_90()`

Analyzes the combination of sensor readings to determine if the robot has encountered a 90-degree curve. The logic identifies a curve when at least 2 central sensors are active and one of the side curve sensors is also active.

```
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: Checks for the existence of a 90-degree curve.

4.4 Function `calcula_posicao()`

Calculates a weighted average position of the line based on active sensors, returning a value between -2 and 2. It is used to make small angle adjustments before a 90-degree turn.

```
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: Calculates the weighted position of the line.

4.5 Function `turn_90()`

Executes the 90-degree turn maneuver. The function moves forward until it passes the initial marking, stops, calculates an adjusted turn angle based on the line's position, and after rotating, moves forward again until it realigns on the track to avoid duplicate detections.

```
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);
    }

    stop_motors();
    delay(100);
    while (true) {
      calcula_erro();
      calcula_PID();
      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: Executes an improved 90-degree turn.

4.6 Function `calibrate_gyro()`

Calibrates the gyroscope by measuring its drift (bias) while the robot is stationary. The average of multiple readings is calculated and stored to correct future measurements.

```
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;
19 }
```

Listing 15: Calibrates the gyroscope.

4.7 Function `turn_until_angle()`

Performs a precise rotation of the robot until a target angle is reached, using the gyroscope.

```
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: Rotates the robot to a specific angle.

4.8 Function `inverte_sensor()`

Inverts the logic of a sensor (1 becomes 0, 0 becomes 1), allowing the robot to follow a line of inverted color.

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

Listing 17: Inverts the value of a sensor.

4.9 Function `verifica_inversao()`

Implements a state machine to detect and manage crossing a section of the track with inverted colors.

- **Entry**: Activates the inversion mode (`inversaoAtiva`) when only 1 line sensor is active.
- **During Inversion**: Inverts the reading of all sensors so that the line follower logic continues to work.
- **Exit**: Deactivates the inversion mode when the curve and line sensors indicate the end of the section. Upon exiting, it activates the `inversao_finalizada` flag to signal a possible pedestrian crossing.

```
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: Checks and handles the track color inversion.

4.10 Function `realiza_faixa_de_pedestre()`

Executes a robust routine for the pedestrian crossing. The robot stops for the defined time, reverses slightly to find the line again, realigns on it, and then moves forward at high speed to ensure a complete crossing.

```
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);

  // ... (Crossing finalization logic)

  inversao_finalizada = false;
}
```

Listing 19: Executes the pedestrian crossing challenge with realignment.

4.11 Function `realiza_marcha_re()`

Executes the reverse gear challenge. The robot stops, moves backward at high speed, stops again, moves forward a little to position itself, and then performs the 90-degree turn to the determined side.

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

```
5
6   run_backward(255, 255);
7   delay(1200);
8
9   stop_motors();
10  delay(500);
11
12  run(velocidadeBaseDireita, velocidadeBaseEsquerda);
13  delay(200);
14
15  turn_90(lado_da_curva);
16
17  stop_motors();
18  delay(500);
19 }
```

Listing 20: Executes the reverse gear challenge.

4.12 Function `realiza_rotatoria()`

Controls navigation in the roundabout. After entering, the robot follows the line and uses logic with a flag (`aguardando_realinhar`) and a detection counter to reliably count the exits, preventing the same exit from being counted multiple times. Upon reaching the desired exit, it executes the maneuver to leave the roundabout.

```
1  void realiza_rotatoria(int saidaCurva, int saidaDesejada) {
2    bool aguardando_realinhar = false;
3    static int contador_deteccao_saida = 0;
4    const int TOLERANCIA_SAIDA = 2;
5    int saidaAtual = 0;
6
7    if (saidaCurva == SAIDA_ESQUERDA) {
8      turn_left(velocidadeBaseDireita, velocidadeBaseEsquerda);
9      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: Executes the roundabout challenge with improved exit counting.

4.13 Function `tenta_recuperar_linha()`

Recovery routine executed when the line is lost. The robot moves backward for a time limit (`TIME_WITHOUT_LINE`), searching for the line. To confirm that the line has been found, it requires 3 consecutive valid readings.

```
1  bool tenta_recuperar_linha() {
2    unsigned long tempoPerdido = millis();
3    int leiturasValidas = 0;
4
```

```
  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: Attempts to recover the line by moving backward.

4.14 Function `conta_marcacao()`

Performs the counting of markers (black squares). The logic has been improved to count on the transition to white and require a minimum number of readings (`TOLERANCIA_MARCACAO`) to avoid noise, ensuring each marker is counted only once.

```
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: Track marker counting with a noise filter.

4.15 Function `analisa_marcacoes()`

Orchestrates the detection of challenges. It enters a loop for a defined time (`TIMEOUT_MARCACAO`) after detecting a curve, calling `conta_marcacao` to register the markers on each side and their respective detection times.

```
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);
      marcacoesDireita  = conta_marcacao(SENSOR_CURVA[1],
    marcacoesDireita, jaContouDireita, contadorBrancoDir,
    tempoMarcacaoDireita);

      if (marcacoesEsquerda > marcacoesAntesEsq ||
    marcacoesDireita > marcacoesAntesDir) {
        tempoUltimaDeteccao = millis();
      }

      if (marcacoesDireita >= 1 && marcacoesEsquerda >= 1) {
        marcacoesDireita = 1;
        marcacoesEsquerda = 1;
      }
      calcula_erro();
```

```
28        calcula_PID();
29        ajusta_movimento();
30      }
31    }
32  }
```

Listing 24: Analyzes and counts track markers for a set duration.

4.16 Function `area_de_parada()`

Final function that is called when the robot cannot recover from a line loss or reaches the attempt limit. It moves forward a little, stops the motors, turns on the LED, and enters an infinite loop, finishing the course.

```
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: Finishes the course in the stop area.

5 Motor Control File: `motors.cpp`

This file represents the hardware abstraction layer for controlling the robot's motors. It is responsible for initializing the microcontroller's pins and providing a simple interface to perform basic movements like moving forward, backward, turning, and stopping.

5.1 Function `setup_motor()`

Initializes the motor control pins as outputs.

```
1  void setup_motor() {
2    pinMode(MOTOR_LEFT_CLKWISE, OUTPUT);
```

```
3   pinMode(MOTOR_LEFT_ANTI, OUTPUT);
4   pinMode(MOTOR_RIGHT_CLKWISE, OUTPUT);
5   pinMode(MOTOR_RIGHT_ANTI, OUTPUT);
6   stop_motors();
7 }
```

Listing 26: Initial configuration of the motor pins.

5.2 Function `set_state_motor()`

Low-level function that directly applies PWM values (0-255) to the motor control pins.

```
1 void set_state_motor(int leftCw, int leftCcw, int rightCw,
     int rightCcw) {
2   analogWrite(MOTOR_LEFT_CLKWISE, leftCw);
3   analogWrite(MOTOR_LEFT_ANTI, leftCcw);
4   analogWrite(MOTOR_RIGHT_CLKWISE, rightCw);
5   analogWrite(MOTOR_RIGHT_ANTI, rightCcw);
6 }
```

Listing 27: Sets the state and speed of each motor.

6 Constants and Global Variables File: `constants.cpp`

This file centralizes all constants and global variables. This approach facilitates parameter adjustment and robot state management without altering the main logic.

6.1 Pins and Hardware

Defines the physical connections for the sensors, LEDs, and the Chip Select pin for the SD card.

6.2 Sensor State Variables

Arrays that store the readings of the line sensors (`SENSOR`) and curve sensors (`SENSOR_CURVA`).

6.3 Speed and Motor Settings

- `velocidadeBase`: The default speed, which is higher (255) in drag race mode and lower (210) in competition mode.
- `OFFSET_MOTORS`: An adjustment value to compensate for mechanical differences between the motors, ensuring straight movement.
- `velocidadeDireita`, `velocidadeEsquerda`: Variables that store the speed adjusted by the PID.

6.4 PID Control

- **Gains**: `Kp`, `Ki`, `Kd`. Notably, in drag race mode (`arrancadaMode`), the `Ki` and `Kd` gains are zeroed, turning the control into a purely Proportional system, which is simpler and more aggressive.
- **Calculation Variables**: `erro`, `P`, `I`, `D`, `PID`, etc., used for real-time calculation.

6.5 Challenge State Variables

Flags and counters that manage the robot's state during the competition.

- `inversaoAtiva`: Flag that indicates if the robot is on an inverted track section.
- `inversao_finalizada`: Flag activated when exiting an inversion, signaling a possible pedestrian crossing.
- `marcacoesDireita`, `marcacoesEsquerda`: Counters for the challenge markers.

- `tempoMarcacaoDireita`, `tempoMarcacaoEsquerda`: Store the `millis()` at the moment of a marker's detection, used to differentiate challenges.
- `ultima_posicao_linha`: Stores the last valid position of the line to aid in recovery.

6.6 Debug Control

Boolean flags to activate different test modes.

- `debugMode`: Activates printing data to the serial monitor.
- `debugMotor`: Activates a motor test in the `loop()`.
- `debugSD`: Enables logging to the SD card.
- `arrancadaMode`: Activates a simplified performance mode, with higher speed and only Proportional control.

6.7 Tolerance and Time Parameters

Constants that define the robot's temporal behavior.

- `TIMEOUT_FAIXA_PEDESTRE`: Stop time at the pedestrian crossing.
- `TIMEOUT_MARCACAO`: Duration of the time window to analyze the markers of a challenge.
- `DEBOUNCE_TEMPO_CURVA`: Waiting time to avoid double detection of the same curve.
- `TOLERANCIA_TEMPO_SIMULTANEO`: Time threshold to differentiate a reverse gear from an intersection.
- `LIMITE_TOLERANCIA_LINHA_PERDIDA`: Number of "line lost" cycles before initiating recovery.

7 Analysis and Visualization Script: `plot_log.py`

To assist in the PID controller calibration process, a Python script, `plot_log.py`, was developed. This tool allows for the analysis of the `.TXT` log files generated by the robot, visualization of performance in detailed graphs, and most importantly, simulation of the robot's behavior with different PID constants without the need for a new physical run. The script recalculates the PID output based on the error history and the new gains provided by the user, also offering automatic suggestions for optimization.

7.1 Function `calculate_pid_values()`

This is the central function of the simulation. It iterates over each error record from the log file and recalculates, line by line, what the Proportional, Integral, and Derivative terms and the final PID output would have been if the new `Kp`, `Ki`, and `Kd` constants had been used. The logic, including the constraint on the integral term, exactly mirrors the microcontroller's behavior.

```
def calculate_pid_values(df, Kp, Ki, Kd):
    pid_values = []
    I = 0
    erroAnterior = 0
    OFFSET = 0

    for erro in df['Error']:
        P = erro
        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

```

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

Listing 28: Recalculates the PID output with new constants.

7.2 Function `plot_data()`

This function is responsible for generating the graphical visualization of the data. It creates a window with four superimposed plots:

- **Error vs. Time**: Shows the robot's deviation from the line, including markers for the competition challenges. A smoothed error curve (using a Savitzky-Golay filter) is overlaid to show the overall trend of the run.
- **Smoothed Error vs. Time**: Presents the smoothed error curve for a clearer view of the trajectory.
- **PID Output vs. Time**: Displays the correction value calculated by the PID.
- **Motor Speeds vs. Time**: Shows how the PID output was translated into the speed of each motor.

```
1  def plot_data(df, Kp, Ki, Kd):
2      fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize
       =(14, 12), sharex=True)
3      fig.suptitle(f"Analysis of PID Control with Kp={Kp}, Ki={
       Ki}, Kd={Kd}", fontsize=16)
4
5      try:
6          window_length = min(51, len(df['Error']))
7          if window_length % 2 == 0:
8              window_length -= 1
9
10         if window_length > 3:
11             df['Smoothed_Error'] = savgol_filter(df['Error'
       ], window_length, 3)
12         else:
13             df['Smoothed_Error'] = df['Error']
14     except Exception:
15         df['Smoothed_Error'] = df['Error']
16
17     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('Motor Speeds vs. Time')

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

Listing 29: Generates the performance analysis graphs.

7.3 Function `analyze_performance_and_suggest_pid()`

After visualizing the graphs, this function performs a quantitative analysis of the robot's performance. It calculates metrics such as mean absolute error (MAE), maximum error (overshoot), and the number of oscillations (using peak detection). Based on these metrics, it prints a diagnosis of the robot's behavior in the terminal and suggests clear actions for adjusting the PID gains, automating the calibration process.

```
def analyze_performance_and_suggest_pid(df, Kp, Ki, Kd):
    print("\n--- Performance Analysis & Adjustment Suggestions ---")

    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"Mean Absolute Error (MAE): {mae:.4f}")
    print(f"Maximum Absolute Error (Overshoot/Undershoot): {max_abs_error:.4f}")
    print(f"Detected {num_oscillations} significant oscillations.")

    print("\nSuggestions:")
    suggestion_made = False

    if num_oscillations > 8:
        print("- Behavior Detected: The robot oscillates too much...")
        print(f"  - Recommended Action: Decrease the value of Kp (current: {Kp}).")
        suggestion_made = True

    # ... (logic for other suggestions) ...

    if not suggestion_made:
        print("- Performance seems reasonable.")
```

Listing 30: Analyzes metrics and suggests adjustments for the PID.

7.4 Example of Graphical Output

Figure 1 illustrates the analysis window generated by the `plot_log.py` script. It is possible to observe the error behavior (top graph), the PID controller output (middle graph), and the resulting motor speeds (bottom graph). The colored markers on the first graph indicate the exact moments when competition events occurred, such as the detection of an 'inverted$_l$$ine$'($bluesquare$), the'$Pede$

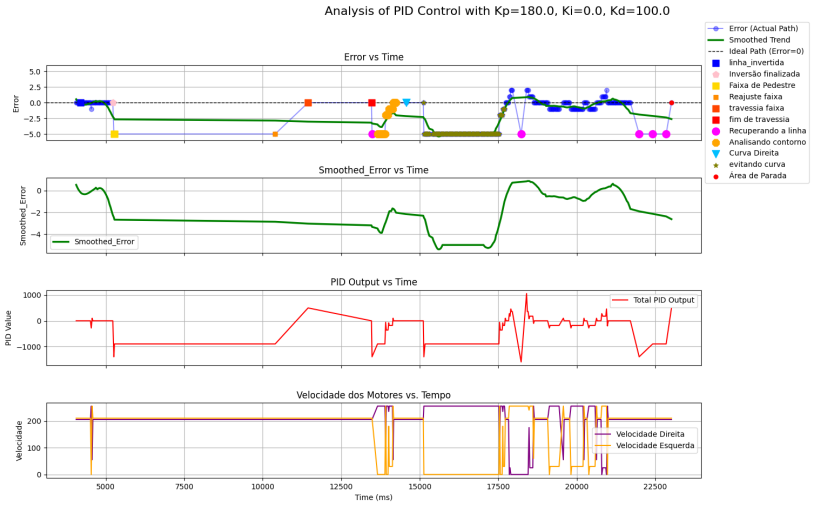

Figure 1: Example of the analysis window generated by the `plot_log.py` script.

8 Bibliographical References

8.1 PID Control

PID (Proportional Integral Derivative) Control is used for process control. Its objective is, through mathematical equations involving integrals and derivatives, to minimize the system's error. The proportional control adjusts the control variable proportionally to the error. The integral control adjusts the control variable based on the time the error occurs. The derivative control adjusts the control variable based on the rate of change of the error.

PID control equation:

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

In this equation, we have the following parameters:

- K_p: proportional gain;
- K_i: integral gain;
- K_d: derivative gain;
- e: instantaneous error (difference between the desired value and the measured value);
- τ: integration variable;
- t: current time.

Below is a table explaining how to adjust the PID parameters according to the error the robot exhibits.

Robot Behavior	**Parameter to Adjust**	**Recommended Action**
Oscillates a lot around the line (zig-zag)	Proportional (Kp)	Decrease the value of Kp. A very high Kp causes exaggerated reactions.
Corrects slowly and cannot follow the line well	Proportional (Kp)	Increase the value of Kp. A very low Kp generates weak corrections.
A small error persists for a long time	Integral (Ki)	Increase Ki to correct slowly accumulating errors.
Starts oscillating after following well for a while	Integral (Ki)	Decrease Ki. A high Ki can accumulate too much error and cause instability.
Very abrupt reaction to rapid changes in the line	Derivative (Kd)	Increase Kd to smooth the response. It helps to "brake" sudden changes.
Slow response to rapid changes	Derivative (Kd)	Decrease Kd if the robot takes too long to react to curves or deviations.

Table 1: Adjusting PID parameters based on the robot's behavior

9 Conclusion

This document presents the structure and functionalities of the code, facilitating the understanding of the system and promoting its democratization. It provides freedom and autonomy for conducting tests and using the

code in different contexts, allowing for greater flexibility in the development and improvement of the project.