Braço robótico

(Software)

RAITEC UFC - Fortaleza

Universidade Federal do Ceará
Brasil
Fortaleza, 2025

Temos que o código do projeto anterior do braço era feito da seguinte maneira, segundo a documentação:

> "A movimentação do Braço Robótico é realizada por meio do uso de um código atrelado ao microcontrolador ESP32 e ao aplicativo Blynk IoT. Com isso, as posições dos ângulos dos servos motores, que são ajustadas no aplicativo, definirão as posições das movimentações do braço. Portanto, o usuáio deve acessar o aplicativo para manualmente definir os ângulos que o braço utilizará para agarrar uma peça, por exemplo."

Neste novo projeto temos uma tarefa para implementar o que foi feito acima, agora, temos que substituir esse "manualmente"por algo automático.

E é nesse contexto que a implementação por voz surge.

Mas antes vamos analisar o código do projeto inicial:

1 Análise do código antigo

```
#include <Blynk.h>
#include <WiFiClient.h>
#include <BlynkSimpleEsp32.h>
#include <ESP32Servo.h>

Servo servo1, servo2, servo3, servo4, servo5, servo6;

char auth[] = "cWOUXwKCihglXntLt4E2lhCZvYBXB83U";
char ssid[] = "alunos-01";
char pass[] = "";

bool buttonState = false;

int pos1Atual = 0;
int pos2Atual = 0;
int pos3Atual = 0;
int pos4Atual = 0;
int pos5Atual = 0;
int pos6Atual = 0;
void setup() {
    Serial.begin(9600);
    servo1.attach(2); // D2
    servo2.attach(16);
    servo3.attach(19);
    servo4.attach(22); // D13
    servo5.attach(12); // Adiciona servo5
    servo6.attach(27);

    Blynk.begin(auth, ssid, pass);
    delay(2000);
}

void loop() {
    Blynk.run();
}

void moveServo(Servo &servo, int &currentPos, int targetPos) {
    if (targetPos > currentPos) {
        for (int pos = currentPos; pos <= targetPos; pos++) {
            servo.write(pos);
```

```
            delay(15);
}
} else {
    for (int pos = currentPos; pos >= targetPos; pos--) {
        servo.write(pos);
        delay(15);
        }
    }
currentPos = targetPos;
}

BLYNK_WRITE(V4) // Button Widget on virtual pin 4
{
    buttonState = param.asInt();
}

BLYNK_WRITE(V0) // Slider Widget for Servo 1 on V0
{
    int pos = param.asInt();
    if (!buttonState) {
        moveServo(servo1, pos1Atual, pos);
    } else {
        moveServo(servo5, pos5Atual, pos);
    }
}

BLYNK_WRITE(V1) // Slider Widget for Servo 2 on V1
{
    int pos = param.asInt();
    if (!buttonState) {
        moveServo(servo2, pos2Atual, pos);
    } else {
        moveServo(servo6, pos6Atual, pos);
    }
}

BLYNK_WRITE(V2) // Slider Widget for Servo 3 on V2
{
    int pos = param.asInt();
    if (buttonState) {
        moveServo(servo3, pos3Atual, pos);
    }
}


BLYNK_WRITE(V3) // Slider Widget for Servo 4 on V3
{
    int pos = param.asInt();
    if (buttonState) {
        moveServo(servo4, pos4Atual, pos);
    }
}
```

Este código é feito para controlar até 6 servomotores conectados a uma ESP32 usando o aplicativo Blynk via rede Wi-Fi. O controle é feito por sliders e botões virtuais no app.

Principais componentes:

- ESP32Servo: biblioteca para controlar servomotores com a ESP32.
- Blynk: plataforma para criar interfaces de controle (botões, sliders, etc.) em smartphones.
- Wi-Fi: conexão com a rede "alunos-01"(sem senha, neste caso).

1 Como o código funciona

1.1 Setup inicial

Conecta à rede Wi-Fi.

Inicia a comunicação com o Blynk.

Associa 6 servos aos pinos da ESP32.

1.2 Loop principal

Mantém o Blynk funcionando com Blynk.run().

1.3 Função moveServo

Move o servo do ângulo atual até o desejado suavemente (passo a passo, com delay(15) ms entre os movimentos).

1.4 Interações com o Blynk

Um botão no pino virtual V4 ativa/desativa um modo alternativo de controle (armazenado na variável buttonState).

Sliders nos pinos virtuais V0 a V3 controlam pares de servos:

V0: controla servo1 ou servo5, dependendo do estado do botão.

V1: controla servo2 ou servo6.

V2 e V3: só funcionam se o botão estiver ativado, controlando servo3 e servo4, respectivamente.

Em resumo: O app Blynk envia comandos via Wi-Fi para a ESP32.

Um botão virtual muda entre dois "modos"de controle.

Sliders controlam os servos conforme o modo selecionado.

2 Ideias de melhorias

1. Tornar o movimento simultâneo entre servos
2. Filtragem de entrada (evitar comandos desnecessários)
3. Modularização
4. Adicionar interpolação com aceleração/desaceleração

2 Como implementar comando de voz

Uma opção a ser considerada é a utilização do Vosk, que é uma biblioteca de reconhecimento de voz offline, ou seja, não precisa de internet. Ela funciona em várias plataformas (Windows, Linux, Android, Raspberry Pi, etc.) e pode ser usada com Python. Ela converte fala em texto.

Em conjunto com o Vosk, temos o uso do MQTT(Message Queuing Telemetry Transport), que é um protocolo leve de mensagens muito usado em IoT, ideal para comunicação entre dispositivos como a ESP32 e um computador/celular.

Ele se dar por:

- Broker: servidor intermediário (por exemplo, Mosquitto).
- Publisher: envia mensagens (ex: PC com Vosk).
- Subscriber: recebe mensagens (ex: ESP32).

1 Como unir Vosk, MQTT e a ESP32

Podemos ter o seguinte fluxo:

1. O PC ou celular capta o áudio (com Vosk).
2. O Vosk reconhece o que foi dito e transforma em texto.
3. Esse texto é interpretado por um script Python que envia um comando via MQTT.
4. A ESP32 escuta (subscribe) esse comando e movimenta os servos de acordo.

Temos então o seguinte fluxograma:

[Usuário Fala]

↓

[Reconhecimento de Voz com Vosk]

↓

[Script Python interpreta o comando]

↓

[Comando enviado via MQTT]

↓

[ESP32 recebe o comando]

↓

[Servomotores executam a ação]

2 Testagem 1

Pré-requisitos

Python:

-vosk

-pyaudio

-paho-mqtt

(pip install pipwin) (pipwin install pyaudio) (pip install vosk pyaudio paho-mqtt)

Modelo de voz do Vosk (https://alphacephei.com/vosk/models, escolha vosk-model-small-pt-0.3.zip)

Arduino IDE:

-ESP32 instalada

-Biblioteca PubSubClient (para MQTT)

-Biblioteca ESP32Servo

2.1 O que vai acontecer

1. O script em Python usa Vosk para transformar sua voz em texto.
2. Esse texto (ex: "servo 90") é enviado para um broker MQTT.
3. A ESP32 fica escutando o tópico e move o servo motor para o ângulo informado.

2.2 Códigos da testagem

Vamos fazer uma testagem utilizando apenas um servo de acordo com o que foi falado acima.

Python:

```python
# Importa bibliotecas para reconhecimento de voz e MQTT
from vosk import Model, KaldiRecognizer
import pyaudio
import json
import paho.mqtt.client as mqtt

# Configurações MQTT
broker = "broker.hivemq.com"
topico = "braco/comando"
cliente_mqtt = mqtt.Client()
cliente_mqtt.connect(broker, 1883)

# Carrega o modelo Vosk (você deve baixar e extrair a pasta "model")
model = Model("model")  # substitua por caminho correto da pasta do modelo
rec = KaldiRecognizer(model, 16000)

# Configura microfone
p = pyaudio.PyAudio()
stream = p.open(format=pyaudio.paInt16, channels=1, rate=16000,
                input=True, frames_per_buffer=8000)
stream.start_stream()

print("Fale um comando como 'servo 90'")

# Loop de escuta
while True:
    data = stream.read(4000, exception_on_overflow=False)
    if rec.AcceptWaveform(data):
        resultado = json.loads(rec.Result())
        texto = resultado.get("text", "").strip()
        print("Texto reconhecido:", texto)

        # Envia comando se não for vazio
        if texto:
            cliente_mqtt.publish(topico, texto)
```

ESP32:

```
#include <WiFi.h>
#include <PubSubClient.h> //instala
#include <ESP32Servo.h> //instala

const char* ssid = "SUA_REDE_WIFI";           // Altere para sua rede WiFi
const char* password = "SUA_SENHA_WIFI";      // Altere para sua senha WiFi
```

```
const char* mqtt_server = "broker.hivemq.com";

WiFiClient espClient;
PubSubClient client(espClient);
Servo servo;
int pinoServo = 2;  // GPIO onde o servo está ligado

void setup_wifi() {
  delay(100);
  Serial.print("Conectando-se a ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("WiFi conectado.");
}

void callback(char* topic, byte* payload, unsigned int length) {
  String comando = "";
  for (int i = 0; i < length; i++) {
    comando += (char)payload[i];
  }

  Serial.print("Comando recebido: ");
  Serial.println(comando);

  if (comando.startsWith("servo ")) {
    int angulo = comando.substring(6).toInt(); // Extrai número após "
       servo "
    angulo = constrain(angulo, 0, 180);        // Garante valores vá
       lidos
    servo.write(angulo);
    Serial.print("Servo movido para: ");
    Serial.println(angulo);
  }
}

void reconnect() {
  while (!client.connected()) {
    Serial.print("Tentando conexão MQTT...");
    if (client.connect("ESP32Client")) {
      Serial.println("conectado!");
      client.subscribe("braco/comando"); // Topico que escutamos
    } else {
      Serial.print("Falha, rc=");
      Serial.print(client.state());
      delay(2000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  servo.attach(pinoServo);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
```

```
  client.setCallback(callback);
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();  // Mantém conexão ativa
}
```

Configure o Wi-Fi no código do ESP32, rode o script Python no computador com microfone, fale um comando como: servo noventa (Vosk converte "noventa"em 90). O servo motor deve ir para a posição indicada.

2.3 Comparativo

Agora vamos fazer alguns comparativos do novo código com o anterior:

1. Comunicação

Recurso	Código com Blynk	MQTT + Vosk
Bibliotecas	`Blynk`, `BlynkSimpleEsp32`	`WiFi.h`, `PubSubClient.h`
Interface de controle	App do Blynk (sliders)	Microfone + comandos de voz
Transmissão de dados	Via app (nuvem Blynk)	Via broker MQTT
Servidor usado	Blynk Cloud	HiveMQ ou Mosquitto

2. Recebimento e Interpretação do Comando

Função	Blynk (Widgets)	Vosk + MQTT (Texto)
`BLYNK_WRITE(V0)`	Executa com slider	Não existe, substituído por MQTT
`param.asInt()`	Lê valor do widget	Valor vem em string (ex: "servo 90")
`moveServo(...)`	Usada	Continua sendo usada

3. Forma de Enviar Comando

Quem Envia	Blynk	Vosk + MQTT
App Blynk	Envia sliders	Não utilizado
Script Python	Não existe	Usa Vosk + PyAudio + MQTT
ESP32	Reage ao app	Reage ao broker MQTT

4. Flexibilidade e Expansão

Aspecto	Blynk	MQTT + Vosk
App externo necessário	Sim (Blynk App)	Não (roda localmente)
Customização por voz	Limitada aos widgets	Alta, comandos por voz podem ser expandidos
Expansão de comandos	Depende da interface gráfica	Pode aceitar comandos complexos via script
Dependência de nuvem	Sim (servidor Blynk)	Não (broker local possível)

Tabela 1: Comparação da flexibilidade entre Blynk e Vosk + MQTT

5. Exemplo prático em código

-Comando de voz "servo 90"via MQTT

```
// Código que reage ao comando "servo 90"
if (comando.startsWith("servo ")) {
    int angulo = comando.substring(6).toInt(); // Extrai o número
    servo.write(angulo); // Move o servo
}
```

3 Testagem 2 - melhorias no reconhecimento

Testamos o reconhecimento anterior com um dicionário de palavras mais específico e percebemos que o reconhecimento melhorou:

```
   import json
from pathlib import Path
from vosk import Model, KaldiRecognizer
import pyaudio
import paho.mqtt.client as mqtt

# -------- CONFIGURAÇÕES --------

# Pasta do modelo Vosk (ajuste o caminho para onde seu modelo está)
BASE_DIR = Path(__file__).resolve().parent
MODEL_DIR = BASE_DIR / "model_vosk"  # sua pasta do modelo em português

# Lista de palavras esperadas (gramática)
palavras = ["servo", "servo motor", "mover", "ângulo", "posicao", "posição",
            "um", "dois", "tres", "quarenta e seis" "quatro", "cinco", "seis", "sete", "oito", "nove", "zero",
            "90", "45", "180", "seabra", "letícia", "filipe", "mateus", "natan", "léo", "sair"]

# Inicializa modelo e reconhecedor
model = Model("/home/natan/Área de trabalho/recvoz/modelo_vosk")
rec = KaldiRecognizer(model, 16000, json.dumps(palavras))

# MQTT
broker = "broker.hivemq.com"
```

```
topico = "braco/comando"
cliente_mqtt = mqtt.Client()
cliente_mqtt.connect(broker, 1883)
cliente_mqtt.loop_start()

# Áudio (microfone)
p = pyaudio.PyAudio()
stream = p.open(format=pyaudio.paInt16,
                channels=1,
                rate=16000,
                input=True,
                frames_per_buffer=4000)
stream.start_stream()

print("Fale um comando tipo 'servo 90' ou 'ângulo 45'. Diga 'sair' para encerrar.")

try:
    while True:
        data = stream.read(4000, exception_on_overflow=False)
        if rec.AcceptWaveform(data):
            resultado = json.loads(rec.Result())
            texto = resultado.get("text", "").strip()
            if texto:
                print("Texto reconhecido:", texto)
                if "sair" in texto:
                    print("Comando 'sair' recebido, finalizando...")
                    break
                cliente_mqtt.publish(topico, texto)
        else:
            # Resultado parcial (pode ajudar para debug)
            parcial = json.loads(rec.PartialResult())
            texto_parcial = parcial.get("partial", "").strip()
            if texto_parcial:
                print("Reconhecimento parcial:", texto_parcial)

finally:
    print("Encerrando stream e conexão MQTT...")
    stream.stop_stream()
    stream.close()
    p.terminate()
    cliente_mqtt.loop_stop()
    cliente_mqtt.disconnect()
    print("Programa finalizado.")
```

4 Testagem 3 - implementação da gramática

Com o melhoramento da gramática como no código anterior, demos início a uma testagem usando gramáticas mais robustas para o nosso projeto. Vamos esmiuçar detalhadamente esse processo abaixo:

4.1 Arquivo voskoff

Começamos deszipando o arquivo voskoff.zip, nele temos as seguintes pastas e código em python:

- **entrada**:

  contém um arquivo chamado **words.txt**, ele é basicamente o dicionário do nosso sistema, contendo uma lista de palavras que o sistema é capaz de reconhecer. Ele converte as palavras em números (IDs)

para o processamento. É importante destacar as seguintes palavras e IDs:

-<esp> 0: vazio, usado no processamento interno

-!SIL 99089: silêncio (pra pausas na fala)

[unk] 99090: palavra desconhecida por não estar no vocabulário

- **grammar**

Nessa pasta temos 4 testes no formato .jsgf, esse formato é usado para criar regras de reconhecimento de voz definindo combinações válidas de palavras, que serão escolhidas por nós e serão entendidas pelo braço. o JSGF não faz conversão, só define a estrutura. É um modelo de frase.

estrutura:

```
#JSGF V1.0;          // Cabeçalho obrigatório
grammar comando;     // Nome da gramática

// REGRAS:
public <comando> = (mova para frente) | (abra a garra);
```

usando o JSGF personalizado nós temos um foco apenas para os comandos do robô, sendo esses comandos estruturados, porntos para o uso, filtrando ruídos, e com latência menor

Agora vamos ver o que cada teste faz:

– teste0(Comandos Direcionais):

```
<cDireita> = ([cento e] quarenta);      // "cento e" é opcional
<cEsquerda> = esquerda | direita;       // "esquerda" OU "direita"
<cFrente> = frente {um};                // "frente" seguido
    opcionalmente de "um"
<ruido> = "<unk>";                      // Palavra desconhecida

public <comandosTestagem3> = (
    <cDireita> | <cEsquerda>+ | <cFrente> | <cTras> | <ruido>
); //esquerda+ eu posso ter esquerda esquerda seguidos
```

Temos que os comandos dentro dos <> funcionam como uma palavra chave e tudo após o "="são seus sinônimos/ações realizadas por ele, tipo:

<mover> = andar | correr | deslizar

Ou seja, o cDireita iria capturar a palavra quarente, sendo o "cento e"opcional, o <cEsquerda> a palvra direita ou esquerda e por aí vai. Com o detalhe que frenteum esse um seria um peso opcional, ou sejam quando o sistema ouve frente, ele entenderia tanto frente quanto frente um (ignora esse um).

O public é a regra principal que define todas as opções de comandos válidos

– teste1 (Comandos Numéricos):

Movendo só um servo, só dizendo a angulação e pronto.

```
#JSGF V1.0 UTF-8;
grammar grammar.teste1;

<unidade> = cinco;
<zero> = zero;

<dezena> = (dez | vinte | trinta | quarenta | cinquenta | sessenta |
    setenta | oitenta | noventa);
<dezenasPrimarias> = quinze;


<centenas_agrupadas> = cento;
<centenas> = cem;
<ruido> = "<unk>";
```

```
14
15 public <juncao> =  (([<centenas_agrupadas> e]  [<dezena> e] <unidade
       >)  | ( [<centenas_agrupadas> e] (<dezenasPrimarias> | <dezena>))
         | <centenas> | <zero> |<ruido>);
```

Agora temos números compostos até centenas e combinações variadas, aceitando omissões do "e"

– teste2 (Comandos de Controle do Robô):

Mover todos os servos separadamente, dando o nome do servo e do angulo.

```
1     #JSGF V1.0 UTF-8;
2  grammar grammar.teste2;
3
4  <unidade> = cinco;
5  <zero> = zero;
6  <dezena> = (dez | vinte | trinta | quarenta | cinquenta | sessenta |
       setenta | oitenta | noventa);
7  <dezenasPrimarias> = quinze;
8  <centenas_agrupadas> = cento;
9  <centenas> = cem;
10 <ruido> = "<unk>";
11 <juncao> =  (([<centenas_agrupadas> e]  [<dezena> e] <unidade>)  | (
       [<centenas_agrupadas> e] (<dezenasPrimarias> | <dezena>))  | <
      centenas> | <zero>);
12
13
14
15 <servo> = (um | dois | três | quatro | cinco);
16
17 <garra> = (pegar | fechar | abrir | largar);
18
19 <pC1> = (servo | motor | mova | o);
20 <pC2> = (para);
21 <pC3> = (graus);
22
23 public <comandosTestagem> = (S
24     (
25         <pC1>*
26         <servo>
27         [<pC2>]
28         <juncao>
29         [<pC3>]
30     )
31     |
32     (
33         <garra>
34     )
35     |
36     (
37         <ruido>
38     )
39
40 );
```

Agora temos uma seleção explícita dos servos e o suporte para dois comandos tipos de comando, o servo+angulo e ações da garra. ELe aceita variações na fala e tolera frasem como por exemplo que não tenham a palavra "graus"e ignora sons irreconhecíveis.

– teste3 (Comandos Direcionais Simplificados)

A lógica é mover os servos mais de um por vez. Vai ser mais díficil

```
#JSGF V1.0 UTF-8;
grammar grammar.teste3;

<cDireita> = (direita);
<cEsquerda> = (esquerda);
<cFrente> = (frente);
<cTras> = (trás);
<cCima> = (cima);
<cBaixo> = (baixo);
<garra> = (pegar | fechar | abrir | largar);
<ruido> = "<unk>";


public <comandosTestagem3> = (
    <cBaixo> | <cCima> | <cDireita> | <cEsquerda> | <cFrente> | <
        cTras> | <garra> | <ruido>
);
```

Aqui voltamos a ter uma ação direta e única, com palavras isoladas, usado em controle de tempo real para ambientes ruidosos e treinamento rápido sem implementar em qual servo será e etc. ao contrário do teste 2 que é usado para configuração dos Ângulos esse é o controle em tempo real

- **modelotest0**

nele vamos ter uma pasta chamada ivector, que possui:

- final.dubm: modelo UBM diagonal usado na extração de i-vector, é uma GMM que representa distribuições acústicas gerais. é um modelo global do som de fala genérico o qual cada fala específica é comparada
- final.ie: arquivo extrator de ivector, contém parametros treinados que computam o vetor caracteŕisticos do louctor a partir do áudio usando o UBM acima
- final.mat: aplica transformação linear nas características ante de extrair o i-vector, melhorando a representação
- onlinecmvn.conf: É usado tanto na extração de i-vectors quanto durante o reconhecimento online. Basicamente diz "como normalizar cada frame de áudio de modo adaptativo".
- splice.conf: : configura opções de "splicing", ou seja, agrupar múltiplos frames de características para formar vetores maiores com contexto. Especifica quantos frames anteriores e posteriores, usado na geração de ivector.
- globalcmvn.stats: sado para ajustar as características acústicas ao falante. Este arquivo contém a média e a variância de todas as características de treinamento (em formato "ark"), utilizado para normalização (CMVN) durante a extração de i-vectors.

Além da pasta ivector temos também:

-disambingtid.int: : contém símbolos de desambiguação do léxico. São marcadores especiais (como 0, 1...) inseridos no léxico FST para resolver ambiguidades na estrutura das palavras. Em geral, é uma tabela gerada automaticamente para garantir que cada caminho no transdutor de léxico seja unívoco. (Este arquivo é um detalhe interno do FST, pouco usado manualmente.)

-final.mdl: o modelo acústico final treinado (neste caso GMM-HMM ou DNN-HMM). Contém todos os parâmetros do modelo que converte características acústicas (MFCCs) em probabilidades de fonemas. Em Kaldi, um final.mdl inclui a parte de transição (TransitionModel) e as distribuições (GMMs ou pesos da rede). Em termos simples, é o "cérebro" que ouve as características e estima quais sons/fones estão sendo pronunciados. Ele foi produzido no fim do treinamento e é exigido para o reconhecimento

-Gr.fst e HCLr.fst: : juntos formam o grafo de decodificação dinâmico (equivalente ao HCLG.fst tradicional). Em Kaldi usual, HCLG.fst combina H (modelo acústico/transições), C (contexto de trífono), L (léxico) e G (modelo de linguagem/gramática) num único FST. No formato Vosk/Kaldi dinâmico, usa-se HCLr.fst (que contém o modelo acústico+léxico) e Gr.fst (o modelo de gramática/palavras). A vantagem é permitir atualizar a gramática dinamicamente (útil para rescoring). O documento oficial diz: "use Gr.fst e HCLr.fst em vez de um único HCLG.fst se quiser fazer rescoring". Na prática, durante

a decodificação o Kaldi junta esses FSTs. O HCLr.fst é o "mapa de todas as transições possíveis" dado o modelo acústico e léxico, e o Gr.fst restringe as sequências de palavras permitidas pela gramática.

-mfcc.conf: arquivo de configuração dos coeficientes cepstrais (MFCC). Define parâmetros como taxa de amostragem, número de filtros Mel, janela de análise, etc. Por exemplo, pode especificar –sample-frequency=16000, número de filtros, etc. Isso diz ao Kaldi como extrair as características de áudio iniciais a partir da onda sonora.

-phones.txt: : tabela de fones (símbolos fonéticos) usados pelo modelo. Cada linha mapeia um símbolo de fonema a um número (ID).Por exemplo, <eps> 0, SIL 1, etc. Um fonema é a menor unidade de som no idioma. O reconhecimento trabalha em fones para depois formar palavras. Assim como words.txt lista palavras, phones.txt lista sons básicos (e.g. "AA", "B", "D" na tabela CMU) usados no modelo acústico kaldi-asr.org.

-wordboundary.int: (gerado pela preparação do idioma) contém marcações de onde as palavras começam/terminam em termos de fones, usado internamente para alinhamento de palavra. É criado se o sistema usar fones dependentes de posição (início, meio, fim de palavra) e ajuda na etapa de alinhamento de palavras (por exemplo, para alocação de tempo da palavra).

-words.txt: o mesmo mencionando anteriormente.

- **modelotest1**

  -Possui os mesmos arquivos do modelo 0.

- **modelotest2**

  -Possui os mesmos arquivos do modelo 0.

- **modelotest3**

  -Possui os mesmos arquivos do modelo 0.

- **modelovoz-Copia-Copia**

  -Possui os mesmos arquivos do modelo 0.

- **rec.py**

  -temos o seguinte código:

```python
from pathlib import Path
from vosk import Model, KaldiRecognizer
import pyaudio
import json



# Caminhos para o modelo e o FST da gramática
DIR_BASE = Path(__file__).resolve().parent
MODEL_DIR = DIR_BASE / "modelo_test3"
model = str(MODEL_DIR)
grammar_path = str(MODEL_DIR / "Gr.fst")  # Ajuste o caminho se necessário


model = Model(model)

rec = KaldiRecognizer(model, 16000)

p = pyaudio.PyAudio()
stream = p.open(format=pyaudio.paInt16,
                channels=1,
                rate=16000,
                input=True,
                frames_per_buffer=8192)
stream.start_stream()
print('iniciando o reconhecedor\n')
```

```
while True:
    data = stream.read(4096, exception_on_overflow=False)
    if rec.AcceptWaveform(data):
        resultado = json.loads(rec.Result())
        texto = resultado.get('text', '').strip()
        print('Texto reconhecido: ', texto)
```

Esse código vai:
-Escuta o microfone.
-Usa o Vosk com um modelo de linguagem completo.
-Reconhece fala contínua.
-Exibe o texto reconhecido
Problemas:
Não está usando a gramática Gr.fst.
O caminho grammar-path é declarado, mas não está sendo passado para o reconhecedor.
O Vosk não usa diretamente arquivos Gr.fst com KaldiRecognizer.
Não trata erros e interrupções com segurança.
Não há nenhuma verificação se o modelo foi carregado corretamente

4.2 Implementação do tratamento de voz

```
#------------------importações--------------------
import json # tradutor de formatos json
import os
from pathlib import Path #pra encontrar as pastas
from vosk import Model, KaldiRecognizer #nosso modelo (um mapa de sons) e nosso reconhecedor (o kaldi)
import pyaudio #pra capturar audio
import paho.mqtt.client as mqtt #transmissão de mensagens


#------------------configurações globais ------------------------------

BASE_DIR = Path(__file__).resolve().parent #é o path deste arquivo python usando o resolve).parent pra calcular o caminho absoluto
                                           #BASE_DIR é o diretorio base do projeto

BROKER = "broker.hivemq.com"  #nosso servidor gratuiro pra central de mensagens
TOPICO = "braco/comando"  #este vai ser o endereço especifico onde iremos escutar, pra onde a mensagem sera encaminhada, os publishers sao os dispositivos que enviam essas mensagens
TESTE_ATIVO =  2 #ESCOLHA 1 2 ou 3

#------------------paths pra cada modelo implementado -----------------

MODEL_PATHS = {
    1: BASE_DIR / "modelo_test1",  # Setor de angulos precisos
    2: BASE_DIR / "modelo_test2",  # Setor pra varios servos
    3: BASE_DIR / "modelo_test3"   # Setor de comandos rapidos
} # este dicionário vai mapear o teste pra pasta do modelo

#------------------dicionários----------------------------------------

```

```
NUMEROS_MAP = {
    "zero": 0, "cinco": 5, "dez": 10, "quinze": 15, "vinte": 20,
    "vinte e cinco": 25, "trinta": 30, "trinta e cinco": 35,
    "quarenta": 40, "quarenta e cinco": 45, "cinquenta": 50,
    "cinquenta e cinco": 55, "sessenta": 60, "sessenta e cinco": 65,
    "setenta": 70, "setenta e cinco": 75, "oitenta": 80,
    "oitenta e cinco": 85, "noventa": 90, "noventa e cinco": 95,
    "cem": 100, "cento e cinco": 105, "cento e dez": 110,
    "cento e quinze": 115, "cento e vinte": 120, "cento e vinte e cinco": 125,
    "cento e trinta": 130, "cento e trinta e cinco": 135,
    "cento e quarenta": 140, "cento e quarenta e cinco": 145,
    "cento e cinquenta": 150, "cento e cinquenta e cinco": 155,
    "cento e sessenta": 160, "cento e sessenta e cinco": 165,
    "cento e setenta": 170, "cento e setenta e cinco": 175,
    "cento e oitenta": 180
}
#traduz a palavra pro numero

SERVO_MAP = {
    "um": 1, "dois": 2, "três": 3, "quatro": 4, "cinco": 5
}
#traduz a palavra pro id do servo

DIRECAO_MAP = {
    "cima": "UP", "baixo": "DOWN", "direita": "RIGHT",
    "esquerda": "LEFT", "frente": "FORWARD", "trás": "BACKWARD"
}
#traduz direção pra direção no código

GARRA_MAP = {
    "abrir": "OPEN", "fechar": "CLOSE", "pegar": "GRAB", "largar": "RELEASE"
}
#traduz ação pra ação no código


#----------------------funções de processamento ----------------------------

def extrair_valor_numerico(texto):
    """Encontra a sequência numérica mais longa no texto"""
    palavras = texto.split() #o texto vira uma lista de palavras,divide cento e trinte etc em cento, e, trinta, ...
    melhor_seq = "" #guarda a melhor sequencia
    melhor_valor = 0 #guarda o valor numerico dessa melhor sequencia

    # procura sequências de 1 a 5 palavras que valem pros mapas, sendo 5 o nosso maximo de palavras numa frase de um numero
    #ex: cento e (alguma coisa) e (alguma coisa) -> 5 palavras
    for comprimento in range(5, 0, -1): # vou de 5 ate zero diminuindo 1 por vez
        for i in range(len(palavras) - comprimento + 1): #isso aqui é uma janela, vou explicar melhor abaixo:
            #se eu falar cento e vinte e cinco graus serao 6 palavras:
            #começando com comprimento = 5 (que é o maior numero de palavras de um numero que estamos analisando) temos que esse range vai ser (2) pois 6 - 5 + 1 = 2, pra i = 0 e i
```

```
                = 1
            #o proximo comprimento teremos (3), que vai ser i =0, i=1, i=2
            sequencia = " ".join(palavras[i:i+comprimento]) #desplinta o splint
            #aqui pro comprimento 5 a gente vai ter no i=0 -> join [0:0+5] ai ele junta da palavra
            if sequencia in NUMEROS_MAP: #ve se a sequencia esta no dicionario
                valor = NUMEROS_MAP[sequencia] #atribui ela ao nosso valor
                # prefere sequências mais longas (mais específicas)
                if comprimento > len(melhor_seq.split()):
                    melhor_seq = sequencia
                    melhor_valor = valor

    return melhor_valor
#sobre a janela:
"""
Se o comprimento for 5 e a len de palavras for 5 nossa formula fica 5 - 5 + 1 = 1, logo teremos 1 janela
com o unico exemplo de [0:5], agora se trocamos o nosso comprimento pra 4 e mantermos a len de palavras obtemos 2 janelas
[0:4] e [1:5] ja que 5 - 4 + 1 = 2, agora se eu tiver um comprimento de 3 (ou seja teria 3 palavras no nosso numero)
eu teria 5-3+1 = 3 janelas de intervalos pra percorrer procurando um numero que esteja nos mapas, ou seja teriamos 3 exemplos de palavras em:
[0:3], [1:4], [2:5]

ou seja, se eu falei uma frase com um numero x de palavras, primeiro a gente ve nessa frase temos um numero
que tenha o comprimento de 5 palavras e o nosso range seria x - 5 + 1 (numero esse que indica o quanto a janela desliza).
eu poderia falar uma frase de tipo 8 coisas podendo o numero que eu quero começar na posição 0, 1, 2 ou 3 (janela = 8-5+1 = 4)
ai ele iria ver se em cada deslize a gnt acharia o numero legivel com o if sequencia in NUMEROS_MAP, SE eu nao achei
o numero no dicionario, eu deslizaria uma unidade e veria se a nova sequencia existe novamente.
se ela existe ai atribuimos essa sequencia numa variavel valor.
Finalmente, apos um if pra ver se o comprimento do numero que foi entendido for maior que o split da melhor_seq, que no inicio nao tem nada,
entao ele sera verdadeiro e vai so adicionar a sequencia e o valor correspondente a ela no melhor_seq e melhor_valor e depois retornar o melhor valor
"""

def processar_teste1(texto):
    # ignora ruído
    if "<unk>" in texto or not texto.strip():
        return None

    if any(p in texto for p in ["sair", "parar", "encerrar", "finalizar"]):
        return "SAIR"

```

```
    valor = extrair_valor_numerico(texto)

    # aplica limites na angulação ja que nao pode ser menor que zero e maior que 180 (VAMOS MUDAR ISSO FUTURAMENTE APOS A TESTAGEM DE CADA ANGULAÇÃO)
    valor = max(0, min(valor, 180))

    if valor > 0:
        return f"SERVO:{valor}" #consideraremos que cada servo testado no teste 1 seja o servo 1, mesmo que nao seja
    return None

def processar_teste2(texto):
    # ignora ruído
    if "<unk>" in texto or not texto.strip(): # o strip ignora espaços pra evitar processamento
        return None

    # comando de saída (sair, parar, encerrar...)
    if any(p in texto for p in ["sair", "parar", "encerrar", "finalizar"]):
        return "SAIR"

    # processa comandos de garra
    for palavra, acao in GARRA_MAP.items():
        if palavra in texto:
            return f"GARRA:{acao}"

    # processa comandos de servo
    servo_id = None
    for palavra, num in SERVO_MAP.items():
        if palavra in texto:
            servo_id = num
            break

    valor = extrair_valor_numerico(texto)
    valor = max(0, min(valor, 180))

    if servo_id is not None and valor > 0:
        return f"SERVO{servo_id}:{valor}"
    elif servo_id is None and any(p in texto for p in NUMEROS_MAP):
        return "ERRO:Servo não especificado"

    return None


def processar_teste3(texto): #pode ser melhor implementado no arduino IDE
    """processa comandos diretos de direção e garra"""
    # ignora ruído
    if "<unk>" in texto or not texto.strip():
        return None

    if texto in DIRECAO_MAP:
        return f"DIR:{DIRECAO_MAP[texto]}"
    p
    if texto in GARRA_MAP:
        return f"GARRA:{GARRA_MAP[texto]}"
```

```
    return None

#-----------------configurações do recvoz
    ---------------------------------

#abaixo basicamente temos o padrao de config do recvoz
model_path = MODEL_PATHS.get(TESTE_ATIVO, MODEL_PATHS[2])
model = Model(str(model_path))
rec = KaldiRecognizer(model, 16000) # quando inicializamos o
    reconhecedor vosk, ele automaticamente carrega a gramatica compilada
     Gr.fst que esta no diretorio do modelo


"""
o vosk vai procurar no diretorio do modelo, seja ele o test 1 2 ou 3
    pelos arquivos Gr.fst e pelo HCLr.fst
"""

# configura MQTT
cliente_mqtt = mqtt.Client()
cliente_mqtt.connect(BROKER, 1883)
cliente_mqtt.loop_start()

# configura pyaudio
p = pyaudio.PyAudio()
stream = p.open(
    format=pyaudio.paInt16,
    channels=1,
    rate=16000,
    input=True,
    frames_per_buffer=4000
)
stream.start_stream()

print(f"Sistema iniciado no modo Teste {TESTE_ATIVO}")
print("Aguardando comandos...")

#-------------------loop com tratamento
    ---------------------------------

try:  # até receber "sair"
    while True:
        data = stream.read(4000, exception_on_overflow=False)

        if rec.AcceptWaveform(data):
            resultado = json.loads(rec.Result())
            texto = resultado.get("text", "").strip()

            # limpa o texto de ruídos e marcações do Vosk
            texto_limpo = (
                texto.replace("!SIL", "")
                     .replace("<unk>", "")
                     .strip()
            )

            if not texto_limpo:
                print("Ruído ignorado")
```

```
                continue

            print(f"\nTexto reconhecido: {texto_limpo}")

            # processa de acordo com o teste ativo
            comando = None
            if TESTE_ATIVO == 1:
                comando = processar_teste1(texto_limpo)
            elif TESTE_ATIVO == 2:
                comando = processar_teste2(texto_limpo)
            elif TESTE_ATIVO == 3:
                comando = processar_teste3(texto_limpo)

            # envia comando via MQTT se for válido
            if comando:
                if comando == "SAIR":
                    print("Encerrando o programa por comando de voz.")
                    break
                if comando.startswith("ERRO:"):
                    print(comando)
                else:
                    print(f"Enviando comando: {comando}")
                    cliente_mqtt.publish(TOPICO, comando)

            # comando de sistema
            if "sair" in texto_limpo.lower():
                print("Comando 'sair' recebido, finalizando...")
                break
        else:
            parcial = json.loads(rec.PartialResult())
            texto_parcial = parcial.get("partial", "").strip()
            texto_parcial_limpo = (
                texto_parcial.replace("!SIL", "")
                             .replace("<unk>", "")
                             .strip()
            )
            if texto_parcial_limpo:
                print(f"\rReconhecendo: {texto_parcial_limpo}", end='',
                    flush=True)

finally:
    print("\nEncerrando sistema...")
    stream.stop_stream()
    stream.close()
    p.terminate()
    cliente_mqtt.loop_stop()
    cliente_mqtt.disconnect()
    print("Programa finalizado.")
```

Figura 1: ...