

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

**EDUARDO HENRIQUE JUNIOR DOS SANTOS
HIURI CARRIÇO LIBERATO
LUIZ PHILLYP SABADINI BAZONI
MATEUS TICIANELI SARTORIO**

**RELATÓRIO DO TRABALHO FINAL DE
SISTEMAS EMBARCADOS I**

Vitória

2023

EDUARDO HENRIQUE JUNIOR DOS SANTOS
HIURI CARRICO LIBERATO
LUIZ PHILLYP SABADINI BAZONI
MATEUS TICIANELI SARTORIO

RELATÓRIO DO TRABALHO FINAL DE SISTEMAS EMBARCADOS I

Relatório de Trabalho apresentado à disciplina de Sistemas Embarcados I do curso de graduação em Engenharia de Computação da Universidade Federal do Espírito Santo, como requisito parcial para avaliação. Prof. Camilo Arturo Rodriguez Diaz.

Vitória
2023

SUMÁRIO

1. Introdução	4
2. Objetivo	4
3. O robô	4
3.1. Sensores	6
3.1.1. Sensor infravermelho	6
3.1.2. Sensor ultrassônico	6
3.1. Atuadores	7
3.1.1. Motor CC	7
3.1.2. Servomotor	7
4. O Labirinto	8
5. Solução proposta	8
6. O código	9
6.1 Algoritmo básico	9
6.2 Algoritmo melhorado	20
7. Considerações finais	23
8. Referências	24

1. Introdução

A construção de robôs destinados a competições como a OBR e exposições como a MNR tem crescido como uma atividade científica dedicada à pesquisa e educação. Além disso, desenvolver robôs capazes de realizar tarefas de forma autônoma é uma empreitada desafiadora e estimulante, abrangendo diversas áreas, incluindo programação de sistemas embarcados e construção de robôs, o que envolve desde projeto e manuseio de eletrônicos , até teorias dos cursos de Engenharia de Computação e Engenharia Elétrica. Este artigo aborda a implementação do código de um robô autônomo seguidor de linha, baseado em competições de programação, destinado a resolver labirintos. Seus sensores têm a capacidade de identificar, em tempo real, tanto linhas quanto objetos. A utilização desse sistema possibilitou a identificação precisa de objetos e linhas durante a simulação de uma competição.

2. Objetivo

Aplicar os conhecimentos adquiridos durante o curso no desenvolvimento de um robô seguidor de linha com a habilidade específica de navegar e solucionar labirintos. O projeto buscou integrar os aprendizados teóricos e práticos adquiridos ao longo do curso para implementar um robô capaz de seguir uma trajetória delineada por uma linha, utilizando sensores e algoritmos para navegação autônoma, de forma que fosse capaz de orientar-se e encontrar a saída em um labirinto.

3. O robô

O robô utilizado neste projeto é mostrado na **Figura 3.1**. Ele possui seis sensores infravermelhos para seguir a linha, dois motores de corrente contínua que movimentam suas rodas e o orientam, além de um servomotor para controlar a direção do sensor ultrassônico, usado para detectar as paredes do labirinto.

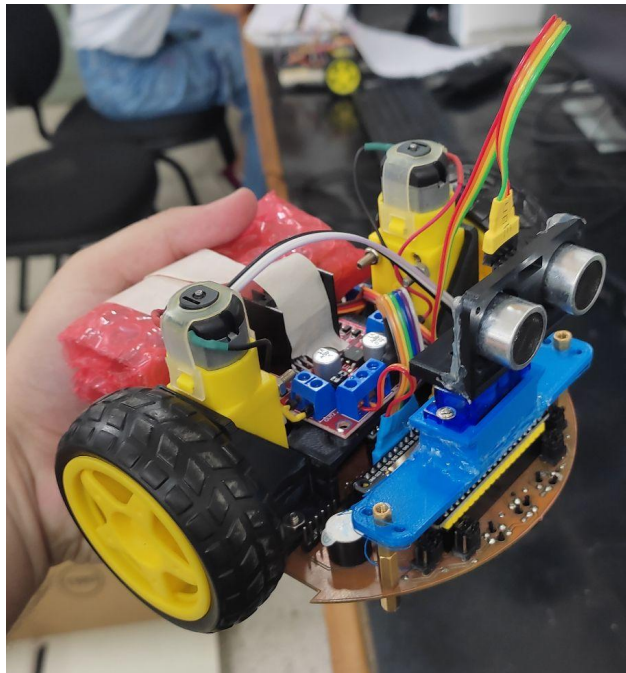


Figura 3.1 - Robô utilizado no projeto

A combinação desses componentes juntamente com os algoritmos implementados, que serão descritos posteriormente, permite que o robô seja capaz de navegar pelo labirinto e encontrar uma saída. Este relatório abordará a seguir detalhes sobre esses componentes, como foram usados e os algoritmos de controle. A organização dos componentes utilizados está disposta na **Figura 3.2**, que representa o esquemático do robô.

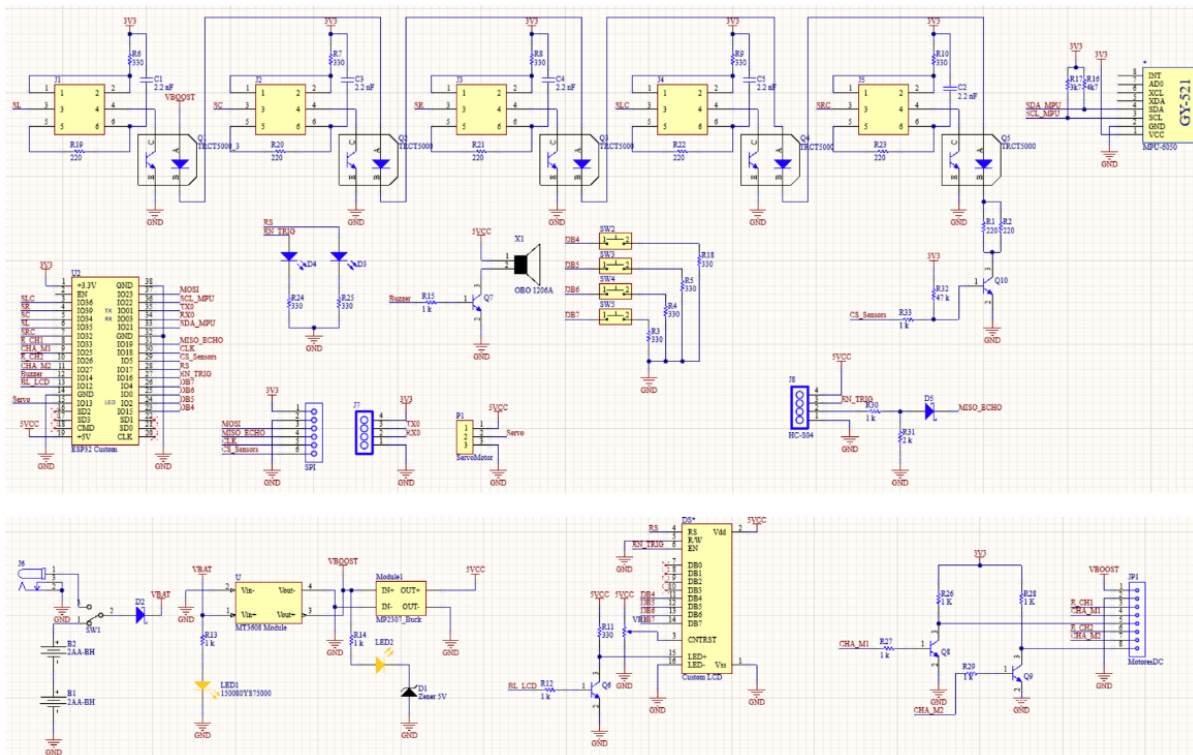


Figura 3.2 - Esquemático do robô

3.1. Sensores

3.1.1. Sensor infravermelho

No total, são seis sensores, conforme evidenciado na **Figura 3.1.1**. Cada sensor é composto por um LED infravermelho como emissor e um fototransistor como receptor. O princípio de operação desses sensores está fundamentado na reflexão da luz. Por meio de uma lente convergente, emite-se um feixe infravermelho. Ao atingir o objeto, uma porção desse feixe é refletida, com sua intensidade variando de acordo com a cor e a textura da superfície. O receptor capta essa luz refletida para realizar a detecção. No contexto deste projeto, optou-se pelo uso dos sensores TCRT5000 para identificar e seguir as linhas presentes no labirinto.

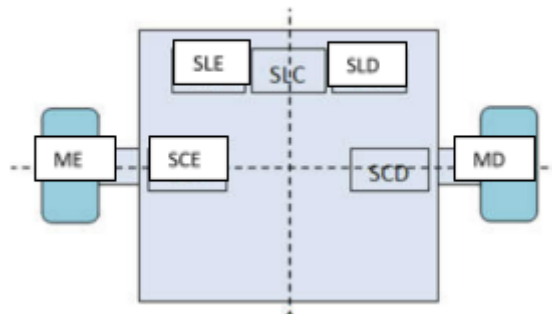


Figura 3.1.1 - Sensores Infravermelhos do robô

O seis sensores distribuem-se da forma apresentada na **Figura 3.2**, onde SLE e SLD são respectivamente os sensores de linha esquerdo e direito, SCE e SCD correspondem respectivamente aos sensores de cruzamento esquerdo e direito e SLC trata-se do sensor de linha central, por onde o robô irá se orientar.

3.1.2. Sensor ultrassônico

O sensor ultrassônico em conjunto com o servomotor desempenha um papel crucial na detecção das paredes no ambiente labiríntico. O processo inicia-se com o transmissor que emite um pulso de ultrassom em direção ao entorno. Este pulso é então refletido pelo objeto mais próximo e detectado pelo receptor no sensor.

A medição da distância se baseia na análise do tempo transcorrido entre a emissão do som e a recepção do eco. A partir dessa informação temporal, é possível determinar com precisão a distância do objeto em relação ao sensor. Contudo, é importante notar que esses sensores apresentam algumas limitações, incluindo sua baixa precisão e a influência do formato da superfície que reflete a onda ultrassônica. Portanto, nesse

projeto para o processamento do valor lido foram calculadas as médias de medições realizadas pelo sensor HC-SR04, para garantir assim uma maior precisão na identificação de paredes.

3.1. Atuadores

3.1.1. Motor CC

Os motores de corrente contínua (CC) são componentes que convertem energia elétrica em movimento mecânico. Funcionam através da interação entre um campo magnético fixo e um campo magnético rotativo, gerando movimento rotacional. Em um motor CC típico, a corrente elétrica é fornecida por uma fonte de tensão contínua, criando um fluxo de corrente nas bobinas do motor. Este fluxo de corrente produz um campo magnético que interage com ímãs permanentes ou enrolamentos específicos, resultando no movimento rotacional do eixo do motor.

No contexto do robô para o labirinto, é importante ressaltar que esses motores serão controlados por modulação de largura de pulso (PWM). O controle PWM permite variar a velocidade dos motores ao ajustar a largura dos pulsos de energia, oferecendo maior precisão no controle de movimento e na manobrabilidade do robô dentro do ambiente do labirinto.

A disposição dos motores cc no robô é dada conforme o apresentado na **Figura 3.2**, identificados como MD (motor direito) e ME (motor esquerdo).

3.1.2. Servomotor

O servomotor é um tipo específico de motor elétrico projetado para controlar tanto a velocidade quanto a posição. Sua estrutura básica inclui um circuito de controle, um motor CC de alta velocidade com baixo torque, um sistema de redução ou caixa de engrenagens para amplificar o torque, e um potenciômetro que fornece feedback ao sistema de controle. O servomotor adotado, TP-SG90, tem um limite de rotação de cerca de 180° devido à natureza não contínua dos potenciômetros utilizados.

No contexto do robô no labirinto, o servomotor TP-SG90 o papel de controlar a orientação do sensor ultrassônico. Essa capacidade de movimento controlado possibilita explorar diferentes direções para identificar as paredes do labirinto, fornecendo informações essenciais para a navegação e tomada de decisões do robô.

4. O Labirinto

O labirinto proposto para o percurso no robô é formado por uma grade feita com fita de cor preta sobre uma superfície branca permitindo o contraste necessário para um bom funcionamento dos sensores infravermelhos. Ao longo da grade formada foram introduzidos obstáculos para de fato construir os caminhos válidos ao longo do labirinto.

Com o objetivo de facilitar a implementação do projeto essa construção faz com que naturalmente cada cruzamento ou curva sejam compostos pelo mesmo padrão de leitura (duas linhas pretas cruzadas) fazendo com que o mesmo trecho de código possa ser usado para verificar as situações onde o robô precisa decidir uma curva.



Figura 4 - Sensores Infravermelhos do robô

5. Solução proposta

Uma solução trivial para encontrar a saída do labirinto consiste em manter o robô virando para uma direção (a direção esquerda foi escolhida arbitrariamente) sempre que possível. Desta forma, em algum momento ele estará em um estado onde esta decisão seria a correta para cruzar a saída do labirinto. No entanto, essa solução não pode ser considerada adequada pois dependendo da configuração dos caminhos existe a possibilidade de que o robô faça decisões que o mantenha preso dentro do labirinto ou até mesmo saia pela entrada e não pela saída como esperado. Uma alternativa que garante o trajeto correto do robô se baseia na solução do clássico problema de *pathfinding*. Nela mapeamos o labirinto formando um grafo não-dirigido ao caminhar com o robô pelos caminhos disponíveis e em seguida precisamos calcular qual trajeto ao longo desse grafo que faz com que o robô percorra a

menor distância até a saída.

Um dos algoritmos mais eficientes para conectar dois pontos em um grafo com mínimo custo trata-se do algoritmo de Dijkstra, algoritmo desenvolvido de maneira brilhante por Edsger Dijkstra em menos de vinte minutos durante a sua juventude. O algoritmo se baseia em caminhar virtualmente ao longo do grafo atualizando a menor distância percorrida até que o vértice de destino seja atingido, sendo garantida em todos os casos a solução ótima para o grafo em questão. Ao utilizar tal algoritmo podemos garantir que o robô encontre a saída de qualquer labirinto montado em tempo polinomial.

6. O código

Para implementar as soluções, utilizou-se a linguagem de programação C++ juntamente com as APIs disponibilizadas pela Arduino IDE. O código pode ser acessado integralmente no repositório padrão do projeto em [1].

6.1 Algoritmo básico

O algoritmo básico pode ser encontrado em [1], na *branch minimal-solution*.

A estrutura de código proposta pela interface do arduino consiste em uma função de setup, para configurar pinagem, PWM e afins e uma função de loop que roda continuamente após o setup. Na **Figura 6.1.1**, é mostrado o código completo de setup.

```

1 void setup() {
2   Serial.begin(115200);
3   Wire.begin();
4
5   byte status = mpu.begin();
6   Serial.print(F("MPU6050 status: "));
7   Serial.println(status);
8   while(status != 0) {} // para tudo se nao conseguiu se conectar ao MPU6050
9
10  Serial.println(F("Calculando offsets, nao mova o MPU6050"));
11  delay(1000);
12  // mpu.upsideDownMounting = true; // descomente essa linha se o MPU6050 esta montado de cabeca para baixo
13  mpu.calcOffsets(); // gyro and accelero
14  Serial.println("Pronto!\n");
15
16  // Sensores IR
17  pinMode(CS_Sensors, OUTPUT);
18  pinMode(SR, INPUT);
19  pinMode(SL, INPUT);
20  pinMode(SLC, INPUT);
21  pinMode(SLC, INPUT);
22  pinMode(SC, INPUT);
23
24  // Giroscopio
25  pinMode(SDA, INPUT);
26  pinMode(SCL, INPUT);
27
28  // Motores da roda
29  pinMode(E_CH1, OUTPUT);
30  pinMode(E_CH2, OUTPUT);
31
32  ledcAttachPin(CHA_M1, PWM1_Ch);
33  ledcSetup(PWM1_Ch, PWM1_Freq, PWM1_Res);
34
35  ledcAttachPin(CHA_M2, PWM2_Ch);
36  ledcSetup(PWM2_Ch, PWM1_Freq, PWM1_Res);
37
38  // Servo motor
39  sg90.setPeriodHertz(50); // frequencia de PWM para SG90
40  sg90.attach(PIN_SG90, 500, 2400); // Minimo and maximo comprimento do pulso (em us) para ir de 0° a 180°
41
42  // Ultrassom
43  pinMode(EN_TRIG, OUTPUT);
44  pinMode(MISO_ECHO, INPUT);
45
46  // Coloca o servo apontando para frente antes de comecar a mover o robo
47  sg90.write(90);
48 }

```

Figura 6.1.1 - Código completo de setup

Da função de setup, vale a pena destacar o trecho de código responsável por calibrar o giroscópio (MPU6050), que calcula os offsets necessários para o funcionamento correto do sensor, como mostrado na **Figura 6.1.2**.

```

1 void setup() {
2   ...
3
4   byte status = mpu.begin();
5   Serial.print(F("MPU6050 status: "));
6   Serial.println(status);
7   while(status != 0) {} // para tudo se nao conseguiu se conectar ao MPU6050
8
9   Serial.println(F("Calculando offsets, nao mova o MPU6050"));
10  delay(1000);
11  // mpu.upsideDownMounting = true; // descomente essa linha se o MPU6050 esta montado de cabeca para baixo
12  mpu.calcOffsets(); // gyro and accelero
13  Serial.println("Pronto!\n");
14
15  ...
16 }

```

Figura 6.1.2 - Calibração de offsets do MPU6050

Além disso, na **Figura 6.1.3**, é mostrada a configuração do servo motor (SG90), que deve ser anexado a um canal PWM do ESP32.

```

1 void setup() {
2   ...
3
4   // Servo motor
5   sg90.setPeriodHertz(50); // frequencia de PWM para SG90
6   sg90.attach(PIN_SG90, 500, 2400); // Minimo and maximo comprimento do pulso (em µs) para ir de 0° a 180°
7
8   // Ultrasom
9   pinMode(EN_TRIG, OUTPUT);
10  pinMode(MISO_ECHO, INPUT);
11
12  // Coloca o servo apontando para frente antes de começar a mover o robo
13  sg90.write(90);
14 }

```

Figura 6.1.4 - Configuração do SG90

Após a configuração do código, a função loop rodará permanentemente enquanto o programa estiver sendo executado. Na **Figura 6.1.5**, é mostrada a implementação completa da função loop.

```

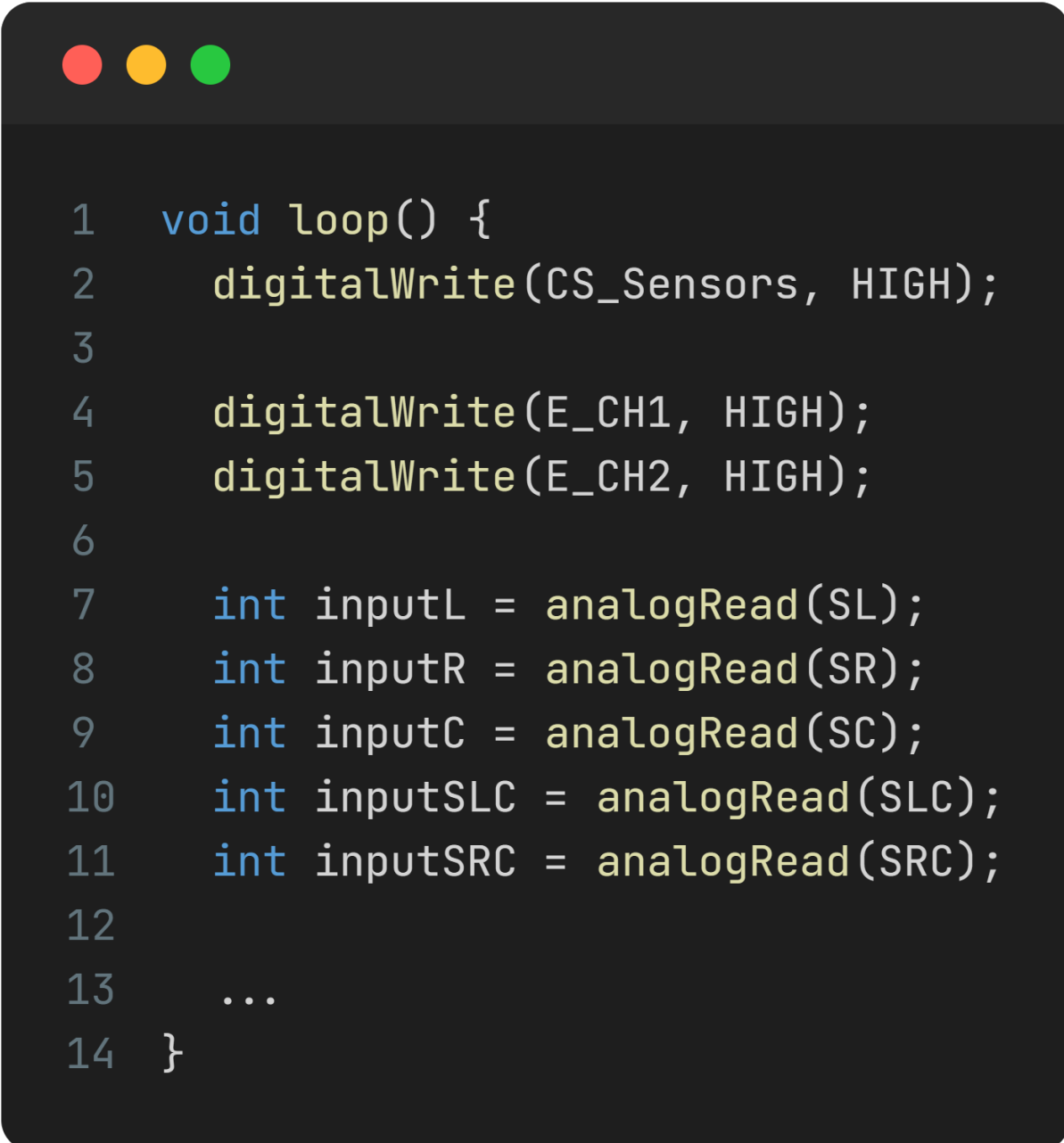
1 void loop() {
2     digitalWrite(CS_Sensors, HIGH);
3
4     digitalWrite(E_CH1, HIGH);
5     digitalWrite(E_CH2, HIGH);
6
7     int inputL = analogRead(SL);
8     int inputR = analogRead(SR);
9     int inputC = analogRead(SC);
10    int inputSLC = analogRead(SLC);
11    int inputSRC = analogRead(SRC);
12
13    if(inputL > 2000 && inputR > 2000 && inputC > 2000) {
14        slow = true;
15    }
16
17    if((inputSLC > 2000 && inputSRC > 2000) && (millis() - ultima_curva > 1000)) {
18        slow = false;
19        processa_cruzamento();
20    }
21    else {
22        int deltaL = (int) (((float) inputL/4000.0)*DELTA_VEL);
23        int deltaR = (int) (((float) inputR/4000.0)*DELTA_VEL);
24
25        int pwmL = 0;
26        int pwmR = 0;
27
28        if(slow) {
29            pwmL = SLOW_VEL - deltaL + deltaR;
30            pwmR = SLOW_VEL - deltaR + deltaL;
31        }
32        else {
33            pwmL = BASE_VEL - deltaL + deltaR;
34            pwmR = BASE_VEL - deltaR + deltaL;
35        }
36
37        // motor direito
38        ledcWrite(PWM1_Ch, pwmR);
39
40        // motor esquerdo (invertido)
41        ledcWrite(PWM2_Ch, 255 - pwmL);
42    }
43 }

```

Figura 6.1.5 - Implementação completa da função loop

A **Figura 6.1.6** mostra o primeiro processo que é executado na função loop: habilitar o LED dos sensores de infravermelho (CS_Sensors) e de habilitar os motores das rodas (E_CH1 e E_CH2). Em seguida, o valor de

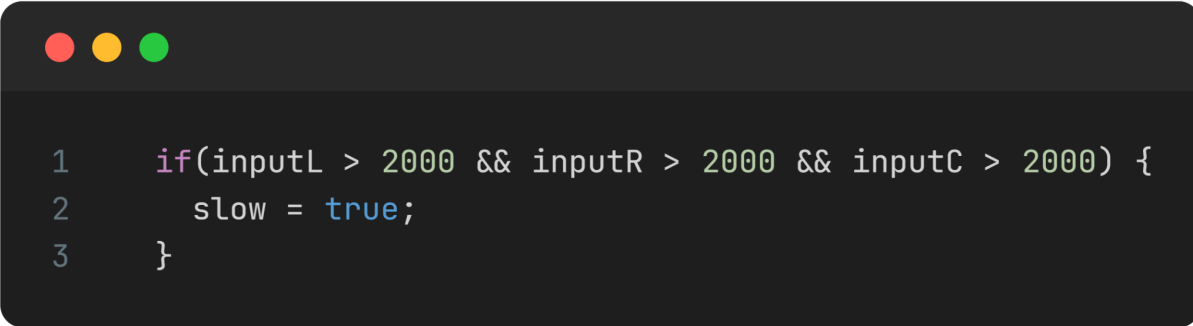
todos os sensores IR são lidos e armazenados em suas respectivas variáveis.



```
1  void loop() {
2      digitalWrite(CS_Sensors, HIGH);
3
4      digitalWrite(E_CH1, HIGH);
5      digitalWrite(E_CH2, HIGH);
6
7      int inputL = analogRead(SL);
8      int inputR = analogRead(SR);
9      int inputC = analogRead(SC);
10     int inputSLC = analogRead(SLC);
11     int inputSRC = analogRead(SRC);
12
13     ...
14 }
```

Figura 6.1.6 - Parte inicial da função de loop

Em seguida, caso os três sensores IR da frente (inputL, inputC e inputR) detectem a passagem pela linha preta, o robô entra no modo lento (slow é setado como *true*). O código que realiza esta verificação é mostrado na **Figura 6.1.7**.



```
1    if(inputL > 2000 && inputR > 2000 && inputC > 2000) {  
2        slow = true;  
3    }
```

Figura 6.1.7 - Sensores da frente verificam se um cruzamento foi encontrado

No entanto, enquanto o robô não encontra o cruzamento com os sensores IR de trás (sensores de cruzamento), ele realiza o controle proporcional do PWM dos motores das rodas, de forma que ele consiga seguir a linha preta. Caso o robô esteja no modo *slow*, os valores de PWM colocados na roda são menores, mas a lógica de controle é a mesma. Os valores lidos pelos sensores IR da esquerda (*inputL*) e da direita (*inputR*) são lidos e divididos por 4000 (que é aproximadamente o valor lido pelos sensores, quando estão em cima da linha preta). A razão entre o valor lido e o valor máximo é então multiplicado por um delta (*DELTA_VEL* = 20). O valor base de PWM de cada roda é subtraído do seu delta e somado do delta da roda oposta, de forma que se o sensor da esquerda começa a ler a linha preta, a roda da esquerda irá girar mais lenta e a da direita mais rápida, o que irá corrigir a direção do robô de forma a seguir a linha. A **Figura 6.1.8** mostra a implementação desta lógica em código. Deve-se observar que a roda esquerda está com seu sentido de rotação invertido em relação à direita.

```

1  if((inputSLC > 2000 && inputSRC > 2000) && (millis() - ultima_curva > 1000)) {
2      ...
3  }
4  else {
5      int deltaL = (int) (((float) inputL/4000.0)*DELTA_VEL);
6      int deltaR = (int) (((float) inputR/4000.0)*DELTA_VEL);
7
8      int pwmL = 0;
9      int pwmR = 0;
10
11     if(slow) {
12         pwmL = SLOW_VEL - deltaL + deltaR;
13         pwmR = SLOW_VEL - deltaR + deltaL;
14     }
15     else {
16         pwmL = BASE_VEL - deltaL + deltaR;
17         pwmR = BASE_VEL - deltaR + deltaL;
18     }
19
20     // motor direito
21     ledcWrite(PWM1_Ch, pwmR);
22
23     // motor esquerdo (invertido)
24     ledcWrite(PWM2_Ch, 255 - pwmL);
25 }

```

Figura 6.1.8 - Lógica de controle com PWM das rodas do robô

Ademais, quando os sensores de cruzamento detectam a linha preta, a variável *slow* é *setada* para *false*, e o robô chama a função de processar cruzamento. A parte do código que faz esta verificação é mostrada na **Figura 6.1.8**.

```

1  if((inputSLC > 2000 && inputSRC > 2000) && (millis() - ultima_curva > 1000)) {
2      slow = false;
3      processa_cruzamento();
4  }
5  else {
6      ...
7  }

```

Figura 6.1.8 - Sensores de cruzamento verificam se encontraram cruzamento

O código completo da função que processa o cruzamento é mostrado na **Figura 6.1.9**.

```

1 void processa_cruzamento() {
2     float distancias[3] = { 0 };
3
4     // motor esquerdo (invertido)
5     ledcWrite(PWM1_Ch, 127);
6     ledcWrite(PWM2_Ch, 127);
7
8     // aponta sensor ultrasom para a esquerda
9     sg90.write(180);
10    delay(1000);
11    distancias[0] = le_distancia();
12    ledcWrite(PWM1_Ch, 127);
13    ledcWrite(PWM2_Ch, 127);
14
15    // aponta sensor ultrasom para o meio
16    sg90.write(80);
17    delay(1000);
18    distancias[1] = le_distancia();
19
20    // aponta sensor ultrasom para a direita
21    sg90.write(0);
22    delay(1000);
23    distancias[2] = le_distancia();
24
25    // Após leitura, ajudar o servo motor para frente novamente
26    sg90.write(80);
27    delay(100);
28
29    // 0 robo vira para esquerda, frente e direita, na seguinte prioridade
30    // 1: esquerda
31    // 2: frente
32    // 3: direita
33    // 4: vira para tras (180°)
34    if(distancias[0] > DISTANCIA_OBSTACULO) {
35        vira_esquerda();
36    }
37    else if(distancias[1] > DISTANCIA_OBSTACULO) {
38        anda_reto();
39    }
40    else if(distancias[2] > DISTANCIA_OBSTACULO) {
41        vira_direita();
42    }
43    else {
44        vira_180();
45    }
46
47    ledcWrite(PWM1_Ch, 127);
48    ledcWrite(PWM2_Ch, 127);
49    delay(1000);
50
51    // Armazena o valor de tempo desde que terminou de fazer a curva
52    ultima_curva = millis();
53 }

```

Figura 6.1.9 - Lógica completa de processamento de cruzamento

Ao encontrar um cruzamento, os dois motores são freados (valor de 127 é colocado em seus PWM's). Após isto, o servo motor é utilizado para virar o sensor de ultrassom para esquerda, frente e direita. Para cada orientação, a função `le_distancia` é chamada, e seu valor armazenado em sua respectiva posição do array de distâncias. Então, caso nenhum obstáculo seja encontrado para a esquerda com distância menor que a distância mínima (`DISTANCIA_OBSTACULO = 22 [cm]`), então o robô chama a rotina de andar para a esquerda. Caso encontre obstáculo à esquerda, é verificada a distância para frente e para direita, nesta ordem, e o robô segue para a primeira destas direções que é possível seguir. Caso não seja possível seguir para nenhuma delas, o robô realiza um giro de 180°, e volta pelo caminho de onde veio. Ao final da função de processar cruzamento, o valor de tempo é anotado, para que o robô só detecte um cruzamento novamente depois de 1 segundo (para que dê tempo de o robô sair do cruzamento que ele está, e para que este não seja imediatamente detectado novamente).

A implementação da função `vira_direita` é mostrada na **Figura 6.1.10**. Nela, o valor de ângulo inicial é lido, e os motores são ligados para girar em sentidos opostos (de forma que o robô gire para a direita). O robô continua girando para a direita até completar um giro de 90°. As implementações das funções de girar a esquerda e de giro completo (`vira_180`) são implementadas de forma análoga, e podem ser encontradas no repositório do código [1].

```

1  void vira_direita() {
2      mpu.update();
3      int angulo_inicial = mpu.getAngleZ();
4      int angulo_atual = angulo_inicial;
5
6      int delta = angulo_atual - angulo_inicial;
7
8      // Vira ate o angulo variar em -90°
9      while(delta > -90) {
10         mpu.update();
11         angulo_atual = mpu.getAngleZ();
12
13         delta = angulo_atual - angulo_inicial;
14
15         ledcWrite(PWM1_Ch, 255 - BASE_VEL);
16         ledcWrite(PWM2_Ch, 255 - BASE_VEL);
17     }
18 }

```

Figura 6.1.10 - Lógica da função de virar à direita

Finalmente, a implementação da função de andar reto no cruzamento é bem trivial, e é mostrada na **Figura 6.1.11**.

```

1  void anda_reto() {
2      return;
3  }

```

Figura 6.1.11 - Lógica da função de andar reto

Por fim, na **Figura 6.1.12**, é mostrada a lógica da função `le_distancias`, que é responsável por ler a distância até o próximo objeto que está logo à frente do sensor de ultrassom. Como pode ser observado, são emitidas ondas de ultrassom, e o tempo que elas demoram para retornar são armazenadas. Este procedimento é realizado `QTD_MEDIDAS_ULTRASOM = 3` vezes, e então uma média de tempo é obtida. Realiza-se então uma medida de tempo a partir da velocidade do som em cm/s (`SOUND_SPEED = 0.034 [cm/s]`).

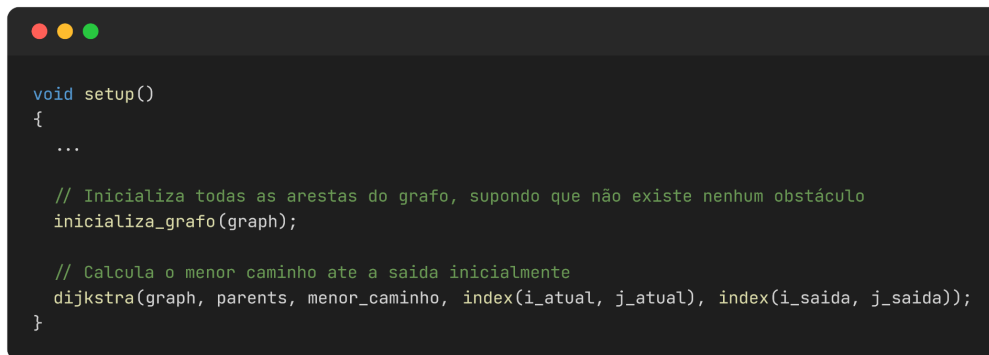
```
1  float le_distancia() {
2      long durations[QTD_MEDIDAS_ULTRASOM] = { 0 };
3
4      // Le o tempo de percurso da onda 3 vezes
5      for(int i = 0; i < QTD_MEDIDAS_ULTRASOM; i++) {
6          digitalWrite(EN_TRIG, LOW);
7          delayMicroseconds(2);
8          // Seta o EN_TRIG em HIGH por 10 microssegundos
9          digitalWrite(EN_TRIG, HIGH);
10         delayMicroseconds(10);
11         digitalWrite(EN_TRIG, LOW);
12
13         // Le o MISO_ECHO, retorna o tempo de percurso em microssegundos
14         durations[i] = pulseIn(MISO_ECHO, HIGH);
15     }
16
17     // Calcula a media dos tempos
18     long soma = 0;
19     for(int i = 0; i < QTD_MEDIDAS_ULTRASOM; i++) {
20         soma += durations[i];
21     }
22
23     double media = (double) soma / (double) QTD_MEDIDAS_ULTRASOM;
24
25     // Calcula a distancia em cm a partir do tempo
26     float distanceCm = media * SOUND_SPEED/2.0;
27
28     return distanceCm;
29 }
```

Figura 6.1.12 - Função que realiza leituras de distância com o sensor de ultrassom

6.2 Algoritmo melhorado

A solução melhorada para o problema é baseada no algoritmo de Dijkstra, onde o robô pressupõe que não existe nenhum obstáculo no labirinto, e a medida que ele vai encontrando os obstáculos, ele vai ajustando a rota de forma a encontrar o menor caminho possível para a saída com as informações que ele tem sobre o labirinto até o momento.

A lógica de configuração, de controle proporcional para seguir a linha e de virar é a mesma da solução básica. A grande diferença se encontra na forma como o robô decide qual caminho tomar. Como mostrado no trecho de código da **Figura 6.2.1**, inicialmente o grafo é inicializado como não contendo nenhum obstáculo, e o algoritmo de Dijkstra é utilizado para encontrar um caminho inicial.



```
void setup()
{
    ...

    // Inicializa todas as arestas do grafo, supondo que não existe nenhum obstáculo
    inicializa_grafo(graph);

    // Calcula o menor caminho ate a saida inicialmente
    dijkstra(graph, parents, menor_caminho, index(i_atual, j_atual), index(i_saida, j_saida));
}
```

Figura 6.2.1 - Setup inicial do grafo

Já a nova lógica de processamento de cruzamento é mostrada na **Figura 6.2.2**. Em cada cruzamento, é verificado se existe algum obstáculo à esquerda, à frente ou à direita. Caso exista, a aresta que conecta o vértice atual onde o robô está e o vértice que não pode ser alcançado por causa do obstáculo é removida. Então, o algoritmo de Dijkstra deve ser executado de novo, para encontrar o novo caminho mínimo atualizado.

```

void processa_cruzamento()
{
    double distancias[3] = {0};
    mede_distancias(distancias);

    // Caso algum objeto seja identificado em alguma direcao, a aresta equivalente eh retirada do grafo e o algoritmo de Dijkstra eh executado novamente
    bool caminho_alterado = false;
    if (distancias[0] < DISTANCIA_OBSTACULO)
    {
        caminho_alterado = true;
        int vertice_a_esquerda = obtem_vertice_a_esquerda(orientacao_atual, i_atual, j_atual);
        if (vertice_a_esquerda != -1)
        {
            graph[index(i_atual, j_atual)][vertice_a_esquerda] = 0;
            graph[vertice_a_esquerda][index(i_atual, j_atual)] = 0;
        }
    }

    if (distancias[1] < DISTANCIA_OBSTACULO)
    {
        caminho_alterado = true;
        int vertice_em_frente = obtem_vertice_em_frente(orientacao_atual, i_atual, j_atual);
        if (vertice_em_frente != -1)
        {
            graph[index(i_atual, j_atual)][vertice_em_frente] = 0;
            graph[vertice_em_frente][index(i_atual, j_atual)] = 0;
        }
    }

    if (distancias[2] < DISTANCIA_OBSTACULO)
    {
        caminho_alterado = true;
        int vertice_a_direita = obtem_vertice_a_direita(orientacao_atual, i_atual, j_atual);
        if (vertice_a_direita != -1)
        {
            graph[index(i_atual, j_atual)][vertice_a_direita] = 0;
            graph[vertice_a_direita][index(i_atual, j_atual)] = 0;
        }
    }

    // O algoritmo de Dijkstra so eh executado na primeira vez que o robo esta resolvendo o labirinto
    if (caminho_alterado && !fez_caminho_uma_vez)
    {
        dijkstra(graph, parents, menor_caminho, index(i_atual, j_atual), index(i_saida, j_saida));
        caminho_alterado = false;
    }

    ...
}

```

Figura 6.2.1 - Nova lógica de processamento de cruzamento

O menor caminho conhecido até a saída pelo robô até o momento é armazenada em uma pilha (`menor_caminho`). Para descobrir para onde o robô deve ir em breve, basta desempilhar o próximo vértice e obter qual a direção de giro o robô deve fazer, baseado no `proximo_vertice_a_andar` desempilhado e na orientação atual do robô. A parte do código que cuida desta lógica é mostrada na **Figura 6.2.3**.

```

void processa_cruzamento()
{
    ...

    // Calcula-se para qual vertice o robo deve ir, e esta informacao eh traduzida para uma direcao para qual ele deve virar
    int proximo_vertice_a_andar = menor_caminho.topElement();
    menor_caminho.pop();

    int direcao_da_curva = obtem_direcao_de_curva(orientacao_atual, i_atual, j_atual, proximo_vertice_a_andar);

    switch (direcao_da_curva)
    {
    case esquerda:
        vira_esquerda(mpu);
        break;
    case frente:
        anda_reto(mpu);
        break;
    case direita:
        vira_direita(mpu);
        break;
    case tras:
        vira_180(mpu);
        break;
    }

    ledcWrite(PWM1_Ch, 127);
    ledcWrite(PWM2_Ch, 127);

    i_atual = get_i(proximo_vertice_a_andar);
    j_atual = get_j(proximo_vertice_a_andar);

    orientacao_atual = obtem_nova_orientacao(orientacao_atual, direcao_da_curva);

    delay(1000);

    // Armazena o valor de tempo desde que terminou de fazer a curva
    ultima_curva = millis();
}

```

Figura 6.2.3 - Lógica para encontrar o próximo vértice para onde o robô deve andar

Finalmente, toda vez que o robô chega em um cruzamento (vértice), ele checa se chegou na saída. Caso tenha chegado, ele para e espera por 20 segundos para que seja posicionado na entrada de novo. Da segunda tentativa em diante, ele sempre faz o melhor caminho possível entre a entrada e a saída do labirinto. Na **Figura 6.2.4** é mostrada a verificação realizada pelo robô ao encontrar um cruzamento.

```

void loop()
{
    ...

    if ((inputSLC > 2000 && inputSRC > 2000) && (millis() - ultima_curva > 1000))
    {
        ...

        if (index(i_atual, j_atual) == index(i_saida, j_saida))
        {
            Serial.printf("Chegou ao final do percurso\n");

            ledcWrite(PWM1_Ch, 127);
            ledcWrite(PWM2_Ch, 127);

            fez_caminho_uma_vez = true;
            i_atual = i_inicial;
            j_atual = j_inicial;
            orientacao_atual = norte;

            dijkstra(graph, parents, menor_caminho, index(i_atual, j_atual), index(i_saida, j_saida));

            delay(20000);
        }
        else
        {
            ...
        }
    }
    else
    {
        ...
    }
}

```

Figura 6.2.4 - Ao chegar na saída, o robô ajusta sua orientação e posição inicial, e espera ser reposicionado na entrada novamente

7. Considerações finais

A solução básica implementada foi amplamente testada no robô, e apresentou resultados aceitáveis em boa parte dos testes. No entanto, em algumas configurações do labirinto, é possível que o robô entre em loop e nunca encontre a saída. Além disso, o robô não aplica nenhum tipo de estratégia para encontrar alguma forma de caminho otimizado.

Já a solução melhorada apresentou desempenho muito melhor, aprendendo o melhor caminho na primeira vez que anda pelo labirinto, e sempre fazendo a melhor trajetória até a saída nos percursos sucessivos.

Vídeos demonstrativos do robô executando ambas as soluções podem ser encontrados em [1].

8. Referências

[1] **robo-seguidor-de-linha**. Disponível em <https://github.com/MateusSartorio/robo-seguidor-de-linha>. Acesso em 18 de dezembro de 2023.