UNIVERSIDADE FEDERAL DO SUL E SUDESTE DO PARÁ - UNIFESSPA

FACULDADE DE COMPUTAÇÃO E ENGENHARIA ELÉTRICA - FACEEL

201940601001 - ALAIM DE JESUS LEÃO COSTA

201940601010 - KLAUBER ARAUJO SOUSA

201940601025 - MANOEL MALON COSTA DE MOURA

FFT EM TEMPO REAL E AMPLIFICAÇÃO DE SINAIS

Marabá – PA

2022

201940601001 - ALAIM DE JESUS LEÃO COSTA
201940601010 - KLAUBER ARAUJO SOUSA
201940601025 - MANOEL MALON COSTA DE MOURA

FFT EM TEMPO REAL E AMPLIFICAÇÃO DE SINAIS

Relatório técnico apresentado como requisito parcial para obtenção de aprovação na disciplina Processamento Digital de Sinais, no curso de Engenharia de Computação, na Universidade Federal do Sul e Sudeste do Pará.

Prof. Leslye Eras

Marabá – PA
2022

RESUMO

As Séries de Fourier permitem representar muitas funções periódicas como uma soma infinita de exponenciais complexas. A representação por meio de séries é bastante vantajosa quando se deseja aproximar os valores da função, por exemplo, quando a função possui uma fórmula complicada, difícil de calcular exatamente, pois a aproximação pode ser feita pelo truncamento da série. Como veremos uma grande vantagem da representação por Séries de Fourier em relação às séries de potências é que, em dadas circunstâncias, aquela é uniforme e global, isto é, sua convergência, uma vez estabelecida, é válida para todo o domínio da função, enquanto que esta geralmente tem convergência apenas local, isto é, dentro de um intervalo chamado intervalo de convergência que depende da série, mas não da função. A discretização das funções de interesse é bastante desejável em muitas aplicações, em especial aquelas que se baseiam em técnicas digitais. Essa discretização dá origem a Transformada Discreta de Fourier (DFT) que provê uma aproximação muito boa para os coeficientes da Série de Fourier de funções periódicas e em alguns casos, essa aproximação é exata. No entanto, o cálculo da DFT costuma ser realizado por um algoritmo pouco intuitivo, mas muito eficiente chamado Transformada Rápida de Fourier (FFT). Este trabalho tem como objetivo demonstrar uma aplicação da FFT em tempo real através do microfone do computador e a reconstrução do áudio capturado pela as amostras exportadas.

Palavras-chaves: Série. Função. Discretização. Reconstrução.

SUMÁRIO

1. INTRODUÇÃO

A Transformada de Fourier é uma transformada integral que expressa uma função em termos de funções de base sinusoidal. Existem muitas variações relacionadas a esta, dependendo das funções a serem transformadas. A Transformada de Fourier decompõe uma função temporal em frequências, de forma semelhante a um acorde de um instrumento musical, o acorde pode ser deduzido como o conjunto harmônico de notas, assim, o acorde pode ser expresso como a amplitude das notas que o compõe. Em termos leigos, a Transformada de Fourier é uma operação matemática que altera o domínio (eixo x) de um sinal no tempo para frequência.

Logo, existe também a Transformada Discreta de Fourier (DFT), que pode ser descrita da seguinte maneira:

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}}$$

Para determinar a Transformada de Fourier de um sinal discreto x[n], multiplicamos cada um de seus valores por e em função de n, depois, soma-se os resultados obtidos. Se usássemos um método computacional para calcular a Transformada Discreta de Fourier de um sinal, precisaria realizar operações de complexidade $O(N^2)$, que em termos de complexidade de algoritmo, seria inviável realizar esta operação para seguir na implementação do projeto.

1. OBJETIVO GERAL

Tendo em vista tal problema em relação a complexidade do algoritmo DFT, surgiu a ideia da Transformada Rápida de Fourier (FFT). Então, a Transformada Rápida de Fourier (FFT) é um algoritmo otimizado para a implementação do Discrete Fourier Transformation (DFT). Um sinal é amostrado durante um período de tempo e dividido em seus componentes de frequência. Estes componentes são oscilações sinusoidais únicas em frequências distintas, cada um com a sua própria amplitude e fase, como mostrado na figura a seguir:

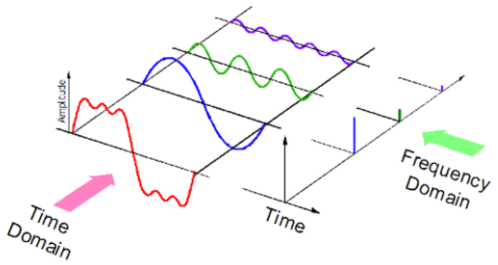


Figura 1 - Decomposição do sinal em frequências.

Para realizar esta transformada, de maneira computacional, uma parte do sinal é escaneada e armazenada na memória para ser processada posteriormente. Com isso, são determinados dois parâmetros: a taxa de amostragem ou frequência de amostragem, que será o número médio de amostras obtidas num segundo; e o segundo parâmetro será o número de amostras selecionadas do sinal. A partir destes parâmetros básicos, pode-se determinar outros parâmetros da medição, como a **largura de banda**: frequência máxima teórica que pode ser determinada pela FFT; a **duração da medição**; e a **resolução de frequência**: indica o espaçamento de frequência entre dois resultados de medição.

Assim, o objetivo geral deste projeto será utilizar os conhecimentos sobre Transformadas de Fourier, especialmente os conceitos sobre Transformada Rápida de Fourier, obtidos durante o curso de processamento digital de sinais para analisar e manipular e amplificar sinais específicos.

2. OBJETIVOS ESPECÍFICOS

O objetivo específico deste projeto está na reconstrução de um sinal de áudio, e na amplificação deste mesmo sinal. Primeiramente, iremos obter um sinal de áudio, este sinal

obtido será inserido no algoritmo por meio da gravação em tempo real, onde o usuário poderá gravar um áudio a partir do momento em que iniciar a execução do algoritmo deste projeto. Logo, após terminar a gravação, será executada a Transformada Rápida de Fourier (FFT) em cima deste sinal de áudio resultante da gravação, a FFT irá decompor o sinal de áudio em suas componentes na frequência. Com o sinal estando em função da frequência, poderá ser manipulado, assim, poderá ser realizada a amplificação deste sinal, como proposto em objetivos gerais.

Uma vez manipulado, o sinal em função da frequência será reconstruído em função do tempo, para que seja notória a manipulação realizada.

3. METODOLOGIA

Para que pudéssemos alcançar nossos objetivos propostos, ao longo do desenvolvimento deste trabalho foi empregado diversas técnicas de colaboração entre a equipe, como encontros em sala de aula e online, a fim de encontrarmos uma melhor maneira de analisarmos um sinal de áudio e assim podermos gerar sua fft e amplificação do sinal gerado. Inicialmente a principal ferramenta utilizada foi dois computadores, um do tipo notebook com sistema operacional Windows e outro do tipo desktop com sistema operacional Linux, ambos pessoais dos membros da equipe. Para o desenvolvimento do programa foi utilizado a linguagem de programação Python na sua versão 3.6, no qual tem vasta aplicação em manipulação de sinais e conta com um modelo comunitário de desenvolvimento ativo de usuários, além de ser uma linguagem orientada a objetos, ágil, fácil e objetiva, o que democratiza seu ensino e a faz ser procurada cada vez mais.

Além disso, foi utilizado também o gerenciador de pacotes Anaconda, que é uma distribuição das linguagens de programação Python para computação científica, que visa simplificar o gerenciamento e implantação de pacotes que irão nos auxiliar durante o desenvolvimento da aplicação. A grande vantagem da linguagem de programação Python é a quantidade de bibliotecas disponíveis que podem ser utilizadas no desenvolvimento de qualquer projeto. Para nos auxiliar na implementação do nosso programa, fizemos uso da biblioteca numpy que tem como função trabalhar com computação numérica, seu principal objeto é o vetor n-dimensional, ou array, um vetor n-dimensional também é conhecido pelo nome de tensor. Também foi utilizada a biblioteca Pygame escrita em linguagem Python e baseada em SDL. Voltada para o desenvolvimento de games e interfaces gráficas, o Pygame fornece acesso a áudios, teclados, controles, mouses e hardwares gráficos via OpenGL. Foi utilizada também a biblioteca Matplotlib, na qual é de software para criação de gráficos e

visualizações de dados em geral, feita para e da linguagem de programação Python e sua extensão de matemática NumPy. Utilizamos também a biblioteca SoundFile no qual é uma biblioteca de áudio baseada em libsndfile, CFFI e NumPy, ela pode ler e gravar arquivos de som suportada por meio de libsndfile, que é uma biblioteca gratuita, multiplataforma e de código aberto (LGPL) para ler e gravar muitos formatos de arquivos de som amostrados diferentes que são executados em muitas plataformas, incluindo Windows. Por fim, utilizamos também a biblioteca numpy.fft que irá nos auxiliar a fazer o cálculo da transformada rápida de fourier, assim como o módulo wave que fornece uma interface conveniente para o formato de som WAV.

4. FUNCIONAMENTO

Este tópico tem como função demostrar toda a base e fundamento teórico em relação ao funcionamento do código e a sua aplicabilidade. Assim também como todo o código criado, suas funções e atribuídos e dependências internas de outras funções.

4.1 Descrição do funcionamento

Como descrito anteriormente, o programa tem como objetivo determinar a transformada rápida de Fourier em tempo real, capturando os sons advindos do microfone conectado ao computador. Além do mais, esses dados são gravados em um arquivo txt com todas as amostras, que posteriormente são utilizadas para reconstruir o sinal original (gerando um arquivo wav) e plotar sua forma e também a transformada rápida de Fourier, ou seja, é aplicada a transformada deste o momento inicial da gravação até o fim e verificada o impacto que ela causa – a visão das frequências, portanto, do sistema inteiro. Outra função em especial, dos códigos é amplificar o sinal reconstruído e comparar com o não amplificado, verificando as características decorridas, ou seja, são gerados dois arquivos, um amplificado e outro não amplificado.

São utilizadas 9 classes, sendo 7 secundárias, contendo todas as funções necessárias e 2 principais responsáveis pela inicialização do espectro em tempo real e outra para reconstrução do áudio. As classes contendo as funções principais estão contidas no diretório **src** do projeto, as quais são denominadas: **fft, stream-analyzer, stream_reader_pyaudio, stream_reader_sounddevice, utils** e **visualizer**. Já as outras classes de inicialização estão no diretório raiz, com o nome de **executar_Analizador_FFT** e **geraAudioeFFT**. Vejamos, em particular, o funcionamento principal de cada uma.

A classe **fft**, como o próprio sugere, aplica a fft em tempo real dos dados obtidos pelo o microfone que são armazenados em um buffer criado de duração de 0,5 segundos – apesar do pouco tempo, a quantidade de amostras é grande, pois é preciso para não perder amostras. Os parâmetros essenciais para gerar a FFT são os dados obtidos pelo microfone e a taxa de amostragem.

A classe **stream_reader_pyaudio** utiliza a biblioteca **pyaudio** para iniciar a gravação e armazenar os dados coletados em um buffer de durção de 0,5 segundos. Contém também, a definição das taxas mais baixas de captura e também as mais altas.

Já a classe **stream_reader_sounddevice** verifica os dispositivos de gravação disponíveis no computador, lista os mesmos e seleciona o primeiro, além de determinar a taxa de amostragem utilizada para a captura dos mesmos.

A classe **utils** contém algumas funções extras para o funcionamento das outras classes, tais como os botões da janela de visualização, a separação das frequências em faixas, alguns filtros disponíveis e suavização e também tem a função de alocação do buffer.

Para demonstrar o gráfico em tempo real, utilizou-se a classe **visualizer** que utiliza o móduelo **pygame** para construir uma janela com os botões e funções, além da visualização dos gráficos, a construção da faixa da frequência, entre outros responsáveis pela a tela.

Com todas as funções secundárias declaradas, é hora de integrar, passando tanto a alocação do buffer utilizado na gravação do microfone através dos dispositivos disponíveis, para a tela de visualização da transformada de Fourier. Para tal objetivo, foi feita a classe **stream_analyzer**, que justamente tem essa função de integrar todas as outras classes explicadas anteriormente, tem o controle e gerenciamento também do sistema e outras funções como a de atualização da tela com o buffer, e verificação de processamento de novos dados. Esta é por sua vez ligada diretamente com a classe principal **executar_Analizador_FFT,** que passa alguns argumentos como o tamanho da janela, a proporção, a taxa de amostragem, a suavização, a taxa de atualização e os quadros por segundos.

Com toda essa integração, o arquivo de txt é gerado pela classe **fft** que depois é utilizado pela a classe **geraAudioeFFT**, fazendo a conversão do arquivo em somente um vetor, reconstruindo o áudio, amplificando e gerando o áudio com esse atribuído, além de plotar os gráficos do áudio e também da FFT toda.

Com toda é descrição, é melhor apresentar um diagrama que contém o esquema de relacionamento entre as classes:

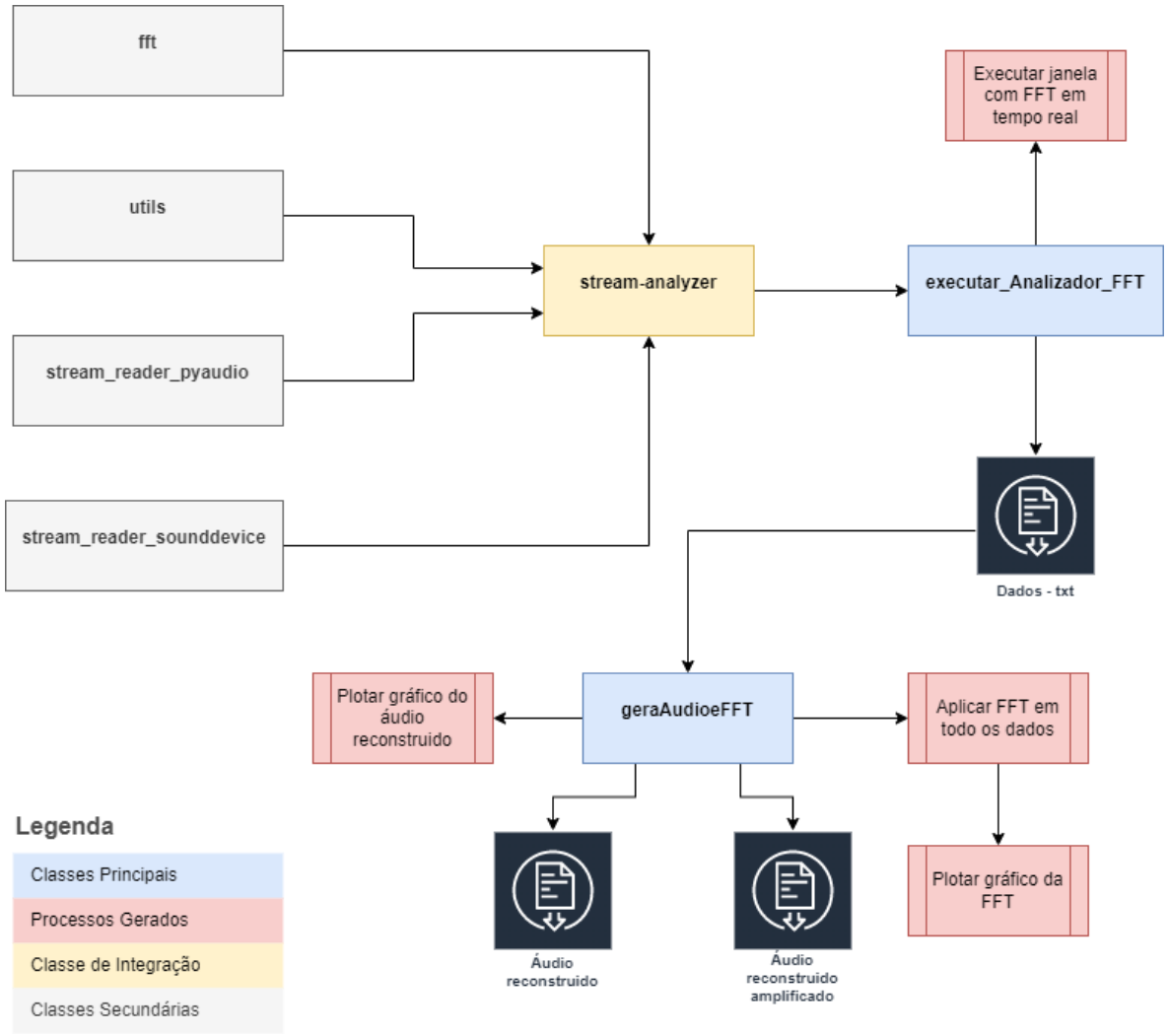


Figura 2 - Fluxo de funcionamento do programa.

Percebe-se todo o envolvimento de geração de arquivos, integração entre as classes, artefatos e o uso dos mesmos. Além do fluxo de execução decorrido deste o acionamento das funções das classes secundárias até a reconstrução dos arquivos e plotagem dos gráficos.

4.2 Código implementado

Com toda a explicação do funcionamento do programa, partiremos para a explanação das funções e consequentemente as classes que as abrange, começando dentre as mais simples até as mais complexas e vitais. Como há muitas classes, é dividido em tópicos para cada uma, iniciando pela as classes secundárias.

5.2.1 fft:

Contém somente uma função responsável por receber os dados e calcular a FFT de acordo com os outros parâmetros repassados.

```
16
17  import numpy as np
18
19  def getFFT(data, rate, chunk_size, log_scale=False):
20      #abre um arquivo com o nome dadosGerados - se não tiver, é criado
21      arquivo = open('dadosGerados.txt', 'a')
22
23      #escreve cada valor no documento
24      for i in range(0, len(data)):
25          arquivo.write(str(data[i])+" ")
26
27      data = data * np.hamming(len(data))
28      try:
29          FFT = np.abs(np.fft.rfft(data)[1:])
30      except:
31          FFT = np.fft.fft(data)
32          left, right = np.split(np.abs(FFT), 2)
33          FFT = np.add(left, right[::-1])
34
35      # fftx = np.fft.fftfreq(chunk_size, d=1.0/rate)
36      # fftx = np.split(np.abs(fftx), 2)[0]
37
38      if log_scale:
39          try:
40              FFT = np.multiply(20, np.log10(FFT))
41          except Exception as e:
42              print('Log(FFT) failed: %s' % str(e))
43
44      return FFT
45
46
```

Figura 3 - Função que calcula a FFT dos dados.

A função **getFFT** recebe primeiramente os dados contidos no buffer (**data**), a taxa de amostragem (**rate**) definido pelo padrão com um valor de 44100, o tamanho da janela apresentada (**chunk_size**) e a aplicação de um log caso seja preferível (**log_scale=False**), neste caso não é aplicado. No corpo da função, abre-se um arquivo com o nome **dadosGerados** que será utilizado para armazenar os dados recebidos do buffer, depois no laço de repetição da linha 24 escreve estes dados. Na linha 27 é aplicada uma função que ajusta o janelamento e depois usa a **rfft**, que é uma função que calcula a transformada de Fourier de uma matriz unidimensional. E caso ocorra algum erro com esta função mais rápida, utiliza a transformada rápida de Fourier tradicional, a **fft**. Na parte da escala, é responsável por utilizar os cálculos necessários para transformar a intensidade do sinal em decibéis.

5.2.2 Stream_reader_pyaudio

Contém algumas funções para determinar o fluxo de áudio advindo do microfone, passando três principais argumentos, **device** que seleciona o qual fluxo de leitura do áudio, **rate**, que é a taxa de amostragem padrão já comentada anteriormente e **updatesPerSecond** que é a frequência de gravação dos novos dados.

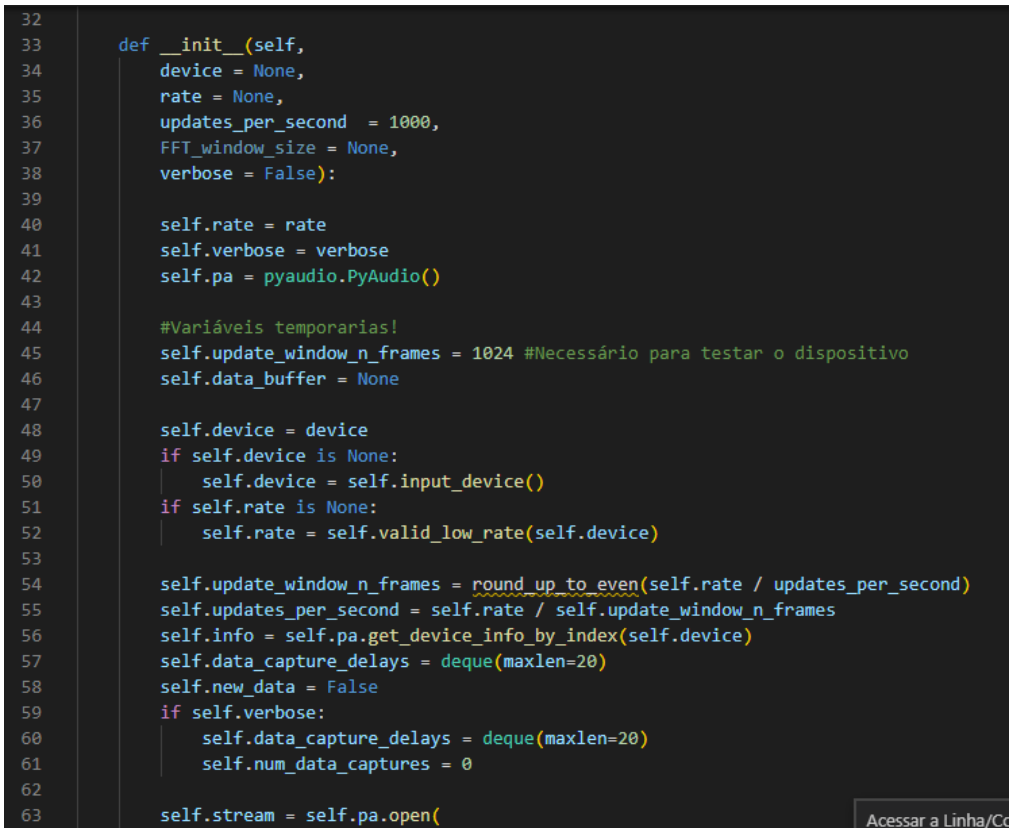


Figura 4 - Função de inicialização da classe stream_reader_pyaudio.

A inicialização é determinada passando os valores para cada variável responsável, depois é criada variáveis temporárias e também é testado o dispositivo para que não ocorra nenhum erro. Posteriormente é selecionado o dispositivo, feito alguns cálculos nas linhas 54 e 55 para a atualização da janela de visualização e as atualizações da própria FFT. Assim também como é demonstrado as informações essenciais na linha 56. Define-se um delay também de 20.

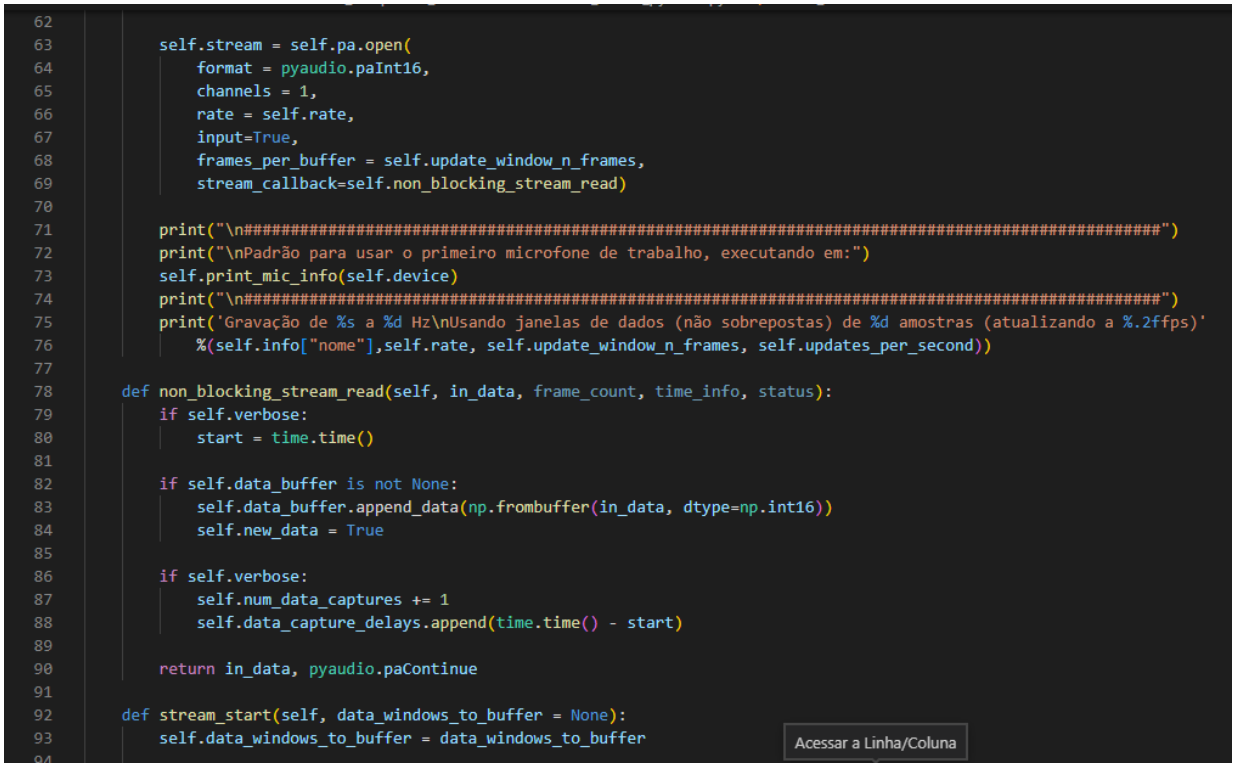
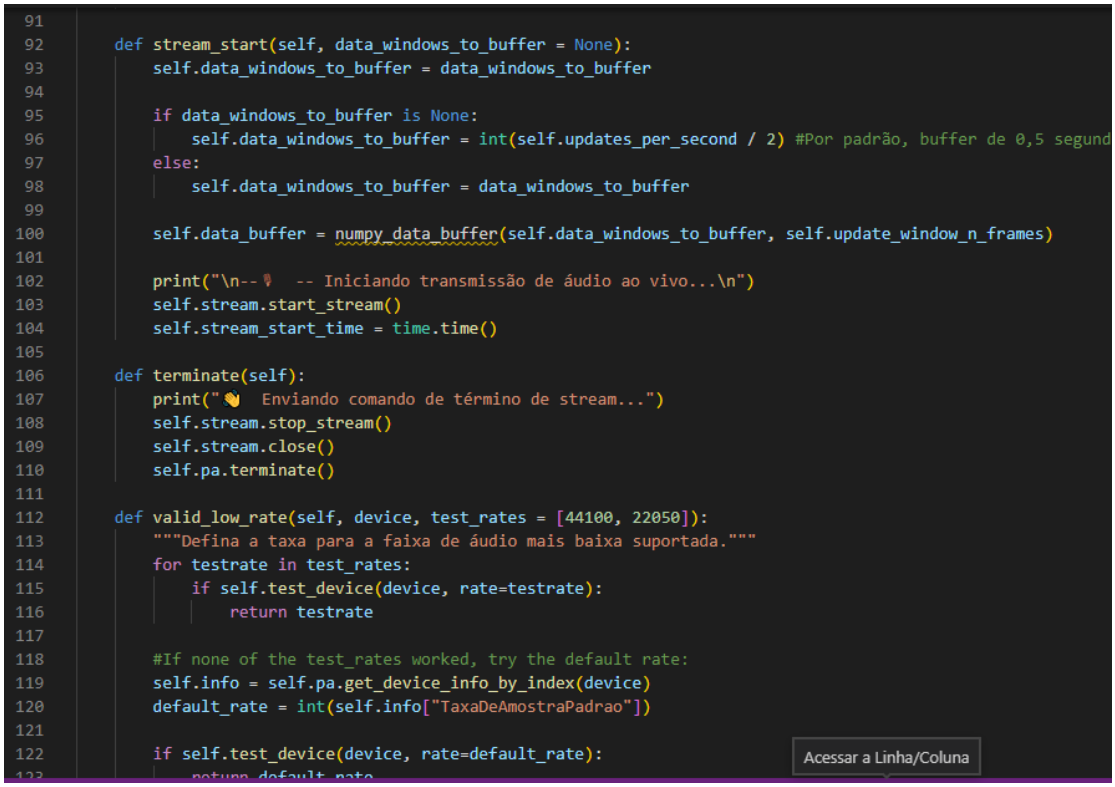


Figura 5 - Funções para reconhecimento do microfone e disponibilidade.

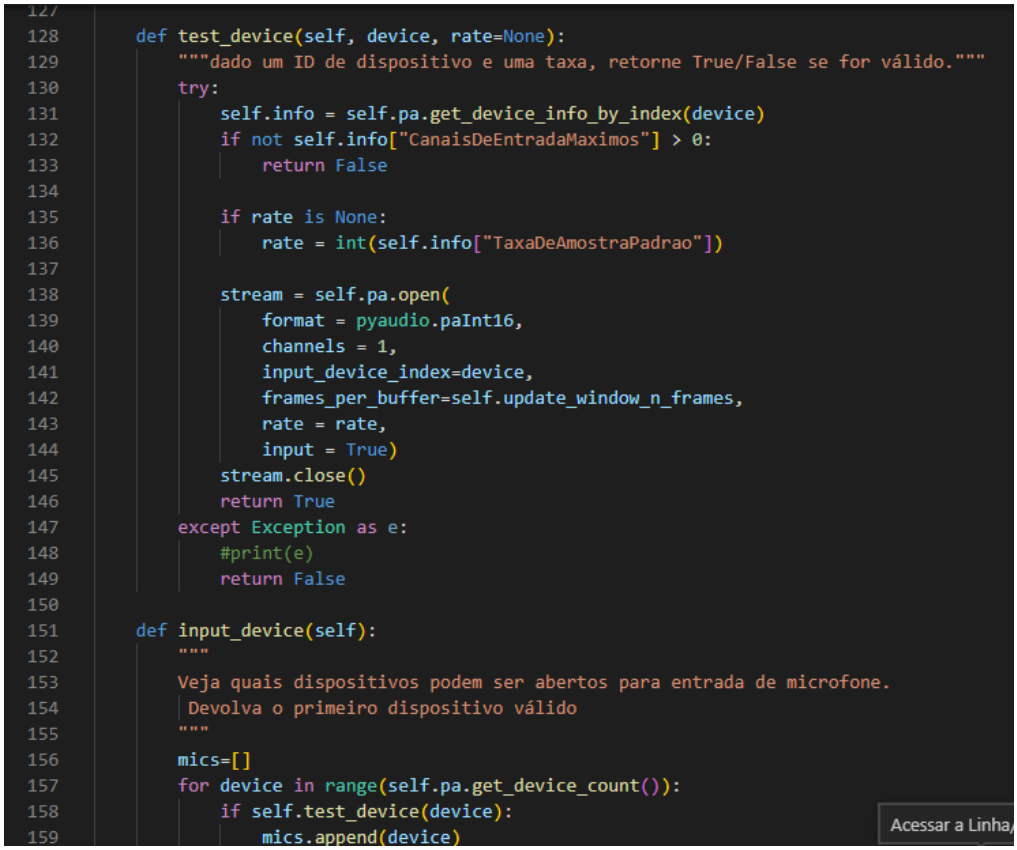
Já na função da linha 63, escolhe o canal de áudio, a taxa de amostragem e o buffer. Nas linhas de 71 a 76 é apresentado alguns dados como o microfone utilizado, a taxa de atualização, a amostragem, etc. Depois é iniciado a contagem do tempo e a leitura do buffer na função seguinte.




```
91
92      def stream_start(self, data_windows_to_buffer = None):
93          self.data_windows_to_buffer = data_windows_to_buffer
94
95          if data_windows_to_buffer is None:
96              self.data_windows_to_buffer = int(self.updates_per_second / 2) #Por padrão, buffer de 0,5 segund
97          else:
98              self.data_windows_to_buffer = data_windows_to_buffer
99
100         self.data_buffer = numpy_data_buffer(self.data_windows_to_buffer, self.update_window_n_frames)
101
102         print("\n--🎙  -- Iniciando transmissão de áudio ao vivo...\n")
103         self.stream.start_stream()
104         self.stream_start_time = time.time()
105
106     def terminate(self):
107         print("👋  Enviando comando de término de stream...")
108         self.stream.stop_stream()
109         self.stream.close()
110         self.pa.terminate()
111
112     def valid_low_rate(self, device, test_rates = [44100, 22050]):
113         """Defina a taxa para a faixa de áudio mais baixa suportada."""
114         for testrate in test_rates:
115             if self.test_device(device, rate=testrate):
116                 return testrate
117
118         #If none of the test_rates worked, try the default rate:
119         self.info = self.pa.get_device_info_by_index(device)
120         default_rate = int(self.info["TaxaDeAmostraPadrao"])
121
122         if self.test_device(device, rate=default_rate):
123             return default_rate
```

Figura 6 - Funções para inicialização, parada e validade da taxa de amostragem.

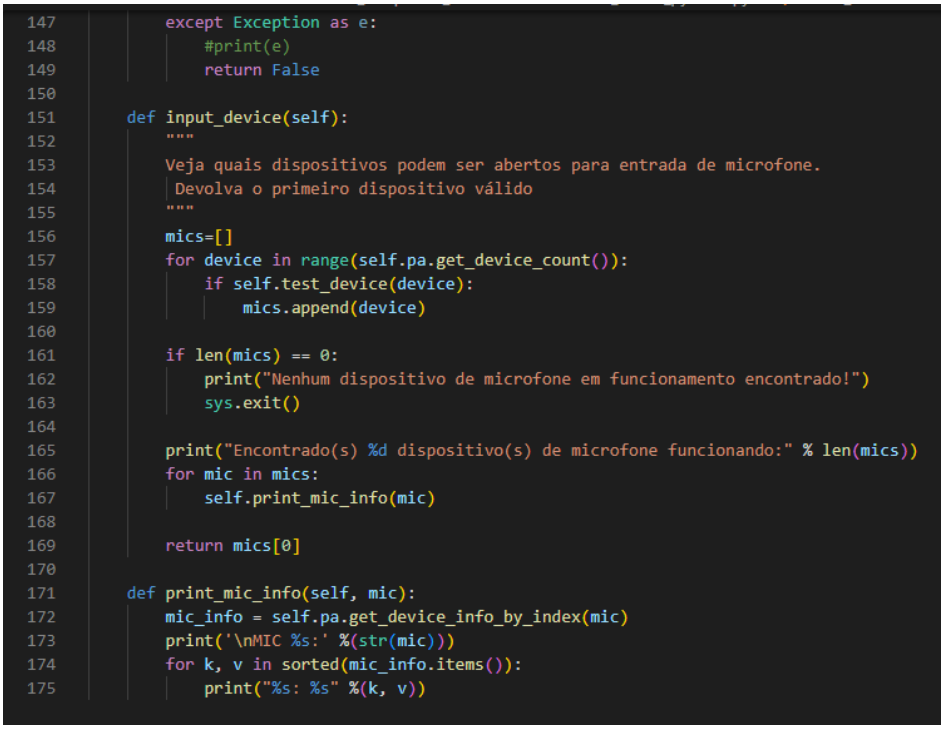
A função **stream_start** define o buffer utilizado e também inicia a gravação. Na linha 106 apenas temos uma função que determina a parada da gravação e na da linha 112, define a taxa para a faixa de áudio mais baixa suportada, além de imprimir a amostragem padrão.



```
127
128     def test_device(self, device, rate=None):
129         """dado um ID de dispositivo e uma taxa, retorne True/False se for válido."""
130         try:
131             self.info = self.pa.get_device_info_by_index(device)
132             if not self.info["CanaisDeEntradaMaximos"] > 0:
133                 return False
134 
135             if rate is None:
136                 rate = int(self.info["TaxaDeAmostraPadrao"])
137 
138             stream = self.pa.open(
139                 format = pyaudio.paInt16,
140                 channels = 1,
141                 input_device_index=device,
142                 frames_per_buffer=self.update_window_n_frames,
143                 rate = rate,
144                 input = True)
145             stream.close()
146             return True
147         except Exception as e:
148             #print(e)
149             return False
150 
151     def input_device(self):
152         """
153         Veja quais dispositivos podem ser abertos para entrada de microfone.
154          Devolva o primeiro dispositivo válido
155         """
156         mics=[]
157         for device in range(self.pa.get_device_count()):
158             if self.test_device(device):
159                 mics.append(device)
```


Figura 7 - Função para teste da entrada.

Como o nome da própria função diz, a função **test_device** testa o microfone selecionado para verificar se o mesmo está funcionando perfeitamente e a **input_device** da linha 151 determina a entrada do microfone, listando todos os disponíveis e selecionando o primeiro.



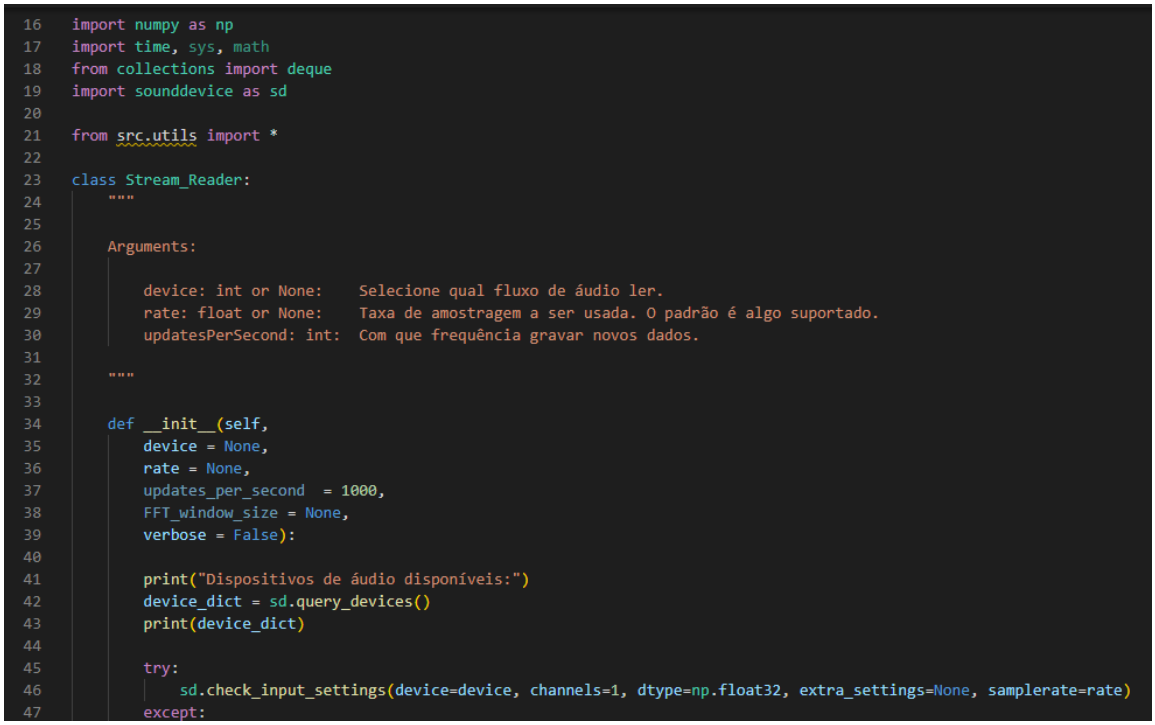
```
147             except Exception as e:
148                 #print(e)
149                 return False
150 
151         def input_device(self):
152             """
153             Veja quais dispositivos podem ser abertos para entrada de microfone.
154              Devolva o primeiro dispositivo válido
155             """
156             mics=[]
157             for device in range(self.pa.get_device_count()):
158                 if self.test_device(device):
159                     mics.append(device)
160 
161             if len(mics) == 0:
162                 print("Nenhum dispositivo de microfone em funcionamento encontrado!")
163                 sys.exit()
164 
165             print("Encontrado(s) %d dispositivo(s) de microfone funcionando:" % len(mics))
166             for mic in mics:
167                 self.print_mic_info(mic)
168 
169             return mics[0]
170 
171         def print_mic_info(self, mic):
172             mic_info = self.pa.get_device_info_by_index(mic)
173             print('\nMIC %s:' %(str(mic)))
174             for k, v in sorted(mic_info.items()):
175                 print("%s: %s" %(k, v))
```

Figura 8 - Função para verificar o microfone e outra para imprimir suas informações.

Na linha 171 é definida uma função que tem como objetivo imprimir as informações do microfone selecionado, como o nome do mesmo e o seu índice.

5.2.3 Stream_reader_sounddevice

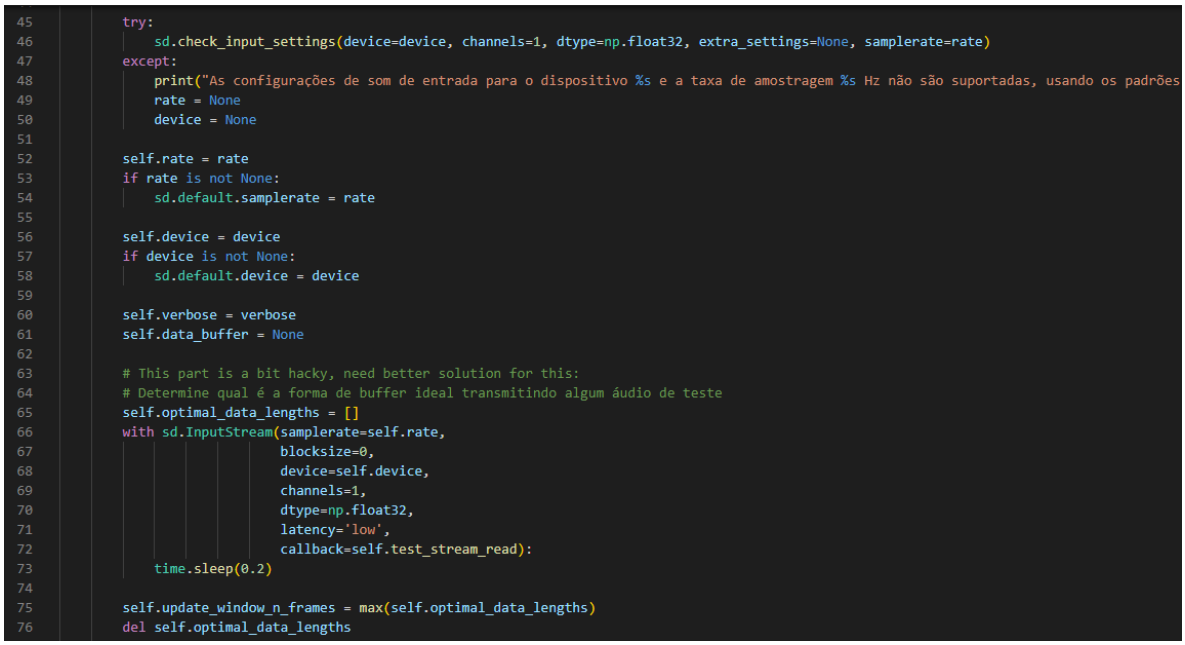
Inicialmente contém os mesmos argumentos da classe anterior, o **device, rate** e **updatesPerSecond**, com os mesmos objetivos, porém há algumas diferenças, pois esta classe determina os dispositivos de som.



```python
import numpy as np
import time, sys, math
from collections import deque
import sounddevice as sd

from src.utils import *

class Stream_Reader:
    """

    Arguments:

        device: int or None:    Selecione qual fluxo de áudio ler.
        rate: float or None:    Taxa de amostragem a ser usada. O padrão é algo suportado.
        updatesPerSecond: int:  Com que frequência gravar novos dados.

    """

    def __init__(self,
        device = None,
        rate = None,
        updates_per_second  = 1000,
        FFT_window_size = None,
        verbose = False):

        print("Dispositivos de áudio disponíveis:")
        device_dict = sd.query_devices()
        print(device_dict)

        try:
            sd.check_input_settings(device=device, channels=1, dtype=np.float32, extra_settings=None, samplerate=rate)
        except:
```

Figura 9 - Função de inicialização da stream_reader_sounddevice.

As bibliotecas das linhas iniciais são utilizadas no decorrer do programa, principalmente a **sounddevice**, pois a mesma é a principal responsável pelo reconhecimento. Algumas variáveis essenciais são declaradas, com os valores padrões e na linha de 41 a 43, apresenta todos os dispositivos listados encontrados no computador de execução.



```python
        try:
            sd.check_input_settings(device=device, channels=1, dtype=np.float32, extra_settings=None, samplerate=rate)
        except:
            print("As configurações de som de entrada para o dispositivo %s e a taxa de amostragem %s Hz não são suportadas, usando os padrões
            rate = None
            device = None

        self.rate = rate
        if rate is not None:
            sd.default.samplerate = rate

        self.device = device
        if device is not None:
            sd.default.device = device

        self.verbose = verbose
        self.data_buffer = None

        # This part is a bit hacky, need better solution for this:
        # Determine qual é a forma de buffer ideal transmitindo algum áudio de teste
        self.optimal_data_lengths = []
        with sd.InputStream(samplerate=self.rate,
                            blocksize=0,
                            device=self.device,
                            channels=1,
                            dtype=np.float32,
                            latency='low',
                            callback=self.test_stream_read):
            time.sleep(0.2)

        self.update_window_n_frames = max(self.optimal_data_lengths)
        del self.optimal_data_lengths
```

Figura 10 - Tratamento de erro e configurações adicionais do microfone.

Na linha 45 é feito um tratamento de erro, onde as configurações do dispositivo são determinadas, que é utilizado somente um canal para facilitar o uso da FFT, o tipo de dados (float de 32 bits), outras configurações padrões e uma taxa de amostragem padrão de 44100. E caso ocorra o erro, imprime na tela do usuário que as configurações do microfone não são suportadas, resultando nas configurações padrões. Caso não se usasse as configurações padrões da taxa de amostragem e de microfone, é feita a atribuição dos respectivos valores nas linhas de 52 a 54 e de 56 a 58. Depois é atribuído os valores do modo **verbose** (modo detalhado, tanto do software, quanto do hardware), que foi atribuído com valor de false e o buffer também é utilizado com **none**, pois posteriormente será inserido seu valor. Na função da linha 66, é criada a entrada do microfone com seus devidos atributos com um tempo de delay de 0,2 segundos. Ao fim é determinado o número de frames que será apresentado na janela.

```
81          self.stream = sd.InputStream(
82                                       samplerate=self.rate,
83                                       blocksize=self.update_window_n_frames,
84                                       device=None,
85                                       channels=1,
86                                       dtype=np.float32,
87                                       latency='low',
88                                       extra_settings=None,
89                                       callback=self.non_blocking_stream_read)
90
91          self.rate = self.stream.samplerate
92          self.device = self.stream.device
93
94          self.updates_per_second = self.rate / self.update_window_n_frames
95          self.info = ''
96          self.data_capture_delays = deque(maxlen=20)
97          self.new_data = False
98          if self.verbose:
99              self.data_capture_delays = deque(maxlen=20)
100             self.num_data_captures = 0
101
102         self.device_latency = device_dict[self.device]['default_low_input_latency']
103
104         print("\n###############################################################################################")
105         print("\nPadrão para usar o primeiro microfone de trabalho, Executando no microfone %s com propriedades:" %str(self.device))
106         print(device_dict[self.device])
107         print('Que tem uma latência de %.2f ms' %(1000*self.device_latency))
108         print("\n###############################################################################################")
109         print('Gravando áudio em %d Hz\nUsando janelas de dados (sem sobreposição) de %d amostras (atualizando em %.2ffps)'
110             %(self.rate, self.update_window_n_frames, self.updates_per_second))
111
112     def non_blocking_stream_read(self, indata, frames, time_info, status):
```

Figura 11 - Inserção dos parâmetros do microfone e seleção.

A primeira função da imagem tem o mesmo objetivo da função explicada anteriormente, mudando somente o último argumento, que é o não bloqueio da entrada (**non_blocking_stream_read**). Após isso, é feito algumas atribuições e cálculos para atualizações, temporização e latência. Tem uma sequência de print's para avisar ao usuário qual o microfone executado com as devidas propriedades e características.

```python
    def non_blocking_stream_read(self, indata, frames, time_info, status):
        if self.verbose:
            start = time.time()
            if status:
                print(status)

        if self.data_buffer is not None:
            self.data_buffer.append_data(indata[:,0])
            self.new_data = True

        if self.verbose:
            self.num_data_captures += 1
            self.data_capture_delays.append(time.time() - start)

        return

    def test_stream_read(self, indata, frames, time_info, status):
        '''
        Função fictícia para determinar qual tamanho de bloco o fluxo está usando
        '''
        self.optimal_data_lengths.append(len(indata[:,0]))
        return
```

Figura 12 - Função de não bloqueio da entrada e teste.

Aqui são definidas algumas funções utilizadas anteriormente no código, como a **non_blocking_stream_read**, que calcula o tempo, se a buffer não está vazio e fazer a atribuição correta. Também contém a função **test_stream_read** que determina qual o tamanho do bloco de fluxo está sendo usado.

```python
    def stream_start(self, data_windows_to_buffer = None):
        self.data_windows_to_buffer = data_windows_to_buffer

        if data_windows_to_buffer is None:
            self.data_windows_to_buffer = int(self.updates_per_second / 2) #Por padrão, buffer 0.5 segundos de audio
        else:
            self.data_windows_to_buffer = data_windows_to_buffer

        self.data_buffer = numpy_data_buffer(self.data_windows_to_buffer, self.update_window_n_frames)

        print("\n--🎙  -- Iniciando transmissão de áudio ao vivo...\n")
        self.stream.start()
        self.stream_start_time = time.time()

    def terminate(self):
        print("👋  Enviando comando de término de stream...")
        self.stream.stop()
```

Figura 13 - Função de inicialização da gravação.

Na primeira função, o buffer recebe os dados com um tempo de 0,5 segundos por padrão e assim é iniciada a transmissão de áudio. E por fim a última função simplesmente fecha o fluxo de áudio.

5.2.4 Utils

Esta classe utiliza algumas funções essenciais para o funcionamento do sistema em si, como cálculos matemáticos ou inserção de botões na tela de visualização.

```
16
17  import numpy as np
18  import math, scipy, pygame
19
20  def round_up_to_even(f):
21      return int(math.ceil(f / 2.) * 2)
22
23  def round_to_nearest_power_of_two(f, base=2):
24      l = math.log(f,base)
25      rounded = int(np.round(l,0))
26      return base**rounded
27
28  def get_frequency_bins(start, stop, n):
29      octaves = np.logspace(log(start)/log(2), log(stop)/log(2), n, endpoint=True, base=2, dtype=None)
30      return np.insert(octaves, 0, 0)
31
32  def gaussian_kernel1d(sigma, truncate=2.0):
33      sigma = float(sigma)
34      sigma2 = sigma * sigma
35      #tornar o raio do filtro igual a truncar os desvios padrão
36      radius = int(truncate * sigma + 0.5)
37      exponent_range = np.arange(1)
38
39      x = np.arange(-radius, radius+1)
40      phi_x = np.exp(-0.5 / sigma2 * x ** 2)
41      phi_x = phi_x / phi_x.sum()
42      return phi_x
```

Figura 14 - Funções para frequência e filtro.

Para realizar todas as operações, é preciso de alguns módulos especiais como o **pygame** e o **math** próprio do python. A primeira função denominada **round_up_to_event** somente calcula o maior inteiro maior ou igual, enquanto que a seguinte faz o cálculo do log na base 2. Uma importante função é a **get_frequency_bins** que retorna o cálculo do log para a frequência, porém inseridos em um vetor. Já a função da linha 32, recebe alguns argumentos para aplicar um filtro tornando igual aos desvios padrão.

```
43
44  def gaussian_kernel_1D(w, sigma):
45      sigma = sigma
46      x = np.linspace(-sigma, sigma, w+1)
47      kern1d = np.diff(scipy.stats.norm.cdf(x))
48      return kern1d/kern1d.sum()
49
50  def get_smoothing_filter(FFT_window_size_ms, filter_length_ms, verbose = 0):
51      buffer_length = round_up_to_even(filter_length_ms / FFT_window_size_ms)+1
52      filter_sigma = buffer_length / 3  #A rapidez com que a influência de suavização cai sobre o comprimento do buffer
53      filter_weights = gaussian_kernel1d(filter_sigma)[:,np.newaxis]
54
55      max_index = np.argmax(filter_weights)
56      filter_weights = filter_weights[:max_index+1]
57      filter_weights = filter_weights / np.mean(filter_weights)
58
59      if verbose:
60          min_fraction = 100*np.min(filter_weights)/np.max(filter_weights)
61          print('\nAplicando suavização temporal aos recursos FFT...')
62          print("O buffer de suavização contém %d janela FFT (sigma: %.3f) --> min_contribuição: %.3f%%" %(buffer_length, filt
63          print("Filter weights:")
64          for i, w in enumerate(filter_weights):
65              print("%02d: %.3f" %(len(filter_weights)-i, w))
66
67      return filter_weights
68
```

Figura 15 - Funções para cálculo do filtro e suavização.

Como também é preciso aplicar uma suavização, a função **get_smoothing_filter** faz essa aplicação no buffer da dados recebidos e ainda a apresenta na tela do usuário.

```
class numpy_data_buffer:
    """
    Um buffer FIFO rápido e circular em numpy com interações mínimas de memória usando uma matriz de ponteiros de índice
    """

    def __init__(self, n_windows, samples_per_window, dtype = np.float32, start_value = 0, data_dimensions = 1):
        self.n_windows = n_windows
        self.data_dimensions = data_dimensions
        self.samples_per_window = samples_per_window
        self.data = start_value * np.ones((self.n_windows, self.samples_per_window), dtype = dtype)

        if self.data_dimensions == 1:
            self.total_samples = self.n_windows * self.samples_per_window
        else:
            self.total_samples = self.n_windows

        self.elements_in_buffer = 0
        self.overwrite_index = 0

        self.indices = np.arange(self.n_windows, dtype=np.int32)
        self.last_window_id = np.max(self.indices)
        self.index_order = np.argsort(self.indices)

```

Figura 16 - Definição do buffer circular.

Nesta classe é definido um buffer circular utilizando o módulo **numpy** para uma melhor rapidez de cálculos. Para tal, é requirido as dimensões dos dados obtidos, a taxa de amostragem, o valor de start e no fim são gerados um array de índices, id e index, além do próprio buffer.

```
    def append_data(self, data_window):
        self.data[self.overwrite_index, :] = data_window

        self.last_window_id += 1
        self.indices[self.overwrite_index] = self.last_window_id
        self.index_order = np.argsort(self.indices)

        self.overwrite_index += 1
        self.overwrite_index = self.overwrite_index % self.n_windows

        self.elements_in_buffer += 1
        self.elements_in_buffer = min(self.n_windows, self.elements_in_buffer)

    def get_most_recent(self, window_size):
        ordered_dataframe = self.data[self.index_order]
        if self.data_dimensions == 1:
            ordered_dataframe = np.hstack(ordered_dataframe)
        return ordered_dataframe[self.total_samples - window_size:]

    def get_buffer_data(self):
        return self.data[:self.elements_in_buffer]

```

Figura 17 - Recebimento dos dados, retorno dos dados e retorno do buffer.

Na função **append_data**, somente é recebido os dados e inseridos no buffer de acordo com o índice respectivo e o id do mesmo. Na linha 105, a função somente devolve os dados mais recentes que para o uso dos frames na tela. A **get_buffer_data** como o próprio nome diz retorna os valores contidos no buffer.

```
114 class Button:
115     def __init__(self, text="", right=10, top=30, width=None, height=20):
116         self.text = text
117         self.top = top
118         self.height = height
119         self.colour1 = (220, 220, 220)  # main
120         self.colour2 = (100, 100, 100)  # border
121         self.colour3 = (172, 220, 247)  # hover
122         self.colour4 = (225, 243, 252)
123         self.fontname = "freesansbold.ttf"
124         self.fontsize = self.height-6
125         self.mouse_over = False
126         self.mouse_down = False
127         self.mouse = "off"
128         self.clicked = False
129         self.pyg = pygame
130         self.font = pygame.font.SysFont(self.fontname, self.fontsize)
131         self.text_width, self.text_height = self.pyg.font.Font.size(self.font, self.text)
132         if width == None:
133             self.width = int(self.text_width * 1.3)
134             self.width_type = "texto"
135         else:
136             self.width = width
137             self.width_type = "usuario"
138 
139         self.left = right - self.width
140         self.buttonUP = self.pyg.Surface((self.width, self.height))
141         self.buttonDOWN = self.pyg.Surface((self.width, self.height))
142         self.buttonHOVER = self.pyg.Surface((self.width, self.height))
143         self.__update__()
144 
145     def __update__(self):
```

Figura 18 - Inicialização da classe Button.

A classe botão é responsável por inserir os mesmos na janela de visualização do usuário e para interagir. Portanto, é determinado sua posição, sua cor, fonte e tipo de click que o mouse deve ter.

```
145     def __update__(self):
146         # up
147         r, g, b = self.colour1
148         self.buttonUP.fill(self.colour1)
149         self.pyg.draw.rect(self.buttonUP, (r+20, g+20, b+20), (0, 0, self.width, self.height/2), 0)
150         self.pyg.draw.line(self.buttonUP, self.colour2, (2, 0), (self.width-3, 0), 1)
151         self.pyg.draw.line(self.buttonUP, self.colour2, (2, self.height-1), (self.width-3, self.height-1), 1)
152         self.pyg.draw.line(self.buttonUP, self.colour2, (0, 2), (0, self.height-3), 1)
153         self.pyg.draw.line(self.buttonUP, self.colour2, (self.width-1, 2), (self.width-1, self.height-3), 1)
154         self.buttonUP.set_at((1, 1), self.colour2)
155         self.buttonUP.set_at((self.width-2, 1), self.colour2)
156         self.buttonUP.set_at((1, self.height-2), self.colour2)
157         self.buttonUP.set_at((self.width-2, self.height-2), self.colour2)
158         self.buttonUP.blit(self.font.render(self.text, False, (0, 0, 0)), ((self.width/2)-(self.text_width/2), (self.height/2)-(self.text_h
159         # hover
160         self.buttonHOVER.fill(self.colour3)
161         self.pyg.draw.rect(self.buttonHOVER, self.colour4, (0, 0, self.width, self.height/2), 0)
162         self.pyg.draw.line(self.buttonHOVER, self.colour2, (2, 0), (self.width-3, 0), 1)
163         self.pyg.draw.line(self.buttonHOVER, self.colour2, (2, self.height-1), (self.width-3, self.height-1), 1)
164         self.pyg.draw.line(self.buttonHOVER, self.colour4, (2, self.height-2), (self.width-3, self.height-2), 1)
165         self.pyg.draw.line(self.buttonHOVER, self.colour2, (0, 2), (0, self.height-3), 1)
166         self.pyg.draw.line(self.buttonHOVER, self.colour4, (1, 2), (1, self.height-3), 2)
167         self.pyg.draw.line(self.buttonHOVER, self.colour2, (self.width-1, 2), (self.width-1, self.height-3), 1)
168         self.buttonHOVER.set_at((1, 1), self.colour2)
169         self.buttonHOVER.set_at((self.width-2, 1), self.colour2)
170         self.buttonHOVER.set_at((1, self.height-2), self.colour2)
171         self.buttonHOVER.set_at((self.width-2, self.height-2), self.colour2)
172         self.buttonHOVER.blit(self.font.render(self.text, False, (0, 0, 0)), ((self.width/2)-(self.text_width/2), (self.height/2)-(self.tex
173         # down
174         r, g, b = self.colour3
175         r2, g2, b2 = self.colour4
176         self.buttonDOWN.fill((r-20, g-20, b-10))
```

Figura 19 - Função de atualização dos botões.

Aqui são definidas as informações necessárias para a posição dos botões de acordo com a atualização dos frames da janela de visualização.

```
190     def draw(self, surface):
191         self.__mouse_check__()
192         if self.mouse == "hover":
193             surface.blit(self.buttonHOVER, (self.left, self.top))
194         elif self.mouse == "off":
195             surface.blit(self.buttonUP, (self.left, self.top))
196         elif self.mouse == "down":
197             surface.blit(self.buttonDOWN, (self.left, self.top))
198 
199     def __mouse_check__(self):
200         _1, _2, _3 = pygame.mouse.get_pressed()
201         mouse_x, mouse_y = pygame.mouse.get_pos()
202         if not _1:
203             self.mouse = "off"
204         if mouse_x > self.left and mouse_x < self.left + self.width and mouse_y > self.top ar
205             self.mouse = "hover"
206         if not self.mouse_down and _1 and self.mouse == "hover":
207             self.mouse = "down"
208             self.clicked = True
209         if self.mouse == "off":
210             self.clicked = False
211 
212     def click(self):
213         _1, _2, _3 = pygame.mouse.get_pressed()
214         mouse_x, mouse_y = pygame.mouse.get_pos()
215         if mouse_x > self.left and mouse_x < self.left + self.width and mouse_y > self.top ar
216             self.clicked = False
217             return True
218         else:
219             return False
220 
```

Figura 20 - Para desenho dos botões, verificação do click e determinação do click.

A primeira função apenas define a posição do mouse ao houver click. Enquanto que a segunda verifica a ação do click do mouse e a última determina se houve o click de fato.

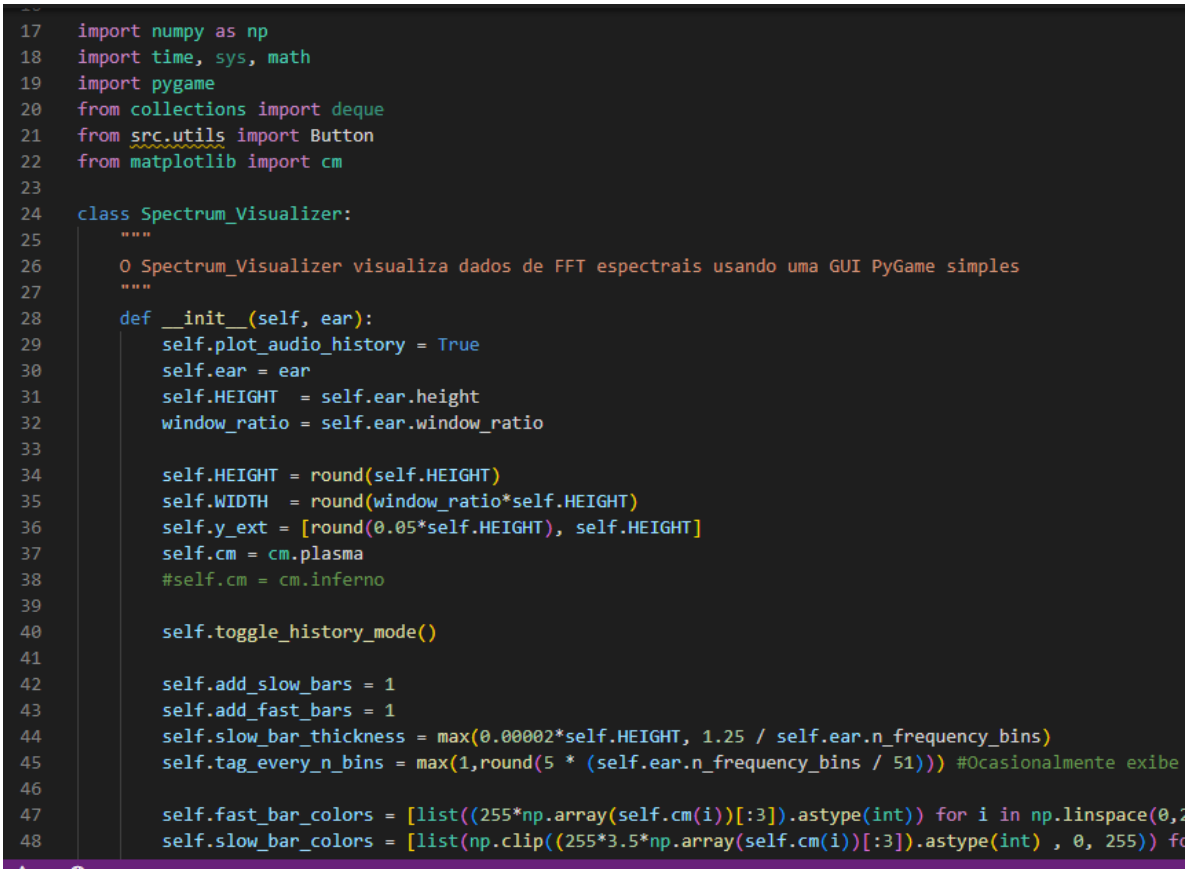
```
220 
221     def set_text(self, text, fontname="Arial", fontsize=None):
222         self.text = text
223         self.fontname = fontname
224         if not fontsize == None:
225             self.fontsize = fontsize
226         self.font = pygame.font.SysFont(self.fontname, self.fontsize)
227         self.text_width, self.text_height = self.pyg.font.Font.size(self.font, self.text)
228         if self.width_type == "text":
229             self.width = self.text_width + 20
230         self.buttonUP = self.pyg.Surface((self.width, self.height))
231         self.buttonDOWN = self.pyg.Surface((self.width, self.height))
232         self.buttonHOVER = self.pyg.Surface((self.width, self.height))
233         self.__update__()
```

Figura 21 - Inserção de texto nos botões.

Finalmente, a última função envia o texto, com as propriedades necessárias quando houver uma atualização da tela de visualização. Observe que as tarefas das funções foram bastante divididas para tornar o código mais limpo.

5.2.5 Visualizer

Como é auto informativo, esta classe tem como objetivo criar uma janela que apresente a FFT em tempo real para o usuário e para cumprir essa tarefa, utilizou-se de algumas funções já implementadas do **pygame**.



```
import numpy as np
import time, sys, math
import pygame
from collections import deque
from src.utils import Button
from matplotlib import cm

class Spectrum_Visualizer:
    """
    O Spectrum_Visualizer visualiza dados de FFT espectrais usando uma GUI PyGame simples
    """
    def __init__(self, ear):
        self.plot_audio_history = True
        self.ear = ear
        self.HEIGHT  = self.ear.height
        window_ratio = self.ear.window_ratio

        self.HEIGHT = round(self.HEIGHT)
        self.WIDTH  = round(window_ratio*self.HEIGHT)
        self.y_ext = [round(0.05*self.HEIGHT), self.HEIGHT]
        self.cm = cm.plasma
        #self.cm = cm.inferno

        self.toggle_history_mode()

        self.add_slow_bars = 1
        self.add_fast_bars = 1
        self.slow_bar_thickness = max(0.00002*self.HEIGHT, 1.25 / self.ear.n_frequency_bins)
        self.tag_every_n_bins = max(1,round(5 * (self.ear.n_frequency_bins / 51))) #Ocasionalmente exibe

        self.fast_bar_colors = [list((255*np.array(self.cm(i))[:3]).astype(int)) for i in np.linspace(0,
        self.slow_bar_colors = [list(np.clip((255*3.5*np.array(self.cm(i))[:3]).astype(int) , 0, 255)) fo
```

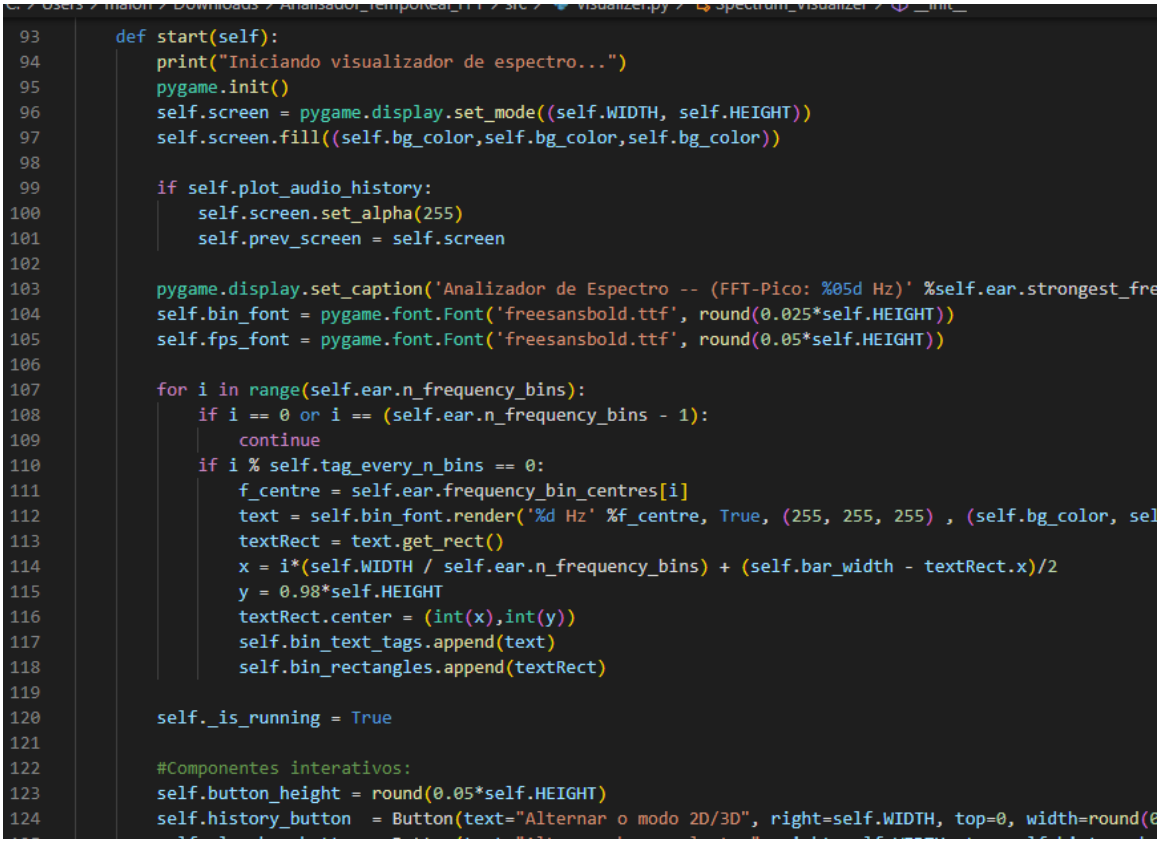
Figura 22 - Inicialização da classe Spectrum_Visualizer.

Há algumas bibliotecas para cálculos e plotagem ainda que serão utilizadas. Primeiramente é definida as variáveis iniciais que serão utilizadas no decorrer do código, como a altura, dimensões, o fundo, etc. Na assinatura da função também é passado os dados coletados (**ear**) para plotar a visualização e observe também que há uma ativação do histórico para a apresentação. Esta variável **ear** tem os atributos já comentados anteriormente, como a frequência, a taxa de amostragem, atualizações por segundos, entre outros. Também são configurados os valores do eixo das abcissas, ou seja, as frequências.

```
63
64      def toggle_history_mode(self):
65
66          if self.plot_audio_history:
67              self.bg_color           = 10    #Cor de fundo
68              self.decay_speed        = 0.10  #Decaimento vertical de barras lentas
69              self.inter_bar_distance = 0
70              self.avg_energy_height  = 0.1125
71              self.alpha_multiplier   = 0.995
72              self.move_fraction      = 0.0099
73              self.shrink_f           = 0.994
74
75          else:
76              self.bg_color           = 10
77              self.decay_speed        = 0.06
78              self.inter_bar_distance = int(0.2*self.WIDTH / self.ear.n_frequency_bins)
79              self.avg_energy_height  = 0.225
80
81          self.bar_width = (self.WIDTH / self.ear.n_frequency_bins) - self.inter_bar_distance
82
83          #Configura as barras:
84          self.slow_bars, self.fast_bars, self.bar_x_positions = [],[],[]
85          for i in range(self.ear.n_frequency_bins):
86              x = int(i* self.WIDTH / self.ear.n_frequency_bins)
87              fast_bar = [int(x), int(self.y_ext[0]), math.ceil(self.bar_width), None]
88              slow_bar = [int(x), None, math.ceil(self.bar_width), None]
89              self.bar_x_positions.append(x)
90              self.fast_bars.append(fast_bar)
91              self.slow_bars.append(slow_bar)
92
```

Figura 23 - Função de histórico da FFT.

Já nesta função, a cor de fundo é personalizada – quanto menor o valor, mais preta a tela. O valor de decaimento das barras lentas também é configurado, tanto no modo 3D, quanto no 2D. Na linha 85 é feito um laço de repetição que determina as barras que representam a amplitude em relação a frequência, portanto também é determinado o tempo de acionamento dessas barras.



```
93      def start(self):
94          print("Iniciando visualizador de espectro...")
95          pygame.init()
96          self.screen = pygame.display.set_mode((self.WIDTH, self.HEIGHT))
97          self.screen.fill((self.bg_color,self.bg_color,self.bg_color))
98
99          if self.plot_audio_history:
100             self.screen.set_alpha(255)
101             self.prev_screen = self.screen
102
103         pygame.display.set_caption('Analizador de Espectro -- (FFT-Pico: %05d Hz)' %self.ear.strongest_fre
104         self.bin_font = pygame.font.Font('freesansbold.ttf', round(0.025*self.HEIGHT))
105         self.fps_font = pygame.font.Font('freesansbold.ttf', round(0.05*self.HEIGHT))
106
107         for i in range(self.ear.n_frequency_bins):
108             if i == 0 or i == (self.ear.n_frequency_bins - 1):
109                 continue
110             if i % self.tag_every_n_bins == 0:
111                 f_centre = self.ear.frequency_bin_centres[i]
112                 text = self.bin_font.render('%d Hz' %f_centre, True, (255, 255, 255) , (self.bg_color, sel
113                 textRect = text.get_rect()
114                 x = i*(self.WIDTH / self.ear.n_frequency_bins) + (self.bar_width - textRect.x)/2
115                 y = 0.98*self.HEIGHT
116                 textRect.center = (int(x),int(y))
117                 self.bin_text_tags.append(text)
118                 self.bin_rectangles.append(textRect)
119
120         self._is_running = True
121
122         #Componentes interativos:
123         self.button_height = round(0.05*self.HEIGHT)
124         self.history_button  = Button(text="Alternar o modo 2D/3D", right=self.WIDTH, top=0, width=round(0
```

Figura 24 - Para inicializar a janela de visualização.

A função **start** é dedicada para iniciar a janela de visualização, com o nome da janela, o pico de frequência, a fonte utilizada, a cor de fundo do eixo das abcissas, as dimensões e proporções adequadas. Ademais, contém os botões que estão na janela, o de alternar os modos de visualização e de ligar/desligar as barras lentas.

```
127     def stop(self):
128         print("Parando o visualizador de espectro...")
129         del self.fps_font
130         del self.bin_font
131         del self.screen
132         del self.prev_screen
133         pygame.quit()
134         self._is_running = False
135 
136     def toggle_display(self):
137 
138         #Esta função pode ser acionada para ligar/desligar o display
139 
140         if self._is_running: self.stop()
141         else: self.start()
142 
143     def update(self):
144         for event in pygame.event.get():
145             if self.history_button.click():
146                 self.plot_audio_history = not self.plot_audio_history
147                 self.toggle_history_mode()
148             if self.slow_bar_button.click():
149                 self.add_slow_bars = not self.add_slow_bars
150                 self.slow_features = [0]*self.ear.n_frequency_bins
151 
152         if np.min(self.ear.bin_mean_values) > 0:
153             self.frequency_bin_energies = self.avg_energy_height * self.ear.frequency_bin_energies / self
154 
155         if self.plot_audio_history:
156             new_w, new_h = int((2+self.shrink_f)/3*self.WIDTH), int(self.shrink_f*self.HEIGHT)
157             #new_w, new_h = int(self.shrink_f*self.WIDTH), int(self.shrink_f*self.HEIGHT)
```

Figura 25 - Funções para terminar a janela, acionar o display e atualização.

A função da linha 127, para o visualizador de espectro, desligando todos as propriedades, enquanto que a função mais baixo serve para ligar/desligar o display e a última faz a atualização da janela com os dados novos obtidos, o histórico para a representação, o tempo que também é necessário e as cores utilizadas também

```
200
201     def plot_bars(self):
202         bars, slow_bars, new_slow_features = [], [], []
203         local_height = self.y_ext[1] - self.y_ext[0]
204         feature_values = self.frequency_bin_energies[::-1]
205
206         for i in range(len(self.frequency_bin_energies)):
207             feature_value = feature_values[i] * local_height
208
209             self.fast_bars[i][3] = int(feature_value)
210
211             if self.plot_audio_history:
212                 self.fast_bars[i][3] = int(feature_value + 0.02*self.HEIGHT)
213
214             if self.add_slow_bars:
215                 self.decay = min(0.99, 1 - max(0,self.decay_speed * 60 / self.ear.fft_fps))
216                 slow_feature_value = max(self.slow_features[i]*self.decay, feature_value)
217                 new_slow_features.append(slow_feature_value)
218                 self.slow_bars[i][1] = int(self.fast_bars[i][1] + slow_feature_value)
219                 self.slow_bars[i][3] = int(self.slow_bar_thickness * local_height)
220
221         if self.add_fast_bars:
222             for i, fast_bar in enumerate(self.fast_bars):
223                 pygame.draw.rect(self.screen,self.fast_bar_colors[i],fast_bar,0)
224
225         if self.plot_audio_history:
226                 self.prev_screen = self.screen.copy().convert_alpha()
227                 self.prev_screen = pygame.transform.rotate(self.prev_screen, 180)
228                 self.prev_screen.set_alpha(self.prev_screen.get_alpha()*self.alpha_multiplier)
229
230         if self.add_slow_bars:
231             for i, slow_bar in enumerate(self.slow_bars):
```

Figura 26 - Controle das barras.

Já esta última função tem como objetivo configurar as barras propriamente ditas, com o histórico do áudio dado, a rapidez das barras, rotação, etc.

5.2.6 Stream_analyzer

Chegamos a classe de integração das classes anteriores, que ainda é responsável pela as atualizações e cálculos de algumas variáveis.

```
17  import numpy as np
18  import time, math, scipy
19  from collections import deque
20  from scipy.signal import savgol_filter
21
22  from src.fft import getFFT
23  from src.utils import *
24
25  class Stream_Analyzer:
26      """
27      Argumentos:
28
29          device: int or None:      Selecione qual dispositivo de áudio ler.
30          rate: float or None:      Taxa de amostragem a ser usada. O padrão é algo suportado.
31          FFT_window_size_ms: int:  Tamanho da janela de tempo (em ms) a ser usado para a transformação FFT
32          updatesPerSecond: int:    Com que frequência gravar novos dados.
33
34      """
35
36      def __init__(self,
37          device = None,
38          rate   = None,
39          tamanhoJanela_ms_FFT  = 50,
40          atualizacaoPorSegundo  = 100,
41          tamanhoSuavizacao_ms = 50,
42          n_compartimentoFrequencia    = 51,
43          visualize = True,
44          verbose   = False,
45          altura    = 450,
46          proporcaoJanela = 24/9):
47
48          self.n_frequency_bins = n_compartimentoFrequencia
```

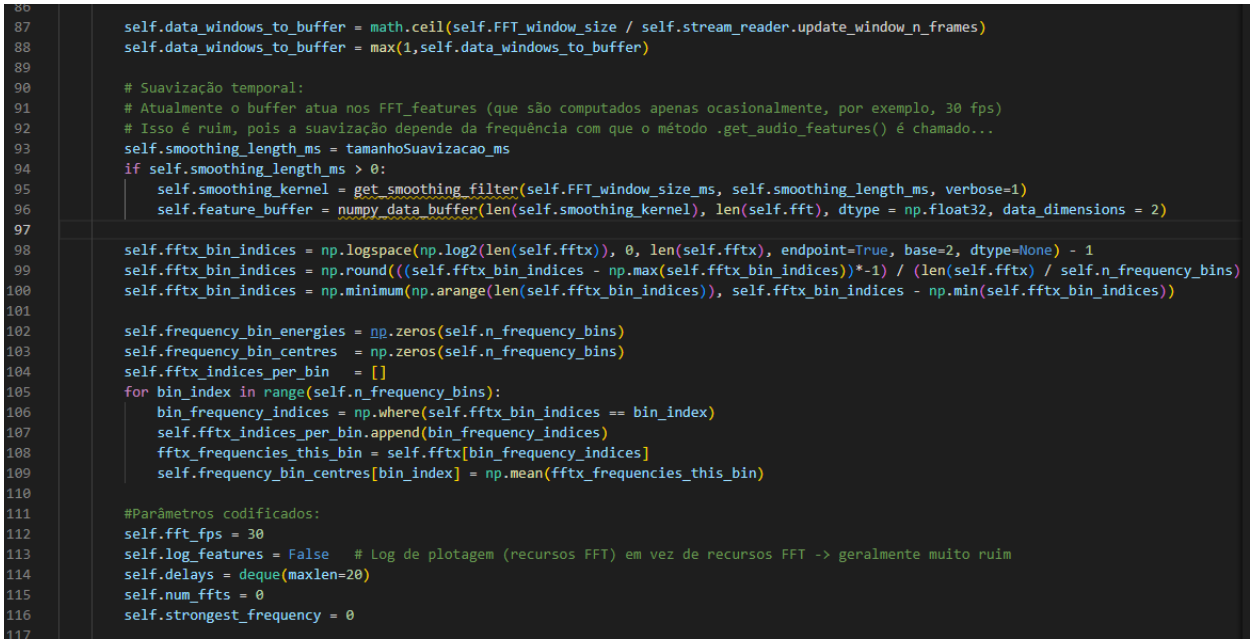
Figura 27 - Inicialização da classe Stream_Analyzer.

Alguns módulos importantes utilizados, como o **numpy** para o cálculo das variáveis, o tempo também para apresentar na tela e um filtro **savgol**, que é usado para suavizar os dados, ou seja, aumentar a precisão dos dados sem distorcer a tendência do sinal. Ademais, é importada a função de cálculo de FFT, a **getFFT**, já explicada anteriormente e todas as funções da classe **utils**. As variáveis principais são **device**, para o dispositivo de gravação; **rate**, para a taxa de amostragem; **FFT_window_size_ms**, que é o tamanho da janela de tempo em microssegundos usado na FFT e finalmente a **updatePerSecond**, responsável por ajustar a frequência de gravação dos novos dados. Importante notar que os parâmetros das linhas de 37 a 46 são inicializados na classe principal **executar_Analizador_FFT**, os outros valores já são dados.

```
55          try:
56              from src.stream_reader_pyaudio import Stream_Reader
57              self.stream_reader = Stream_Reader(
58                  device  = device,
59                  rate    = rate,
60                  updates_per_second  = atualizacaoPorSegundo,
61                  verbose = verbose)
62          except:
63              from src.stream_reader_sounddevice import Stream_Reader
64              self.stream_reader = Stream_Reader(
65                  device  = device,
66                  rate    = rate,
67                  updates_per_second  = atualizacaoPorSegundo,
68                  verbose = verbose)
69
70          self.rate = self.stream_reader.rate
71
72          #Configurações personalizadas:
73          self.rolling_stats_window_s    = 20     # A faixa de eixos dos recursos FFT se adaptará dinamicamen
74          self.equalizer_strength        = 0.20   # [0-1] --> gradualmente redimensiona todos os recursos FFT
75          self.apply_frequency_smoothing = True   # Aplique um filtro de suavização de pós-processamento nas
76
77          if self.apply_frequency_smoothing:
78              self.filter_width = round_up_to_even(0.03*self.n_frequency_bins) - 1
79          if self.visualize:
80              from src.visualizer import Spectrum_Visualizer
81
82          self.FFT_window_size = round_up_to_even(self.rate * tamanhoJanela_ms_FFT / 1000)
83          self.FFT_window_size_ms = 1000 * self.FFT_window_size / self.rate
84          self.fft  = np.ones(int(self.FFT_window_size/2), dtype=float)
85          self.fftx = np.arange(int(self.FFT_window_size/2), dtype=float) * self.rate / self.FFT_window_size
86
```

Figura 28 - Seleção da importação das classes e configurações personalizadas.

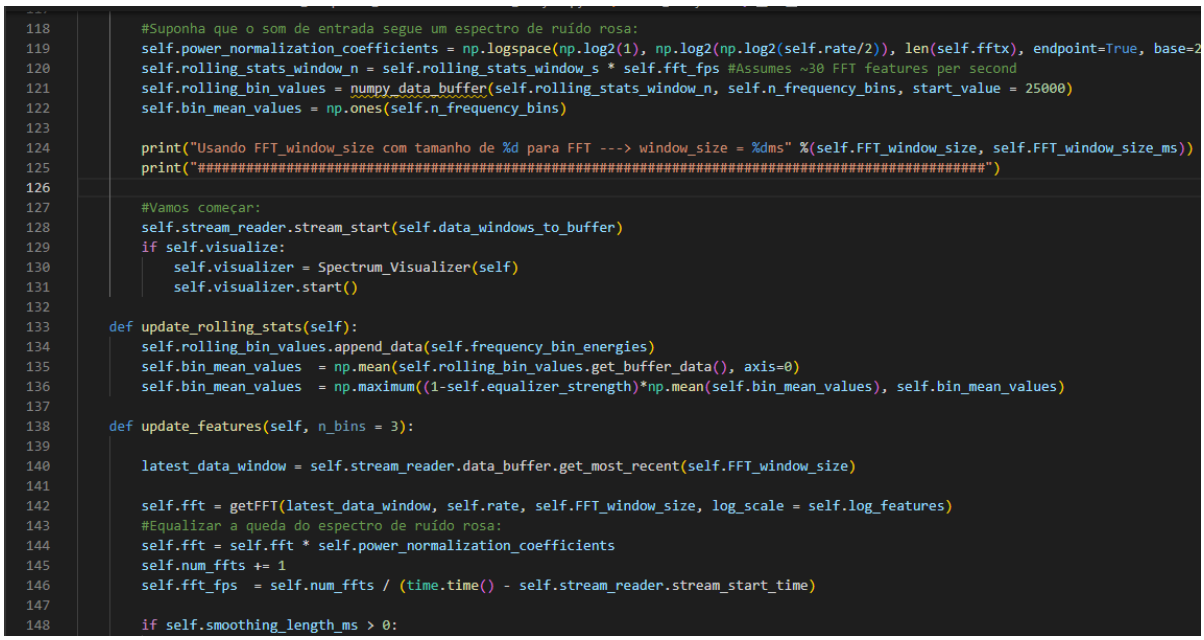
Temos um tratamento de erro, onde faz a importação da classe **stream_reader_pyaudio** caso o **pyaudio** consiga detectar a entrada de som no computador e a importação da classe **stream_reader_sounddevice** caso ocorra algum erro e assim é recorrida a função de leitura com outro módulo. Há também as configurações personalizadas que dita que a faixa de eixos dos recursos de FFT se adaptará automaticamente, um equalizador e um filtro de suavização de processamento dos dados capturados no buffer. Na linha 79, se é selecionado o **visualizer**, é chamada a função **Spectrum_Visualizer** da classe **visualizer** para apresentar na tela do usuário os dados em tempo real. Logo abaixo, é calculado o tamanho do janelamento na FFT.



```
87      self.data_windows_to_buffer = math.ceil(self.FFT_window_size / self.stream_reader.update_window_n_frames)
88      self.data_windows_to_buffer = max(1,self.data_windows_to_buffer)
89
90      # Suavização temporal:
91      # Atualmente o buffer atua nos FFT_features (que são computados apenas ocasionalmente, por exemplo, 30 fps)
92      # Isso é ruim, pois a suavização depende da frequência com que o método .get_audio_features() é chamado...
93      self.smoothing_length_ms = tamanhoSuavizacao_ms
94      if self.smoothing_length_ms > 0:
95          self.smoothing_kernel = get_smoothing_filter(self.FFT_window_size_ms, self.smoothing_length_ms, verbose=1)
96          self.feature_buffer = numpy_data_buffer(len(self.smoothing_kernel), len(self.fft), dtype = np.float32, data_dimensions = 2)
97
98      self.fftx_bin_indices = np.logspace(np.log2(len(self.fftx)), 0, len(self.fftx), endpoint=True, base=2, dtype=None) - 1
99      self.fftx_bin_indices = np.round(((self.fftx_bin_indices - np.max(self.fftx_bin_indices))*-1) / (len(self.fftx) / self.n_frequency_bins)
100     self.fftx_bin_indices = np.minimum(np.arange(len(self.fftx_bin_indices)), self.fftx_bin_indices - np.min(self.fftx_bin_indices))
101
102     self.frequency_bin_energies = np.zeros(self.n_frequency_bins)
103     self.frequency_bin_centres  = np.zeros(self.n_frequency_bins)
104     self.fftx_indices_per_bin   = []
105     for bin_index in range(self.n_frequency_bins):
106         bin_frequency_indices = np.where(self.fftx_bin_indices == bin_index)
107         self.fftx_indices_per_bin.append(bin_frequency_indices)
108         fftx_frequencies_this_bin = self.fftx[bin_frequency_indices]
109         self.frequency_bin_centres[bin_index] = np.mean(fftx_frequencies_this_bin)
110
111     #Parâmetros codificados:
112     self.fft_fps = 30
113     self.log_features = False   # Log de plotagem (recursos FFT) em vez de recursos FFT -> geralmente muito ruim
114     self.delays = deque(maxlen=20)
115     self.num_ffts = 0
116     self.strongest_frequency = 0
```

Figura 29 - Configurações e aplicação da suavização.

Agora é preciso configurar o buffer dos dados e em seguida, aplicar a suavização de acordo com seu tamanho definido, portanto é preciso realizar um vetor que contenha a suavização de todos os pontos e os índices requeridos, o que é feito na linha 105. Já nas linhas de 112 a 116 é inserido dos valores do fps, o log, temporização e as frequências mais fortes.



```
118     #Suponha que o som de entrada segue um espectro de ruído rosa:
119     self.power_normalization_coefficients = np.logspace(np.log2(1), np.log2(np.log2(self.rate/2)), len(self.fftx), endpoint=True, base=2
120     self.rolling_stats_window_n = self.rolling_stats_window_s * self.fft_fps #Assumes ~30 FFT features per second
121     self.rolling_bin_values = numpy_data_buffer(self.rolling_stats_window_n, self.n_frequency_bins, start_value = 25000)
122     self.bin_mean_values = np.ones(self.n_frequency_bins)
123
124     print("Usando FFT_window_size com tamanho de %d para FFT ---> window_size = %dms" %(self.FFT_window_size, self.FFT_window_size_ms))
125     print("#########################################################################################################")
126
127     #Vamos começar:
128     self.stream_reader.stream_start(self.data_windows_to_buffer)
129     if self.visualize:
130         self.visualizer = Spectrum_Visualizer(self)
131         self.visualizer.start()
132
133 def update_rolling_stats(self):
134     self.rolling_bin_values.append_data(self.frequency_bin_energies)
135     self.bin_mean_values  = np.mean(self.rolling_bin_values.get_buffer_data(), axis=0)
136     self.bin_mean_values  = np.maximum((1-self.equalizer_strength)*np.mean(self.bin_mean_values), self.bin_mean_values)
137
138 def update_features(self, n_bins = 3):
139
140     latest_data_window = self.stream_reader.data_buffer.get_most_recent(self.FFT_window_size)
141
142     self.fft = getFFT(latest_data_window, self.rate, self.FFT_window_size, log_scale = self.log_features)
143     #Equalizar a queda do espectro de ruído rosa:
144     self.fft = self.fft * self.power_normalization_coefficients
145     self.num_ffts += 1
146     self.fft_fps  = self.num_ffts / (time.time() - self.stream_reader.stream_start_time)
147
148     if self.smoothing_length_ms > 0:
```

Figura 30 - Determinação dos coeficientes, atualização da tela e dos recursos.

Nas primeiras linhas são é determinado os coeficientes, a quantidade fps utilizada, a frequência, entre outros. Na linha 133 é a função de atualização das estatísticas utilizadas,

usando os dados obtidos e a frequência, enquanto que a função abaixo atualiza os recursos, como a atualização da tela e também realiza a FFT na linha 142, utilizando a importação da função **getFFT** e também aplica a suavização nos dados resultantes.

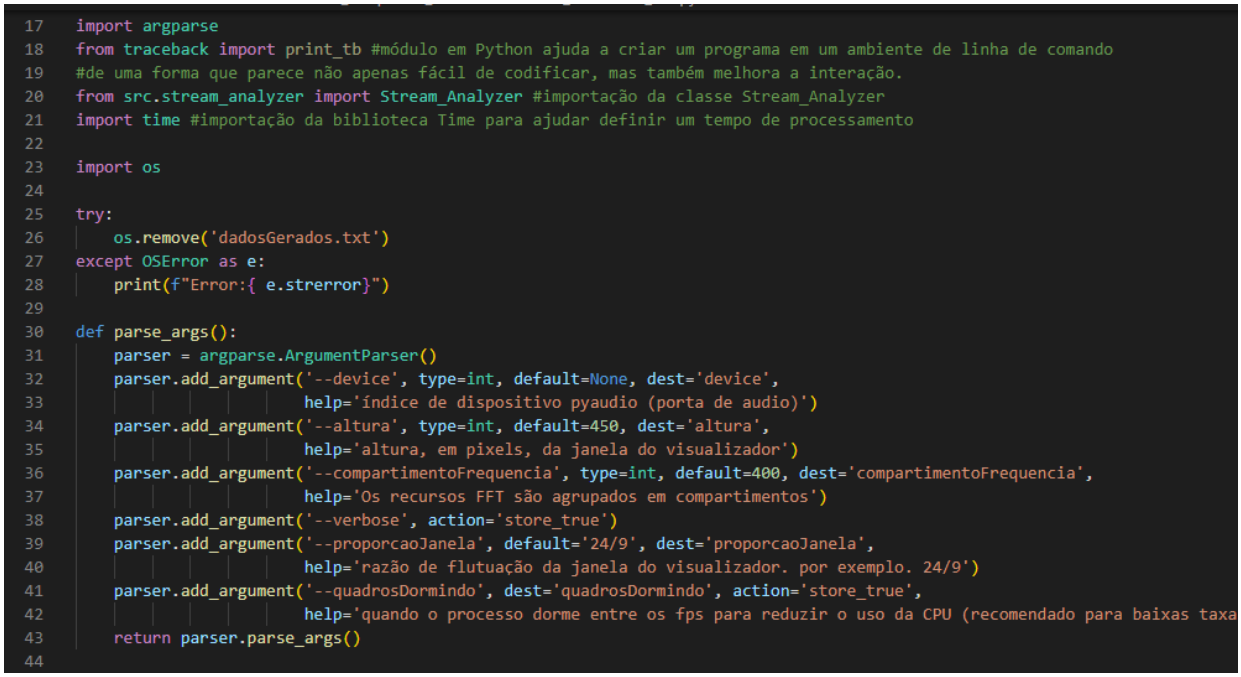
```
169     def get_audio_features(self):
170
171         if self.stream_reader.new_data:  #Verifica se o stream_reader tem novos dados de áudio que precisamos processar
172             if self.verbose:
173                 start = time.time()
174
175             self.update_features()
176             self.update_rolling_stats()
177             self.stream_reader.new_data = False
178
179             self.frequency_bin_energies = np.nan_to_num(self.frequency_bin_energies, copy=True)
180             if self.apply_frequency_smoothing:
181                 if self.filter_width > 3:
182                     self.frequency_bin_energies = savgol_filter(self.frequency_bin_energies, self.filter_width, 3)
183             self.frequency_bin_energies[self.frequency_bin_energies < 0] = 0
184
185             if self.verbose:
186                 self.delays.append(time.time() - start)
187                 avg_fft_delay = 1000.*np.mean(np.array(self.delays))
188                 avg_data_capture_delay = 1000.*np.mean(np.array(self.stream_reader.data_capture_delays))
189                 data_fps = self.stream_reader.num_data_captures / (time.time() - self.stream_reader.stream_start_time)
190                 print("\nAvg delay da fft: %.2fms  -- avg delay dos dados: %.2fms" %(avg_fft_delay, avg_data_capture_delay))
191                 print("Número de capturas de dados: %d (%.2ffps)-- cálculos de número fft: %d (%.2ffps)"
192                     %(self.stream_reader.num_data_captures, data_fps, self.num_ffts, self.fft_fps))
193
194             if self.visualize and self.visualizer._is_running:
195                 self.visualizer.update()
196
197         return self.fftx, self.fft, self.frequency_bin_centres, self.frequency_bin_energies
```

Figura 31 - Verificação de novos dados para processamento.

Esta função verifica se tem novos dados para serem processados, portanto, utiliza as atualizações da tela e de dados discutidos anteriormente e novamente aplica a suavização. E no fim apresenta no promp de comando o delay da FFT e o número de capturas de dados. Na linha 194 inicia a atualização e o seu retorno é a FFT aplicada e a frequência.

5.2.7 Executar_Analizador_FFT

Esta classe chama a classe mostrada anteriormente passando os parâmetros necessários e outros atributos, como a temporização.



```
17  import argparse
18  from traceback import print_tb #módulo em Python ajuda a criar um programa em um ambiente de linha de comando
19  #de uma forma que parece não apenas fácil de codificar, mas também melhora a interação.
20  from src.stream_analyzer import Stream_Analyzer #importação da classe Stream_Analyzer
21  import time #importação da biblioteca Time para ajudar definir um tempo de processamento
22
23  import os
24
25  try:
26      os.remove('dadosGerados.txt')
27  except OSError as e:
28      print(f"Error:{ e.strerror}")
29
30  def parse_args():
31      parser = argparse.ArgumentParser()
32      parser.add_argument('--device', type=int, default=None, dest='device',
33                          help='índice de dispositivo pyaudio (porta de audio)')
34      parser.add_argument('--altura', type=int, default=450, dest='altura',
35                          help='altura, em pixels, da janela do visualizador')
36      parser.add_argument('--compartimentoFrequencia', type=int, default=400, dest='compartimentoFrequencia',
37                          help='Os recursos FFT são agrupados em compartimentos')
38      parser.add_argument('--verbose', action='store_true')
39      parser.add_argument('--proporcaoJanela', default='24/9', dest='proporcaoJanela',
40                          help='razão de flutuação da janela do visualizador. por exemplo. 24/9')
41      parser.add_argument('--quadrosDormindo', dest='quadrosDormindo', action='store_true',
42                          help='quando o processo dorme entre os fps para reduzir o uso da CPU (recomendado para baixas taxa
43      return parser.parse_args()
44
```

Figura 32 - Inicialização da classe executar_Analizador_FFT.

É feito a importação de alguns módulos, como o **argparse**, que ajuda a criar um programa em um ambiente de linha de comando de uma forma que parece não apenas fácil de codificar, mas também melhora a interação e a classe **Stream_Analyzer** que é a responsável por fazer todo o mecanismo funcionar. A importação do **os** somente é para determinar se já tem um arquivo escrito anteriormente como o nome **dadosGerados**, se tiver é incluído para inserir os novos dados da gravação. Já na linha 30 é passado alguns argumentos que identificam o dispositivo, a altura, a proporção da janela e os quadros.

```
44
45  def converter_ProporcaoDeJanela(proporcaoJanela):
46      if '/' in proporcaoJanela:
47          dividendo, divisor = proporcaoJanela.split('/')
48          try:
49              float_razao = float(dividendo) / float(divisor)
50          except:
51              raise ValueError('window_ratio deve estar no formato: float/float')
52          return float_razao
53      raise ValueError('window_ratio deve estar no formato: float/float')
54
55  def executar_AnalizadorDeFFT():
56      args = parse_args()
57      proporcaoJanela = converter_ProporcaoDeJanela(args.proporcaoJanela)
58
59      ear = Stream_Analyzer(
60                      device = args.device,        # Índice de dispositivo Pyaudio (portaudio), padrão para a primeira entrada de microfone
61                      rate   = None,               # Taxa de amostragem de áudio, None usa as configurações de origem padrão
62                      tamanhoJanela_ms_FFT  = 60,    # Tamanho da janela usado para a transformação FFT
63                      atualizacaoPorSegundo  = 1000,  # Com que frequência ler o fluxo de áudio para novos dados
64                      tamanhoSuavizacao_ms = 50,    # Aplique alguma suavização temporal para reduzir recursos ruidosos
65                      n_compartimentoFrequencia = args.compartimentoFrequencia, # Os recursos FFT são agrupados em compartimentos
66                      visualize = 1,               # Visualize os recursos FFT com PyGame
67                      verbose   = args.verbose,    # Imprima estatísticas de execução (latência, fps, ...)
68                      altura    = args.altura,     # Altura, em pixels, da janela do visualizador,
69                      proporcaoJanela = proporcaoJanela  # Taxa de flutuação da janela do visualizador. por exemplo. 24/9
70                      )
71
72      fps = 60  #Com que frequência atualizar os quadros de FFT & tela
73      ultima_Atualizacao = time.time()
74      while True:
75          if (time.time() - ultima_Atualizacao) > (1./fps):
```

Figura 33 - Função converter proporção e de executar o analisador.

A função mais acima tem como objetivo converter para a proporção de tela do usuário, fazendo o cálculo por pontos flutuantes. A **executar_AnalizadorDeFFT** é a função principal que chama a função de proporção de tela, e a função **Stream_Analyzer** passando os argumentos necessários, como o dispositivo de entrada, a taxa de amostragem, as atualizações, a suavização e a proporção da janela, por exemplo. Mais abaixo é definido a quantidade máxima de fps que serão alcançados, porém, é importante ressaltar que isto é variável de acordo com o computador que está sendo utilizado pelo o usuário. Na linha 73 a última atualização é descrita pelo o tempo decorrido desde o início do programa.

```
74      while True:
75          if (time.time() - ultima_Atualizacao) > (1./fps):
76              ultima_Atualizacao = time.time()
77              raw_fftx, raw_fft, binned_fftx, binned_fft = ear.get_audio_features()
78          elif args.quadrosDormindo:
79              time.sleep(((1./fps)-(time.time()-ultima_Atualizacao)) * 0.99)
80
81  if __name__ == '__main__':
82      executar_AnalizadorDeFFT()
83
```

Figura 34 - Chamada infinita do programa e inicialização do programa.

É feito um **while true** para que o programa sempre esteja em execução, até que seja encerrado, portanto é determinado o tempo decorrido deste a última atualização e também verifica se tem quadros que não estão sendo usados, o que diminuí a quantidade de

processamento necessário do computador. E na linha 81, é descrita executada a função de inicialização do programa.

5.2.8 geraAudioeFFT

Esta é outra classe principal que recebe o txt gerado pela a classe **fft**.

```
18  import soundfile as sf
19  import numpy as np
20  import wave
21  import matplotlib.pyplot as plt
22  from src.fft import getFFT
23
24  from numpy import loadtxt
25
26
27  #importar arquivo de texto para o vetor NumPy
28  data = loadtxt('dadosGerados.txt')
29
30  #Quantidade padrão de taxa de amostragem
31  rate = 44100
32
33  #Gerar o arquivo de audio com os dados coletados
34  sf.write('dados.wav', data, rate)
35
36  sf.write('dadosAmplificados.wav', data*10, rate)
37
38  time = np.arange(0, len(data) * 1/rate, 1/rate)
39
40  #Plot do audio original
41  plt.figure(1)
42  plt.title('Sinal Original Capturado')
43  plt.plot(time, data)
44  plt.xlabel('Tempo (s)')
45  plt.ylabel('Amplitude')
46  plt.show()
```

Figura 35 - Importação do arquivo de texto, geração do áudio e plote do gráfico.

É feita a importação de algumas bibliotecas que fazem o processamento de áudio como a **soundfile**, outra que realiza operações matemáticas como a **numpy**, outra para abrir áudio e para plotar os gráficos. Entretanto, primeiramente é preciso abrir o arquivo de texto gerado anteriormente e também definir a taxa de amostragem, que por padrão é 44100. Na linha 34 é exportado o arquivo de áudio chamado **dados.wav** com estes parâmetros e também o áudio amplificado 10 vezes para a comparação. Assim é necessário determinar o tempo de cada amostra para plotar o gráfico do áudio. Com todas essas informações dadas, é plotado o gráfico com o título de **Sinal Original Capturado**, e tem como eixo das ordenadas a amplitude e das abcissas, o tempo em segundos.

```
46  n = len(data)
47  tx = 200
48
49  freq = np.fft.fftfreq(n)
50  mascara = freq > 0
51
52  fft_calculo = np.fft.fft(data)
53  fft_abs = 2.0*np.abs(fft_calculo/n)
54
55  #Plot da FFT
56  plt.figure(2)
57  plt.title('FFT do audio')
58  plt.plot(freq[mascara], fft_abs[mascara])
59  plt.xlabel('Frequência (s)')
60  plt.ylabel('Amplitude (V)')
61  plt.show()
```

Figura 36 - Calcula da FFT dos dados e plote do gráfico.

Agora é preciso plotar o gráfico da FFT, contudo, na linha 46 é guardado em uma variável o tamanho dos dados, o tamanho do eixo das abcissas (**tx**) e a frequência é calculada pela a função **fftfreq** do **numpy**. A **mascara** serve somente para receber os valores positivos da FFT, pois a mesma quando é gerada é simétrica em relação ao eixo das ordenadas. Na linha 52 é calculada a FFT com os dados repassados, porém para não se ter valores negativos, é aplicado a função **abs** que é o módulo do sinal. Das linhas 56 a 61 é somente para plotar o gráfico com o título de **FFT do áudio**, recebendo só os valores positivos da frequência e da FFT e assim é gerado o gráfico.

4.3 Prints do funcionamento

Para executar primeiramente o projeto é necessário executar a classe **executar_Analizador_FFT** pelo o promp de comando, digitando o seguinte comando em python (abra o terminal na pasta raiz do projeto): python3 .\executar_Analizador_FFT.py

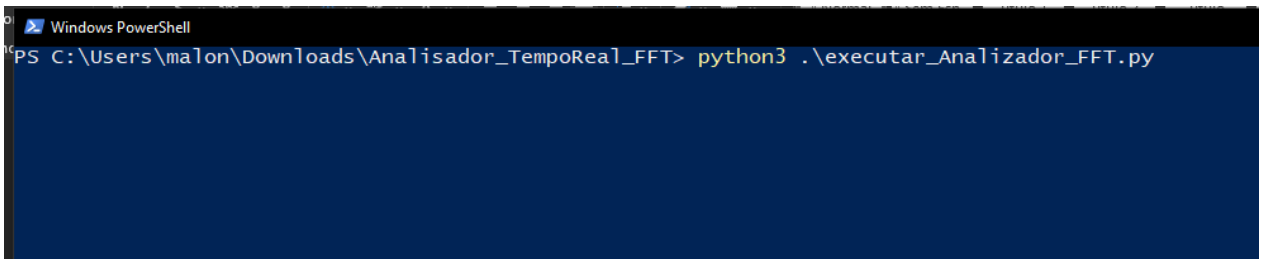


Figura 37 - Comando para executar o analisador de espectro.

Para Linux utilize o comando: python executar_Analizador_FFT.

Após pressionar Enter surgirá na tela o seguinte resultado:

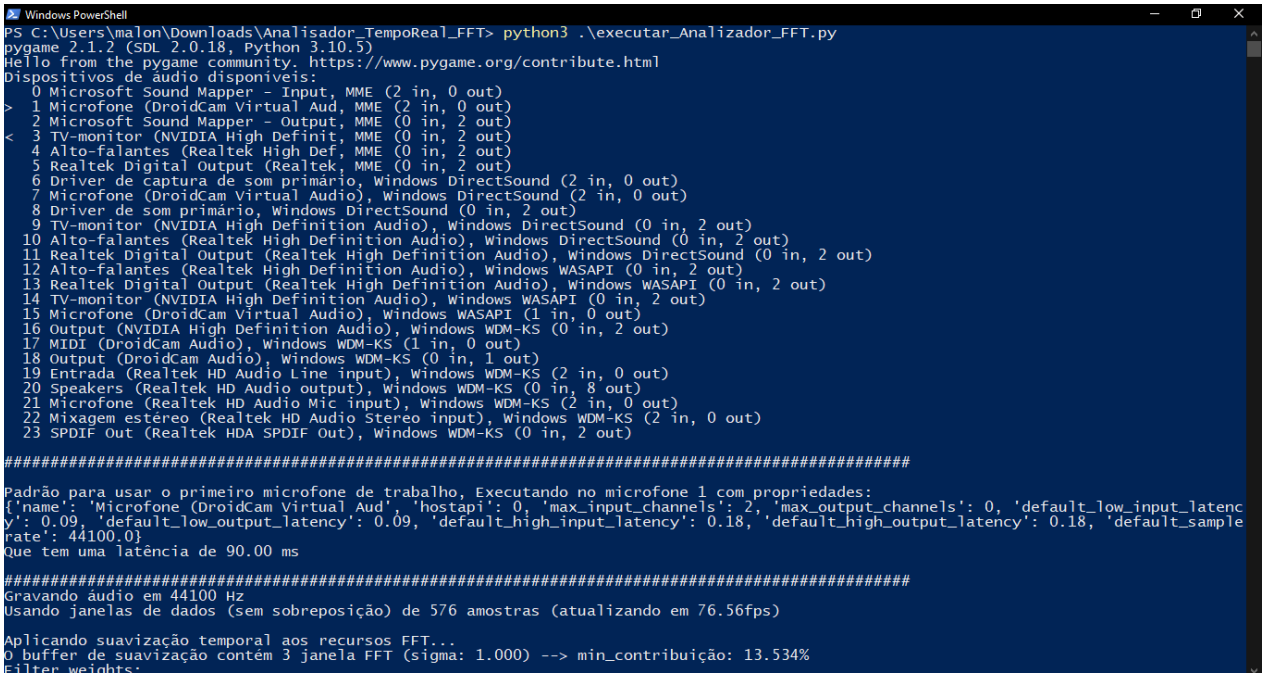


Figura 38 - Informações apresentadas para o usuário.

Apresenta todos os dispositivos de sons tanto de entrada, quanto de saída, apenas posteriormente é verificado o microfone correto e o seleciona demonstrando mais abaixo no terminal.

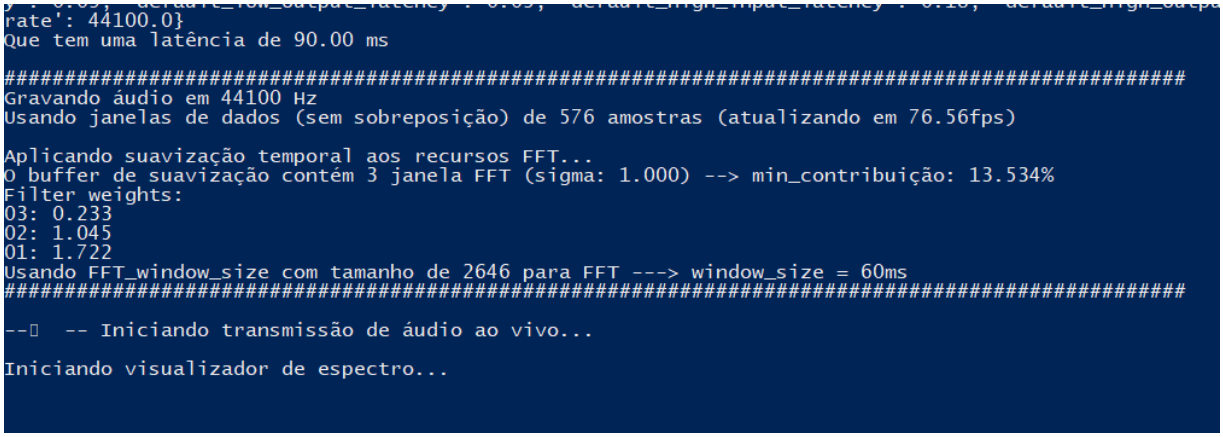


Figura 39 - Parâmetros de utilização da suavização.

Mais abaixo mostra a taxa de amostragem e a latência que ocorre, ou seja, a demora para o cálculo da FFT. Ademais, é apresentada, os parâmetros da suavização aplicada e o janelamento e por fim inicia uma segunda janela, que é a do **pygame**.

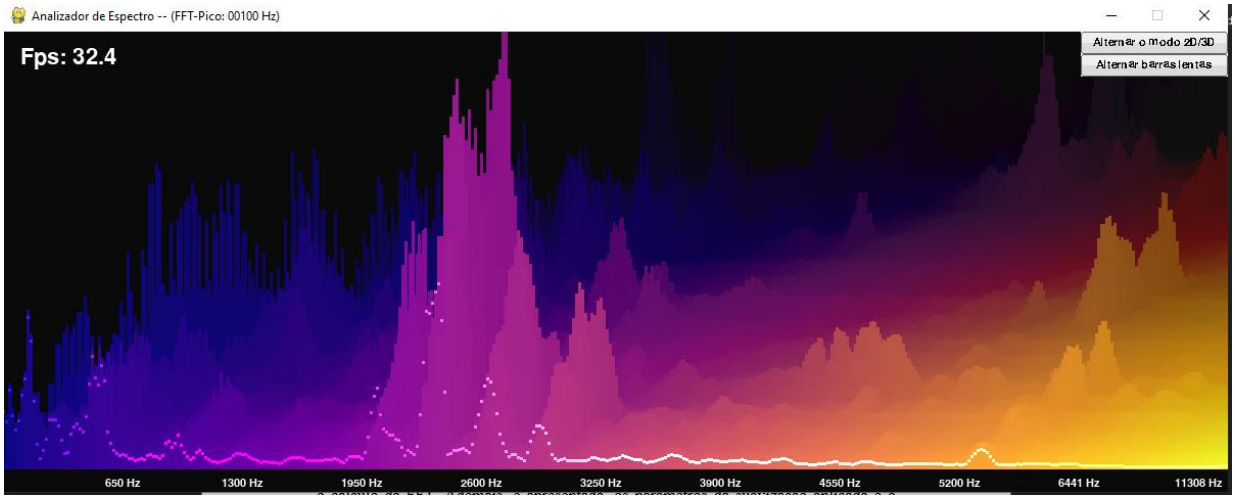


Figura 40 - Modo 3D de visualização.

Este é o modo 3D da FFT, na qual apresenta como título da janela o nome **Analizador de Espectro**, e o pico que foi alcançado, neste caso em particular, com valor de 00100Hz. Na tela central temos os picos de espectros em tempo real de forma colorida, as quais as mais baixas frequências são as azuis e as mais altas são as amarelas. Em baixo temos a escala de frequências que vão desde 0 até a 11308 Hz. É importante também notar que neste modo, temos um breve histórico das frequências que estão ao fundo. No canto superior esquerdo, temos a quantidade de FPS que estão sendo processados em tempo real e no canto direito, há dois botões – o de alternar barras lentas, que somente vai colocar ou retirar as barras descritas nas frequências e outro botão **Alternar o modo 2D/3D** moda a visualização do usuário como intuitivamente quer alegar. Ao clicar neste botão, temos:

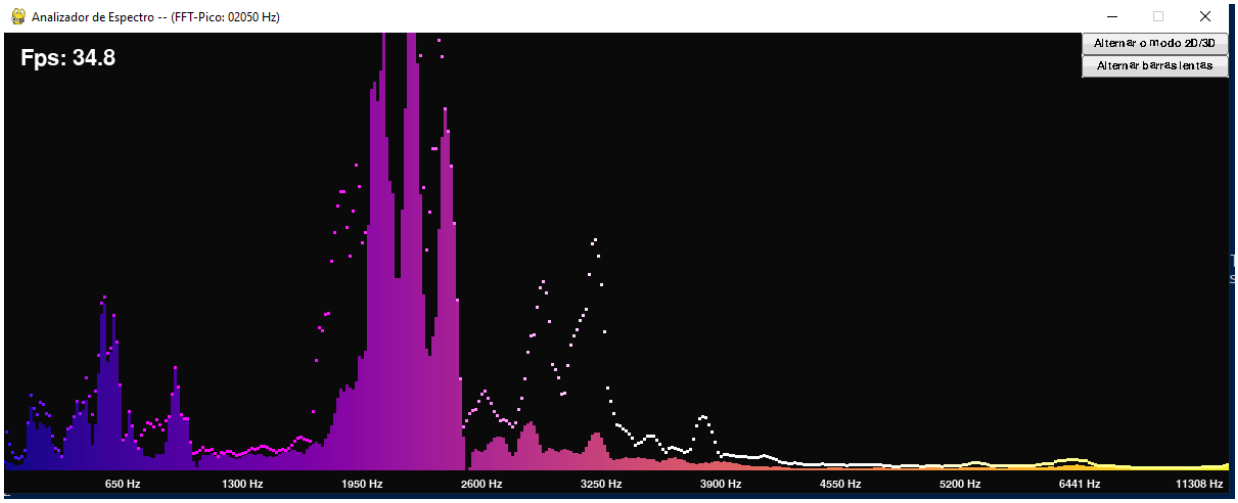


Figura 41 - Modo 2D de visualização.

Como se pode perceber, somente muda a visualização para 2D, e neste modo não tem o histórico da FFT como há no outro modo.

Pode parar o programa, abrindo o terminal do qual estão executando o programa é pressionar **Ctrl+c**. E se verificar o diretório da pasta raiz do projeto:

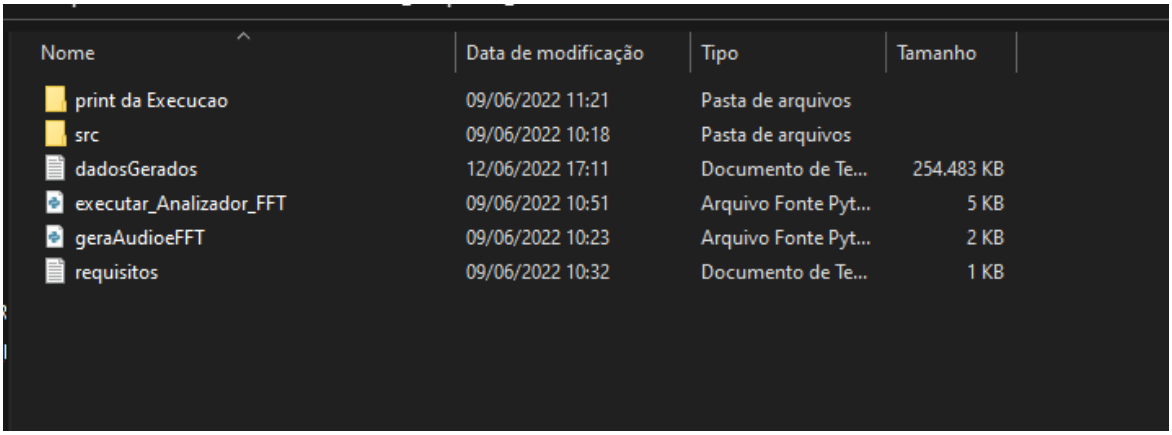


Figura 42 - Arquivo dadosGerados.txt no diretório do projeto.

Foi gerado um arquivo chamado **dadosGerados** que é justamente as amostras coletadas durante todo o buffer da gravação. Se abrimos o arquivo:

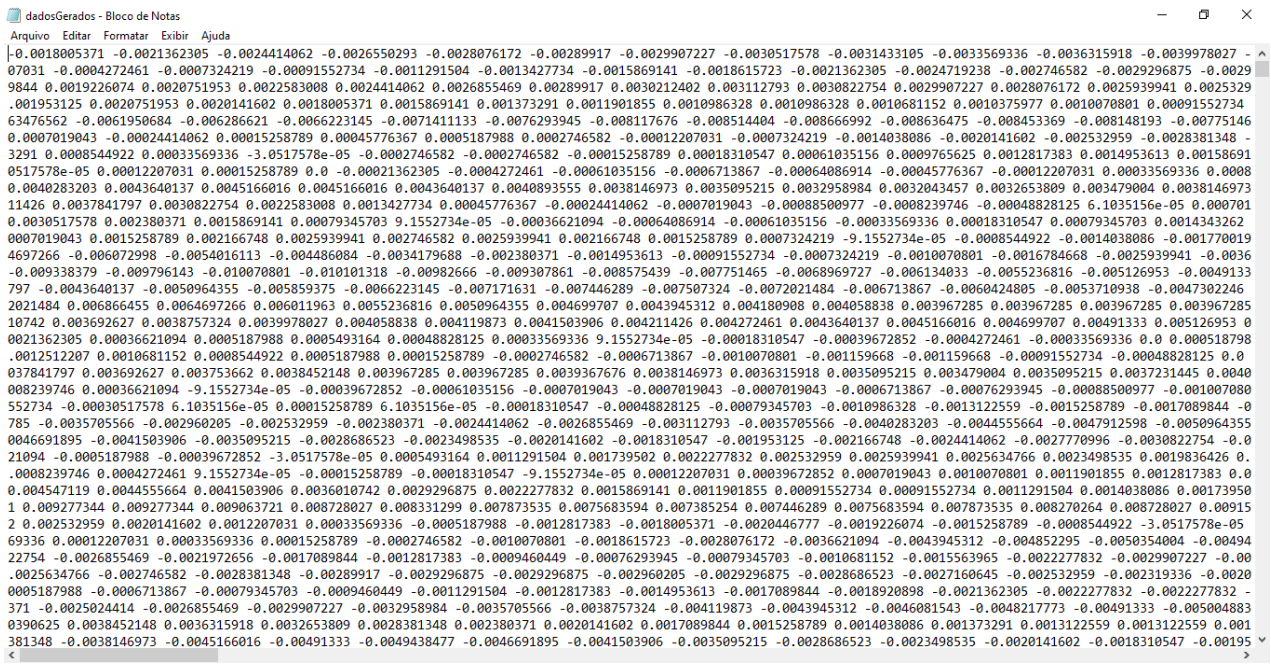


Figura 43 - Conteúdo do dadosGerados.txt.

Há todas as amostras de forma sequencial que vai servir para gerar os áudios e plotar os gráficos. Quando executada a classe **geraAudioeFFT**, esta por sua vez, gera dois arquivos de áudio no diretório:

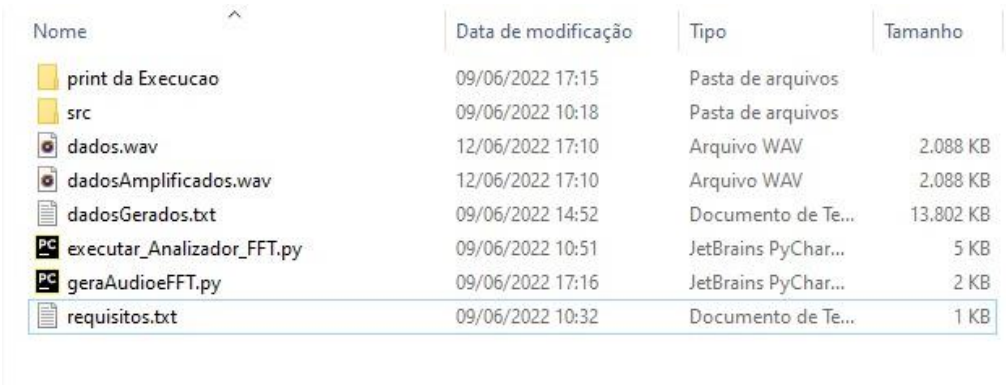


Figura 44 - Arquivos de áudios gerados.

Que é **dados** e **dadosAmplificados**, justamente os arquivos de áudios reconstruídos pela a amostra, que servem para realizar a comparação entre ambos.

5. CONCLUSÕES

Tendo em vista o que foi exposto ao longo do trabalho, compreende-se de forma geral que a disciplina de Processamento Digital de Sinais, nos possibilitou como estudantes a entendermos conceitos importantes da área de propagação e tratamento de sinais, e também, o mais importante, a trabalharmos na prática conceitos abordados durante as aulas. Dessa forma, a amplificação de sinal de voz nos possibilitou entendermos melhor a aplicação da Transformada Rápida de Fourier (FFT) e também, de como podemos aplicar essa análise utilizando bibliotecas e ferramentas que nos auxiliem durante a pesquisa.

6. RECOMENDAÇÕES

Para fins de executar o programa, inicialmente é necessário instalar o arquivo “requirements.txt”, com o seguinte comando, `pip install -r requirements.txt`. Esse arquivo irá instalar os seguintes módulos necessários:

- Pygame (https://www.pygame.org/wiki/GettingStarted) Version: 1.9.6
- Pyaudio (http://people.csail.mit.edu/hubert/pyaudio/) Version: 0.2.11
- Scipy (https://www.scipy.org/install.html) Version: 1.4.1
- Numpy (https://numpy.org/install/) Version: 1.18.4
- Matplotlib (https://matplotlib.org/) Version: 3.2.1

7. REFERÊNCIAS

- Ler e gravar arquivos .WAV, “https://docs.python.org/3/library/wave.html”, acessado em junho de 2022.
- Biblioteca NumPy, “https://numpy.org/install/”, acessado em junho de 2022.

8. TRABALHOS FUTUROS

O trabalho desenvolvido, nos possibilita a entendermos melhor como trabalhar com sinais de áudio e poder utilizar ferramentas tão bem desenvolvidas para essa finalidade, com isso, podemos aprimorar nosso trabalho ao buscar desenvolver um algoritmo que pudesse detectar frequências de sons, imperceptíveis aos ouvidos humano, e tentar amplifi-las tornando-as audíveis.

9. ANEXOS

Link do repositório do código: <https://github.com/alaimcosta/amplificadorSinalFft>