Assignment 1: Fourier Transform

Students Name & ID	Yukun Yuan(2831718Y)、Xueting Pan(2788786P)、Shengtong (2729698Y)、Yang Liu(2749545L)
Subject	Digital Signal Processing
Course ID	ENG 5027
Lecturer/Tutor	Scott Watson & Paul Harvey
Submission Date	18/10/2022

Github Repository: https://github.com/CharlieYuan650/Digital-Signal-Processing-LAB-CODE

1. Introduction

The voice used in this project is in the form of wav (16bit PCM). In addition, the project firstly imported the voice data of the wav file into the python development environment, then carried out voice analysis and voice enhancement, and finally realized the vowel detection of the voice signal. The entire project has a full code development history and canonical meeting minutes, which can be found on GitHub. Also, the complete code screenshot will be in the appendix.

(Github Repository: https://github.com/CharlieYuan650/Digital-Signal-Processing-LAB-CODE)

2. Task 1 - Loading Audio into Python

Goal:

Load audio into a python application and draw time domain graphs; also draw frequency domain graphs by applying Fourier transform.

Methodology:

Commonly used audio processing toolkits include wave toolkit, librosa toolkit, scipy toolkit, etc. Based on Fourier transform this task will apply the wave toolkit and librosa toolkit to read and process speech signals in wav format (16bit PCM).

Steps:

1) Read the voice signal file in wav format

Use the wav toolkit to read the voice signal in wav format, and obtain related parameters such as channel framerate and frames. Since the speech data read by the wav toolkit is binary, in order to visualize the time domain diagram of the speech signal, the read binary data is first converted into decimal and normalized. Finally, the matplotlib drawing toolkit is used for visual display, and the time domain diagram of the speech signal is drawn.

Its normalization is handled as follows:

```
wave_data1 = wave_data1 * 1.0 / (max(abs(wave_data1)))
```

The acquired speech signal related parameters are shown below. It can be seen that the speech signal is mono, the sampling frequency is 44.1KHz, the number of frames sampled is 179675, and it is an uncompressed data file.

```
voice_enhancer
C:\Users\Tom&Jerry\PycharmProjects\practice\venv\Scripts\python.exe C:/Users/Tom&Jerry/PycharmProjects/practice/voice_enhancer.py
_wave_params(nchannels=1, sampwidth=2, framerate=44100, nframes=179675, comptype='NONE', compname='not compressed')
```

2) Fourier transform is performed on the speech signal to obtain the spectrogram of the original sound signal

In order to obtain the spectrogram of the original signal, we will use the functions provided by the librosa toolkit to perform FFT transformation on the read speech signal. The function called is as follows, where n_fft represents the window size of the FFT, which is set to 256 in this project; hop_length represents the frame shift, which is set to 128 frames; win_length represents the window length, which is set to 256.

```
siginal = librosa.stft(originalData, n_fft=256, hop_length=128,
win_length=256)
```

Finally, the signal is visualized through the matplotlib drawing toolkit, and the frequency axis is divided by logarithmic scale.

Results:

The time domain diagram is shown below (***figure 2.1***), and it can be seen that the time of the speech signal lasts about 4s. About the first 0.8 seconds and the last 2.2 to 4 seconds are noise signals. After 0.8 seconds, people start to speak, and it can be seen that the speech signal has obvious amplitude changes.

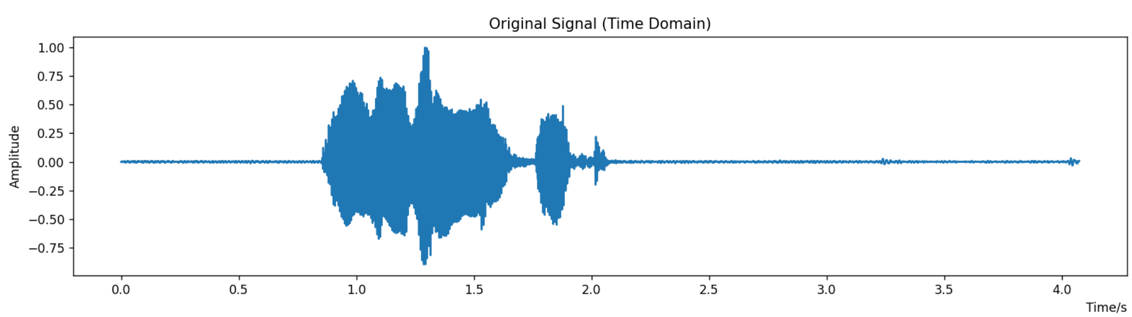

Figure 2.1 The Time Domain Diagram of Voice Signal

The frequency domain diagram after the FFT transformation is shown below (***figure 2.2***). It can be seen that 0-200 Hz is mainly a noise signal with a low decibel value.

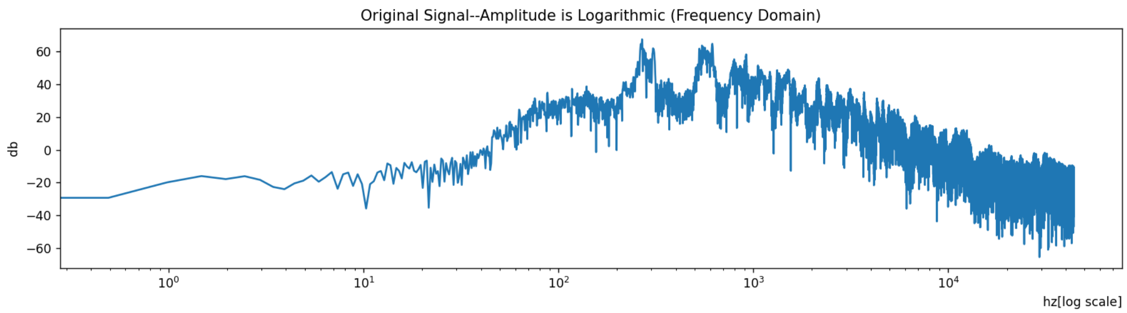

Figure 2.2 The Frequency Domain Diagram of Voice Signal

3. Task 2 - Audio Analysis

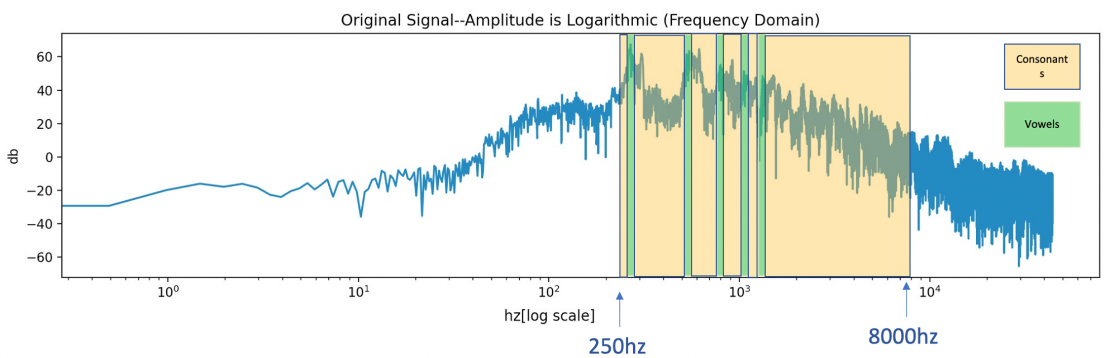

Figure 3.1 The Diagram of Consonants and Vowels Regions

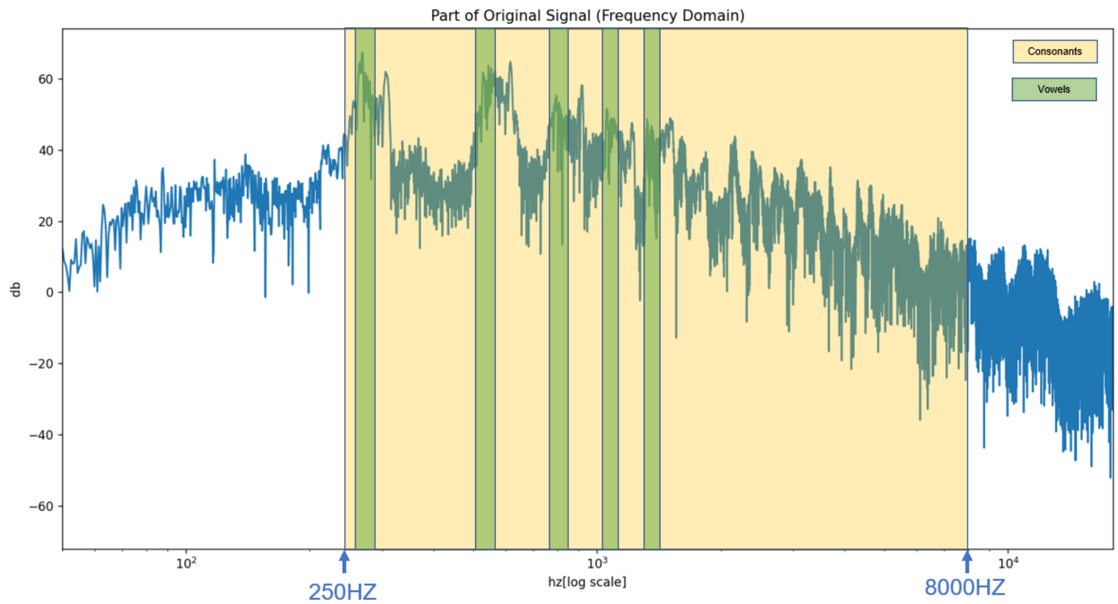

Figure 3.2 The Enlarged Diagram of Consonant and Vowel Regions

"Acoustic Phonetics" based on Cambridge, Stevens, K. N. (1998) mentions that vowels range from 250Hz to 4000Hz and consonants range from 250Hz to 8000Hz. The spectrum is analyzed based on literature knowledge, and vowels, consonants and harmonics are marked in the ***figure 3.1*** and ***figure 3.2***. The analysis is as follows:

1. Mark the fundamental frequency of the vowel signal on the spectrogram. The fundamental frequency is the first place where the peak is generated, which is the corresponding fundamental frequency, and the corresponding frequency range is about 260Hz.

2. The position of the consonant in the spectrogram is the range between the two formants. The yellow area in the figure is the consonant, you can see that the frequency range of the consonant is between 250Hz-8000Hz.

3. Consonants have no harmonics, whereas the harmonics of vowels are where formants occur. The green part marked on the picture is the harmonic of its corresponding vowel.

4. Task 3 - Speech Enhancement by Applying Fourier Transform

Goal:

The aim of this task is to modify the audio signals and increase the voice quality. We aim to strengthen the amplitude of the highest harmonic frequency and reduce the noise to improve voice quality.

Methodology & Steps:

The flow chart of realizing speech enhancement is shown in ***figure 4.1***. After reading the speech data, the FFT transform is used to obtain the energy spectrum of the signal, and then the improved spectral subtraction method is used to reduce the noise of the speech signal. Also, the range of the highest harmonic frequency of the speech signal is found, and its amplitude is strengthened. Finally, IFFT is used to restore the signal, and to improve the quality of speech.

The spectral subtraction method uses the spectrum of the noisy signal to subtract the spectrum of the noisy signal to reduce noise. Its mathematical principle is as follows:

$$| X(\varpi)^2 | = | Y(\varpi)^2 | - | D(\varpi)^2 |$$

eq(1)

Where Y(w) is the bath signal, D(w) is the noise signal, X(w) is the enhanced speech signal.

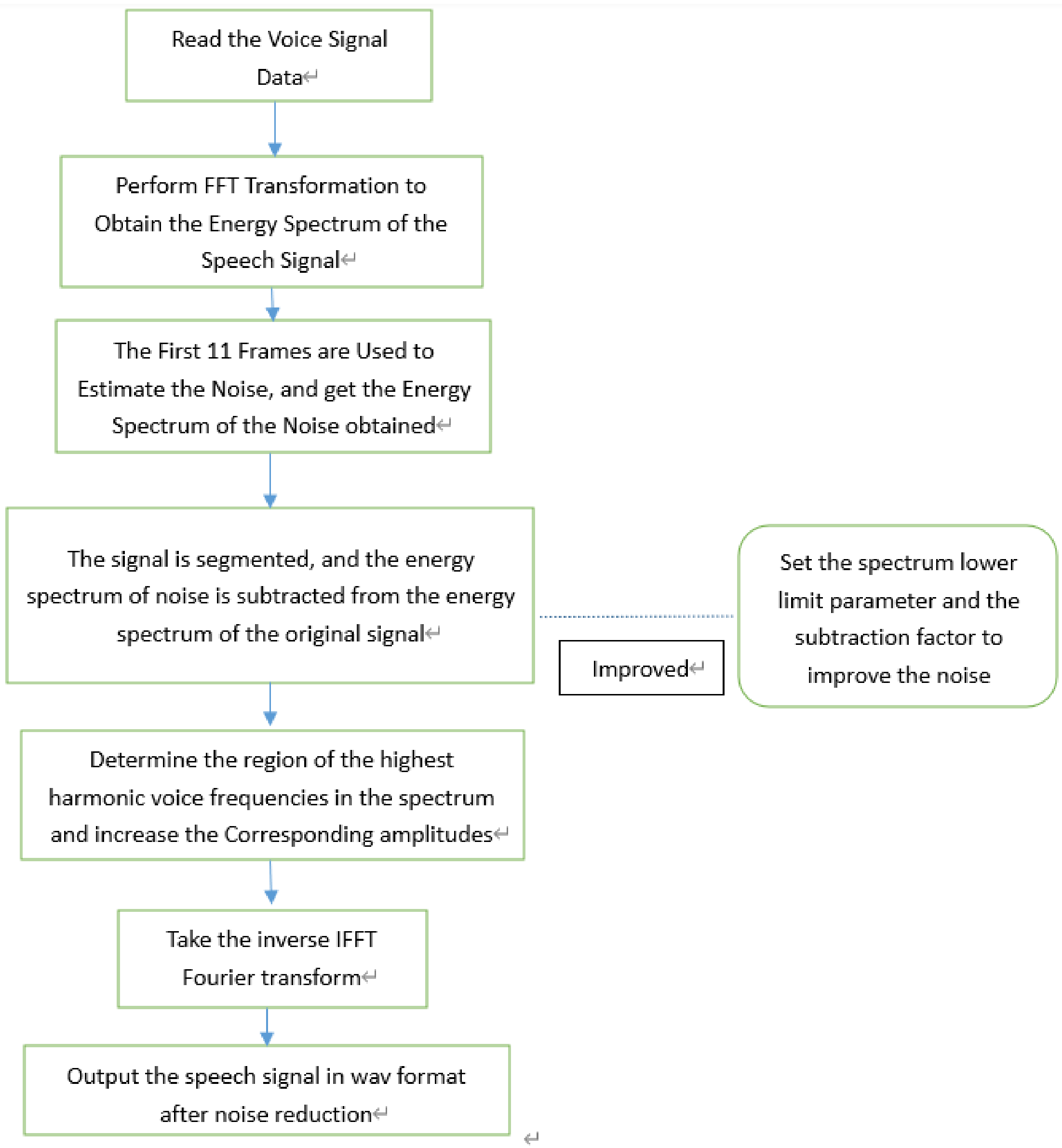

Figure 4.1 The Flow Chart of Realizing Speech Enhancement

In order to prevent the negative value ***(eq1)*** of the original speech spectrum from subtracting from the estimated noise spectrum; here introduce a spectrum lower bound in the algorithm ***(figure 4.1)*** -- there will be set a lower limit when the spectrum is subtracted and there is a negative value, and this lower limit is equal to the spectrum lower limit parameter (beta) times the absolute value of the negative value. On the contrary, if the algorithm is not improved, or simply setting the negative number to 0 will lead to small independent peaks in the signal frame spectrum at random locations and generate musical noise ***(figure 4.2)***.

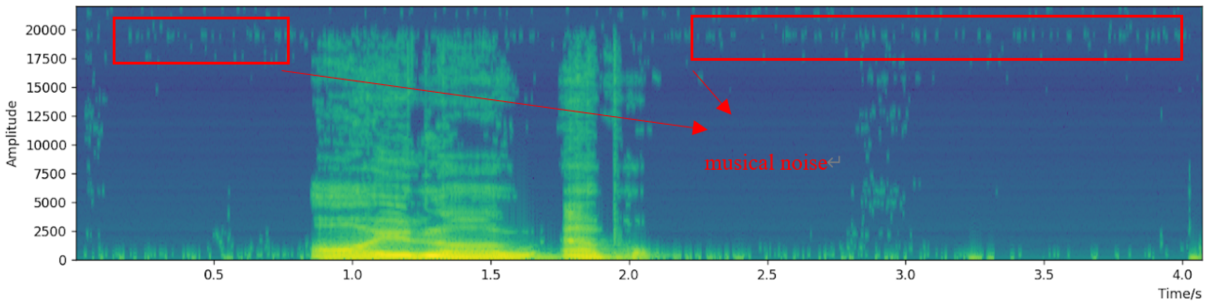

Figure 4.2 The Example of Generate Random Musical Noise

Meanwhile, the value of beta also needs to be set very carefully -- too large beta will cause residual noise to become loud, so beta can only approach zero indefinitely.

In addition, the adjustment of the subtraction factor Alpha is also important -- Alpha is the weight multiplied by the noise, and improper alpha can affect the distortion of the signal. Here, under the condition that the beta is 0.0001 and the number of noise frames is 20, the alpha is set to 2,4,6, and 10, and output their speech signals respectively. The result shows that the speech effect of parameters 4 sounds the best.

Results:

Finally, after finishing the noise reduction processing, here determined the highest range of harmonic frequency as 6000 hz-8000 hz and enhanced it (as shown in the frequency domain ***figure 4.3***)

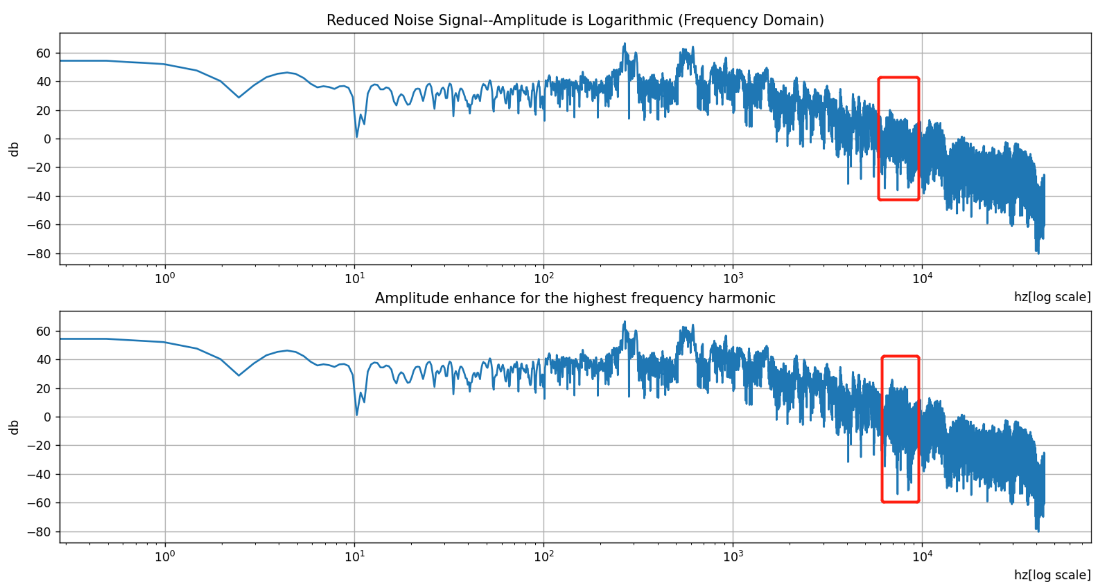

Figure 4.3 The Example of Generate Random Musical Noise

figure 4.4 is the time domain diagram of unenhanced and enhanced speech signals, from which it can be seen that the amplitude of noise signals in the first 0.8 seconds has been significantly weakened. Also, the last subsequent 2.2 to 4 second signal changed to some extent.

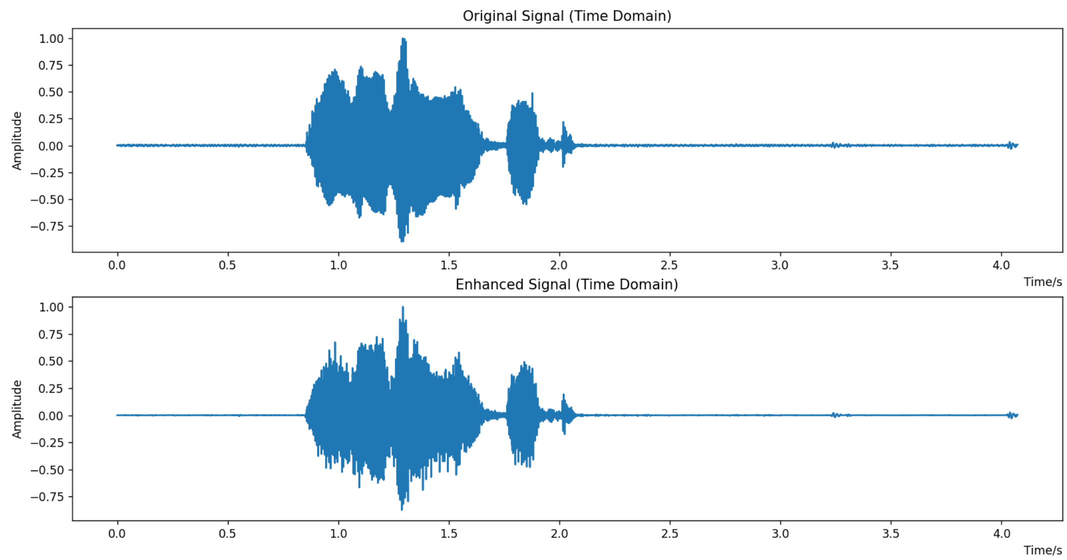

Figure 4.4 The Time Domain Diagram of Unenhanced and Enhanced Voice Signals

figure 4.5 is the spectrum diagram of unenhanced and enhanced speech signals over time. It can be clearly seen from the color changes that the speech signals have been significantly enhanced compared with those before.

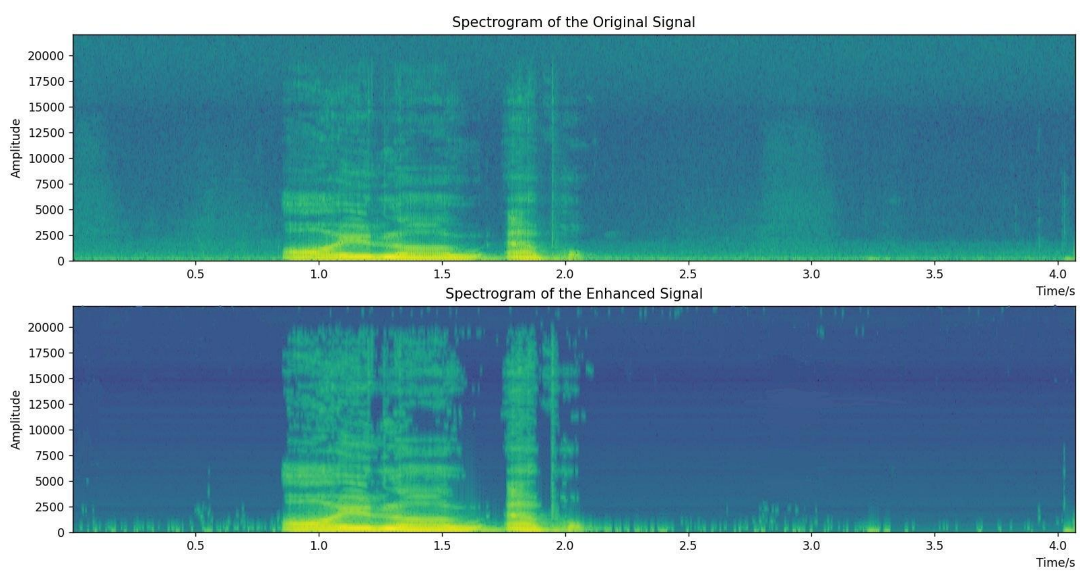

Figure 4.5 The Spectrogram of Unenhanced and Enhanced Voice Signals

5. Task 4 - Vowel Detector

Goal:

The goal of this task is to identify at least 2 vowels from the audio files (in wav format), given every single wav file contains the recording of a single spoken vowel. In order to do this, we are asked to write a simple function which takes a wav file as input, and a vowel (as string) as output, the following is an illustration of such function:

```
def voweldetector(wavfile):
    #<code here>
    vowel = ""
    return vowel
```

Methodology:

The frequency spectrum of a spoken vowel can be interpreted as a distribution of a fundamental frequency and its corresponding harmonic frequencies (formant). The amplitudes of all those frequencies differ from each other in approximate ratios. By identifying those frequencies and the amplitude ratio tied to each frequency, we can generate a pattern consisting of frequencies and amplitudes in 2D array format:

[[freq0, amp0], [freq1, amp1], [freq2, amp2].......]

The 2D array can be seen as a fingerprint for each vowel on a frequency spectrum. We have labeled vowel files (in wav format) as training data. First, we generate patterns from those labeled files and assign "fingerprint" to each known vowel. We store these pieces of information in a dictionary. Here is an example:

{"A" : [[freq0, amp0], [freq1, amp1], [freq2, amp2].......],
"E" : [[freq0, amp0], [freq1, amp1], [freq2, amp2].......],.......}

Then, we take a new input file and generate its pattern. If the input file has a pattern that matches the "fingerprint" in the archived dictionary, we are able to infer which spoken vowel is contained in the input file.

Steps:

***Please be noted: the following functions are called by a single function voweldetector()**

1. Find patterns of all the known vowels, and store the pattern information in a dictionary.

```
vowel_array = ["A", "E", "I", "O", "U"]
def PatternFinder(self):
       Dictionary = {}
```

```
        #Find pattern for each vowel in vowel_array
        for vowel in VowelDetector.vowel_array:
            #Load wav file, and get frames and parameters
            file, param = VowelDetector.LoadFile(self, vowel, True)
            wave_data = np.frombuffer(file, dtype=np.short)  # convert
into decimalism
            wave_data = wave_data * 1.0 / (max(abs(wave_data)))  #
normalization
            framerate = param[2]

            #Generate frequency spectrum and mark patterns
            xf, freqs = VowelDetector.FFT(self, wave_data, framerate)
#Forier Transform
            refine_peaks, refine_amps = VowelDetector.PeakFinder(self, xf)
#Find frequency pattern


            #Generate vowel's pattern in 2D array format
            Pattern = list(zip(freqs[refine_peaks],
refine_amps['peak_heights']))

            #Store vowel in dictionary as key, and pattern as value
            Dictionary[vowel] = Pattern

        # print(Dictionary)
        return Dictionary
```

2. Find the pattern of the input file, and iterate through the archived dictionary in previous steps to find the best match.

```
def PatternMatch(self, wav, param, pattern_dic):
        wave_data = np.frombuffer(wav, dtype=np.short)  # convert into
decimalism
        wave_data = wave_data * 1.0 / (max(abs(wave_data)))  #
normalization
        framerate = param[2]
        xf, freqs = VowelDetector.FFT(self, wave_data, framerate) #Forier
Transform
        new_peaks, new_amps = VowelDetector.PeakFinder(self, xf) #Find
frequency pattern
        new_pattern = list(zip(freqs[new_peaks],
new_amps['peak_heights']))
```

```
        sum_amp = sum(new_amps['peak_heights'])


        # Match new wav file pattern with stored wav file pattern
        match_vowel = "No Match"
        fr_match_ref = 0
        for key in pattern_dic:
            print("Key------------------------", key)
            fr_match = 0
            key_pattern = pattern_dic[key]
            key_sum_amp = 0.1
            # Sum up all amp in key_pattern
            for pair in key_pattern:
                key_sum_amp += pair[1]

            for key_pair in key_pattern:
                for new_pair in new_pattern:
                    if abs(key_pair[0] - new_pair[0]) < 10: # Compare
frequency peak with a tolerance of 10Hz
                        new_amp_ratio = (new_pair[1] / sum_amp) * 100
                        key_amp_ratio = (key_pair[1] / key_sum_amp) * 100
                        if abs(key_amp_ratio - new_amp_ratio) <= 5: #
Matched if corresponding peak has similar amplitude
                            print("amplitude ratio difference = ",
abs(key_amp_ratio - new_amp_ratio))
                            print("match at freq: ", key_pair[0],
new_pair[0])
                            fr_match += 1
            print("fr_match = ", fr_match)
            if fr_match > fr_match_ref:
               match_vowel = key
               fr_match_ref = fr_match

        return match_vowel
```

Results:

By applying our method, we successfully identified 4 of the 5 spoken vowels (A, E, I, O, U). Here is an example of how we identify “E” with our voweldetector() function:

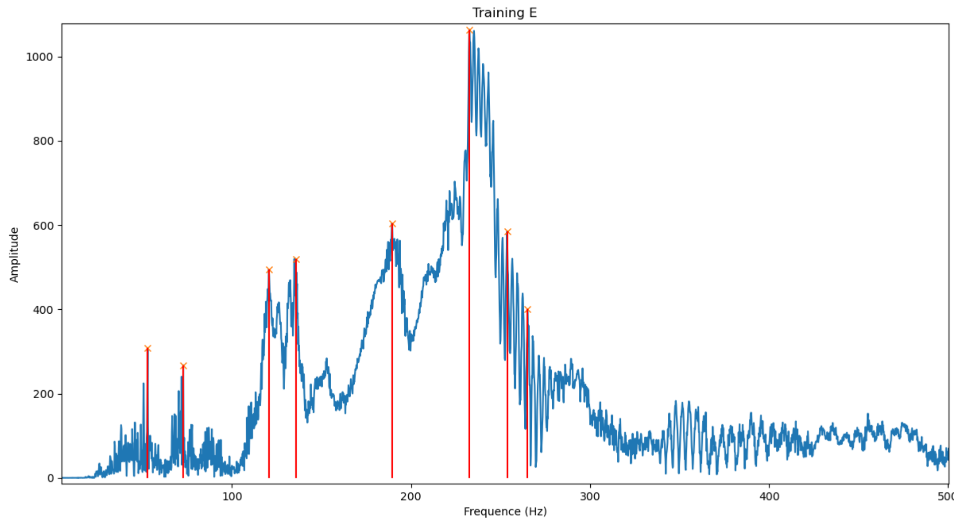

Figure 5.1 - Training Data

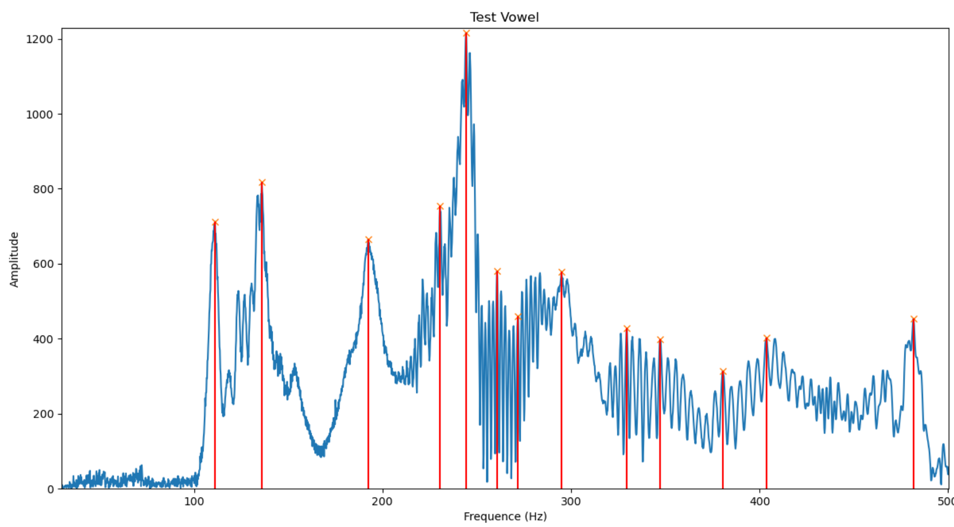

Figure 5.2 - Input Data

In ***figure 5.1*** and ***figure 5.2***, despite the differences on the spectrum, we can see that the vowel - E which we used for training has a very similar pattern with the unknown input vowel. Our function has a certain degree of tolerance for deviation, with some fuzzy processing, it is able to identify the spoken vowel - E. Here are 5 matched frequencies between the training data and the input data:

Key----------------------- E
amplitude ratio difference = 2.4931289476962455
match at freq: 120.59286051265923 111.13507573831232
amplitude ratio difference = 1.7549912718838883
match at freq: 135.61880798867747 135.9652548221875
amplitude ratio difference = 1.85334688956962

match at freq: 253.70734392200032 244.29692324457844
amplitude ratio difference = 1.975801516952754
match at freq: 265.07312470514233 260.7168803806894
amplitude ratio difference = 3.5066319426767336
match at freq: 265.07312470514233 271.930509644375
fr_match = 5

6. Appendix

voice_enhancer.py

```
import wave
import matplotlib.pyplot as plt
import numpy as np
from scipy.fft import fft
import librosa
import soundfile as sf

if __name__ == "__main__":

   f = wave.open("original.wav", "rb")
   parameters = f.getparams()
   print(parameters)  # Print parameters of wav file
   nchannels = parameters[0]  # Get channel parameters
   framerate = parameters[2]  # Get sampling frequency
   nframes = parameters[3]  # Get the number of sampling points
   data = f.readframes(nframes)
   f.close()
   wave_data1 = np.fromstring(data, dtype=np.short)  # convert data to decimal
   # Using Normalization processing due to the original data is a single
channel
   # (there is no need to reshape the wave_data)
   wave_data1 = wave_data1 * 1.0 / (max(abs(wave_data1)))
   x1 = np.arange(0, nframes) * (1.0 / framerate)  # Horizontal and vertical
time coordinates

   # Voice Enhancement
   # Calculate the spectrum of the signal, the frame length win_length, the
frame shift hop_length,
   # And the number of Fourier transform points is n_fft=256 points
   originalData, fs = librosa.load("original.wav", sr=None)
   siginal = librosa.stft(originalData, n_fft=256, hop_length=128,
win_length=256)  # D x T
   D, T = np.shape(siginal)
```

```
   amplitude = np.abs(siginal)
   phase = np.angle(siginal)
   power = amplitude ** 2  # get the energy spectrum of the signal
   print(fs)
   # Estimate the energy of a noisy signal
   # Since the noise signal is unknown, it is assumed that the first 20 frames
of the noisy signal are noise
   amplitude_nosie = np.mean(np.abs(siginal[:, :20]), axis=1, keepdims=True)
   power_nosie = amplitude_nosie ** 2
   power_nosie = np.tile(power_nosie, [1, T])  # Repeat the first 11 frames to
the same length as the noisy speech

   # smoothing method
   amplitude_new = np.copy(amplitude)
   k = 1
   for t in range(k, T - k):
       amplitude_new[:, t] = np.mean(amplitude_new[:, t - k:t + k + 1], axis=1)

   power_new = amplitude_new ** 2

   # Ultra-subtractive denoising
   alpha = 4
   gamma = 1

   Power_enhenc = np.power(power_new, gamma) - alpha * np.power(power_nosie,
gamma)
   Power_enhenc = np.power(Power_enhenc, 1 / gamma)

   beta = 0.0001
   mask = (Power_enhenc >= beta * power_nosie) - 0
   Power_enhenc = mask * Power_enhenc + beta * (1 - mask) * power_nosie

   Mag_enhenc = np.sqrt(Power_enhenc)

   Mag_enhenc_new = np.copy(Mag_enhenc)
   # Calculate the maximum noise residual
   maxnr = np.max(np.abs(siginal[:, :11]) - amplitude_nosie, axis=1)

   k = 1
   for t in range(k, T - k):
       index = np.where(Mag_enhenc[:, t] < maxnr)[0]
       temp = np.min(Mag_enhenc[:, t - k:t + k + 1], axis=1)
```

```
        Mag_enhenc_new[index, t] = temp[index]

    # restore the signal
    S_enhec = Mag_enhenc_new * np.exp(1j * phase)
    enhenc = librosa.istft(S_enhec, hop_length=128, win_length=256)
    sf.write("improved.wav", enhenc, fs)

    f = wave.open("improved.wav", "rb")
    parameters = f.getparams()
    print(parameters)  # Print parameters of wav file
    nchannels = parameters[0]  # Get channel parameters
    framerate = parameters[2]  # Get sampling frequency
    nframes = parameters[3]  # Get the number of sampling points
    data2 = f.readframes(nframes)
    f.close()
    wave_data2 = np.fromstring(data2, dtype=np.short)  # convert data to decimal
    wave_data2 = wave_data2 * 1.0 / (max(abs(wave_data2)))
    x2 = np.arange(0, nframes) * (1.0 / framerate)  # Horizontal and vertical 
time coordinates

    ft1 = fft(wave_data1)  # It should be noted that only the data of one 
channel can be operated
    magnitude1 = np.absolute(ft1)
    magnitude1 = magnitude1[0:int(len(magnitude1) / 2) + 1]
    f1 = np.linspace(0, framerate, len(magnitude1))
    xfp1 = 20 * np.log10(np.clip(np.abs(magnitude1), 1e-20, 1e1000))

    ft2 = fft(wave_data2)  # It should be noted that only the data of one 
channel can be operated
    magnitude2 = np.absolute(ft2)
    magnitude2 = magnitude2[0:int(len(magnitude2) / 2) + 1]
    f2 = np.linspace(0, framerate, len(magnitude2))
    xfp2 = 20 * np.log10(np.clip(np.abs(magnitude2), 1e-20, 1e1000))
    xftp3 = 20 * np.log10(np.clip(np.abs(magnitude2), 1e-20, 1e1000))
    #Amplitude enhancement of signals within the highest harmonic frequency 
range
    for i in range(30000,34000):
        temp = xftp3[i]
        xftp3[i] = 1.5*temp    #multiply 1.5 to enchance



    plt.figure()
```

```
    plt.subplot(2, 1, 1)
    plt.plot(x1, wave_data1)
    plt.xlabel("Time/s", loc='right')
    plt.ylabel("Amplitude")
    plt.title("Original Signal (Time Domain)")

    plt.subplot(2, 1, 2)
    plt.plot(x2, wave_data2)
    plt.xlabel("Time/s", loc='right')
    plt.ylabel("Amplitude")
    plt.title("Enhanced Signal (Time Domain)")
    plt.show()

    plt.figure()
    plt.subplot(2, 1, 1)
    plt.ylabel("Amplitude")
    plt.xlabel("Time/s", loc='right')
    plt.title("Spectrogram of the Original Signal")
    plt.specgram(originalData, NFFT=256, Fs=fs)
    plt.subplot(2, 1, 2)
    plt.specgram(enhenc, NFFT=256, Fs=fs)
    plt.ylabel("Amplitude")
    plt.xlabel("Time/s", loc='right')
    plt.title("Spectrogram of the Enhanced Signal")
    plt.show()


    plt.figure()
    plt.subplot(2, 1, 1)
    plt.plot(f1, magnitude1)
    plt.xscale("log")
    plt.xlabel("hz[log scale]", loc='right')
    plt.ylabel("Amplitude")
    plt.title("Original Signal--Amplitude is not Logarithmic (Frequency
Domain)")

    plt.subplot(2, 1, 2)
    plt.plot(f2, magnitude2)
    plt.xscale("log")
    plt.xlabel("hz[log scale]", loc='right')
    plt.ylabel("Amplitude")
```

```
    plt.title("Enhanced Signal--Amplitude is not Logarithmic (Frequency
Domain)")
    plt.show()

    plt.figure()
    plt.plot(f1, xfp1)
    plt.xscale("log")
    plt.xlabel("hz[log scale]")
    plt.ylabel("db")
    plt.title("Part of Original Signal (Frequency Domain)")
    plt.xlim((50, 18000))
    plt.show()

    plt.figure()
    plt.subplot(2, 1, 1)
    plt.plot(f1, xfp1)
    plt.xscale("log")
    plt.xlabel("hz[log scale]", loc='right')
    plt.ylabel("db")
    plt.title("Original Signal--Amplitude is Logarithmic (Frequency Domain)")

    plt.subplot(2, 1, 2)
    plt.plot(f2, xfp2)
    plt.xscale("log")
    plt.xlabel("hz[log scale]", loc='right')
    plt.ylabel("db")
    plt.title("Enhanced Signal--Amplitude is Logarithmic (Frequency Domain)")
    plt.show()

    plt.figure()
    plt.subplot(2, 1, 1)
    plt.plot(f2, xfp2)
    plt.xscale("log")
    plt.xlabel("hz[log scale]", loc='right')
    plt.ylabel("db")
    plt.title("Enhanced Signal--Amplitude is Logarithmic (Frequency Domain")
    plt.grid(True)

    plt.subplot(2, 1, 2)
    plt.plot(f2, xftp3)
    plt.xscale("log")
    plt.xlabel("hz[log scale]", loc='right')
```

```
    plt.ylabel("db")
    plt.title("Amplitude enchance for the higest frequency harmonic")
    plt.grid(True)
    plt.show()
```

voweldetector.py

```
import sys
import wave
import matplotlib.pyplot as plt
import numpy as np
from scipy.signal import find_peaks

class VowelDetector:
    vowel_array = ["A", "E", "I", "O", "U"]
    # vowel_array = ["E"]
    def LoadFile(self, name, name_mod = True):
        if name_mod == True:
            f = wave.open(name + ".wav", "rb")
            parameters = f.getparams()
            nframes = parameters[3]
            data = f.readframes(nframes)
            f.close()
        else:
            f = wave.open(name, "rb")
            parameters = f.getparams()
            nframes = parameters[3]
            data = f.readframes(nframes)
            f.close()
        return data, parameters

    def FFT(self, data, framerate):
        xf = abs(np.fft.rfft(data)) #Forier Transform
        freqs = np.linspace(0, framerate // 2, len(data) // 2 + 1)
        return xf, freqs

    def PatternMatch(self, wav, param, pattern_dic):
        wave_data = np.frombuffer(wav, dtype=np.short)  # convert into
decimalism
        wave_data = wave_data * 1.0 / (max(abs(wave_data)))  # normalization
        framerate = param[2]
        xf, freqs = VowelDetector.FFT(self, wave_data, framerate) #Forier
Transform
```

```
        new_peaks, new_amps = VowelDetector.PeakFinder(self, xf) #Find frequency
pattern
        new_pattern = list(zip(freqs[new_peaks], new_amps['peak_heights']))
        sum_amp = sum(new_amps['peak_heights'])

        plt.plot(freqs, xf)
        plt.plot(freqs[new_peaks], xf[new_peaks], "x")
        plt.vlines(x=freqs[new_peaks], ymin=0, ymax=xf[new_peaks], colors="r")
        plt.ylabel("Amplitude")
        plt.xlabel("Frequence (Hz)")
        plt.title("Test Vowel")
        plt.show()

        # Match new wav file pattern with stored wav file pattern
        match_vowel = "No Match"
        fr_match_ref = 0
        for key in pattern_dic:
            print("Key-----------------------", key)
            fr_match = 0
            key_pattern = pattern_dic[key]
            key_sum_amp = 0.1
            # Sum up all amp in key_pattern
            for pair in key_pattern:
                key_sum_amp += pair[1]

            for key_pair in key_pattern:
                for new_pair in new_pattern:
                    if abs(key_pair[0] - new_pair[0]) < 10: # Compare frequency
peak with a tolerance of 10Hz
                        new_amp_ratio = (new_pair[1] / sum_amp) * 100
                        key_amp_ratio = (key_pair[1] / key_sum_amp) * 100
                        if abs(key_amp_ratio - new_amp_ratio) <= 5: # Matched if
corresponding peak has similar amplitude
                            print("amplitude ratio difference = ",
abs(key_amp_ratio - new_amp_ratio))
                            print("match at freq: ", key_pair[0], new_pair[0])
                            fr_match += 1
            print("fr_match = ", fr_match)
            if fr_match > fr_match_ref:
               match_vowel = key
               fr_match_ref = fr_match
```

```
        return match_vowel

    def PeakFinder(self, xf):
        peaks, amps = find_peaks(xf, height=200, distance = 50, prominence= 200)
#Find peaks of frequencies and corresponding amplitudes
        return peaks, amps

    def PatternFinder(self):
        Dictionary = {}
        #Find pattern for each vowel in vowel_array
        for vowel in VowelDetector.vowel_array:
            #Load wav file, and get frames and parameters
            file, param = VowelDetector.LoadFile(self, vowel, True)
            wave_data = np.frombuffer(file, dtype=np.short)  # convert into
decimalism
            wave_data = wave_data * 1.0 / (max(abs(wave_data)))  # normalization
            framerate = param[2]

            #Generate frequency spectrum and mark patterns
            xf, freqs = VowelDetector.FFT(self, wave_data, framerate) #Forier
Transform
            refine_peaks, refine_amps = VowelDetector.PeakFinder(self, xf) #Find
frequency pattern

            #Plot frequency spectrum and pattern of the vowel
            plt.plot(freqs, xf)
            plt.plot(freqs[refine_peaks], xf[refine_peaks], "x")
            plt.vlines(x=freqs[refine_peaks], ymin=0, ymax=xf[refine_peaks],
colors="r")
            plt.ylabel("Amplitude")
            plt.xlabel("Frequence (Hz)")
            plt.title("Training " + vowel)
            plt.show()

            #Generate vowel's pattern in 2D array format
            Pattern = list(zip(freqs[refine_peaks],
refine_amps['peak_heights']))

            #Store vowel in dictionary as key, and pattern as value
            Dictionary[vowel] = Pattern

        # print(Dictionary)
```

```python
        return Dictionary

def voweldetector(wavfile):
    Detector = VowelDetector()

    Dic = Detector.PatternFinder()
    f, params = Detector.LoadFile(wavfile, False)
    Output = Detector.PatternMatch(f, params, Dic)
    print("MATCHED VOWEL: ", Output)
    return Output

if __name__ == "__main__":

    file_name = sys.argv[1]
    vowel = voweldetector(file_name)
```