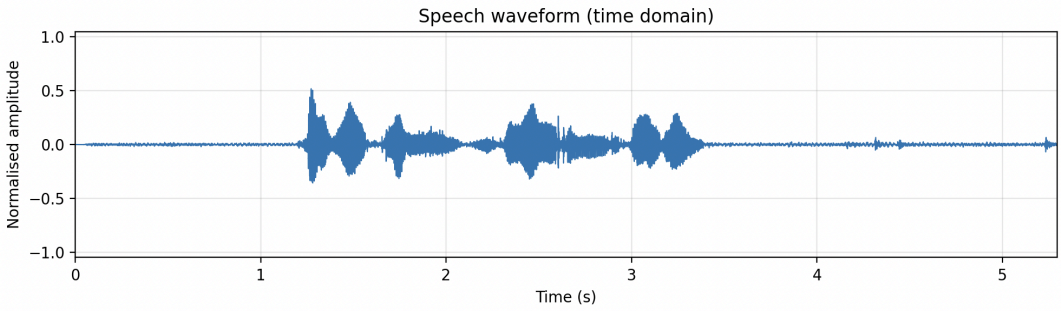
Arda Callialp 2795083C Veer Gokhale 3160362G

Fourier Transform Assignment

Task 1.1: plot voice audio signal in the time domain (linear axis: normalised amplitudes vs time)

The recorded sentence was loaded in Python using scipy.io.wavfile. Since the audio was mono the left channel was used since there wasn't anything on the right channel as its mono. The amplitude values were normalised to the range of −1 to +1 using the integer bit depth. The time axis was generated from the sampling rate (44.1 kHz). The resulting plot shows the speech waveform in the time domain with linear axes (normalised amplitude versus time).

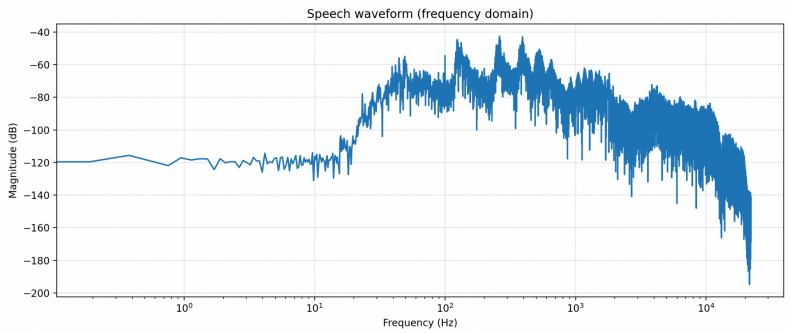
Plot 1: Plotted Audio Signal in the Time Domain



Task 1.2: plot the frequency domain: logarithmic axis for both frequency and amplitude where the amplitude should be in dB with proper axis labels

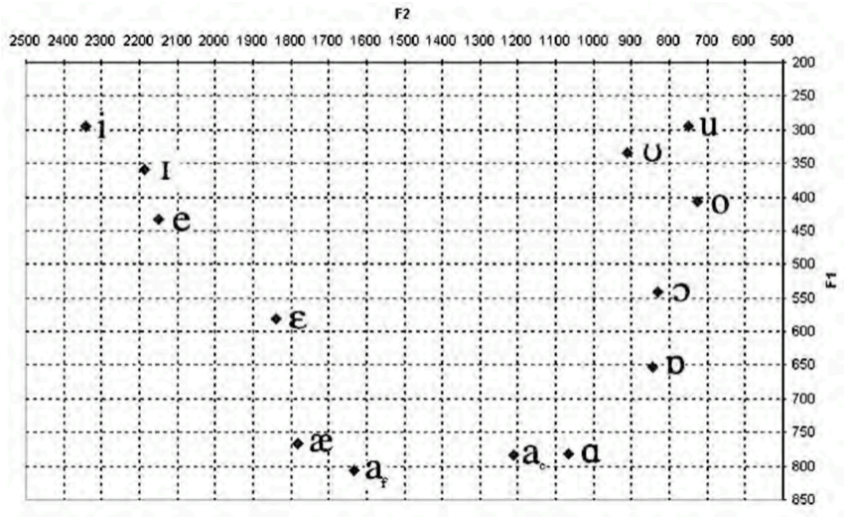
The time domain signal was transformed into the frequency domain using a fast fourier transform (FFT). fd_mag_raw = np.fft.fft(td_normalized) converts the waveform into its frequency components. It produces complex coefficients that represent both the amplitude and phase at each frequency. The corresponding frequency values for each FFT bin were manually calculated using the total number of samples and the sampling frequency, based on $f_k = k \times \frac{F_s}{N}$ for the positive half and $f_k = (k - N) \times \frac{F_s}{N}$ for the negative half. This was implemented as k = np.arange(num_samples) and freq_raw = np.where(k < num_samples/2, k * sample_rate / num_samples, (k - num_samples) * sample_rate / num_samples). Since the input signal is real, its spectrum is symmetric. Therefore, only the positive frequencies were kept: pos_mask = freq_raw >= 0. The magnitude of the complex FFT coefficients was then converted to decibels using the following line: fd_db = 20 * np.log10(2/num_samples * np.abs(fd_mag_raw))[pos_mask]. Where np.abs(fd_mag_raw) gives the magnitude of each complex FFT value. 2/num_samples scales the amplitude for a only positive sided spectrum. The logarithmic function converts the linear magnitude to a logarithmic dB scale and [pos_mask] keeps only the positive half of the spectrum. Finally, the frequency domain plot was created with a logarithmic frequency axis.

Plot 2: Plotted Audio Signal in Frequency Domain



Task 1.3: Mark the peaks in the spectrum which correspond to the fundamental frequencies of the vowels spoken.

Plot 3: Phonetic Pronunciation Vowels and their Frequency Ranges

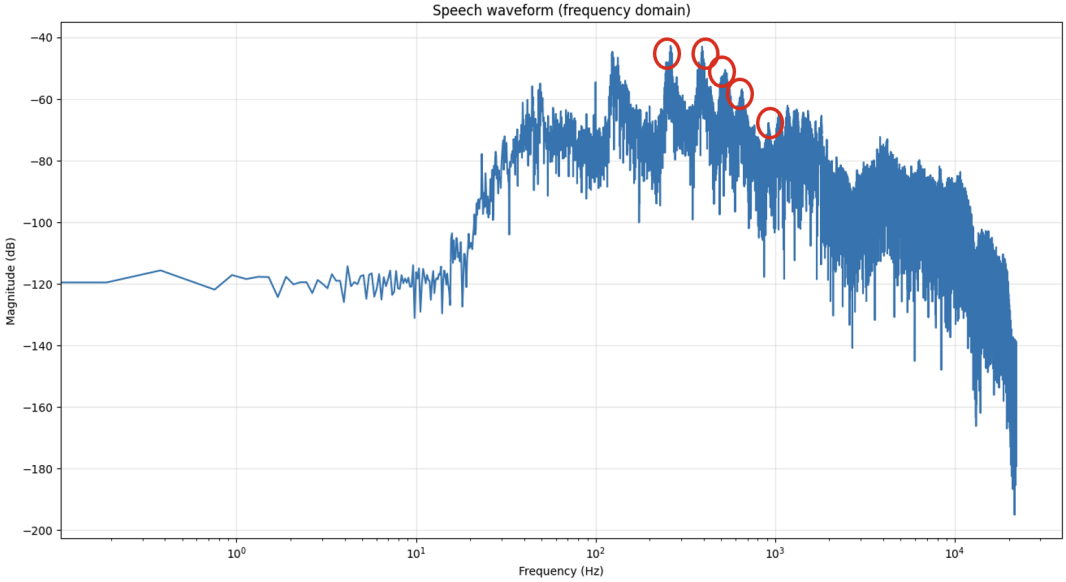


Our sentence was “Our group is Veer and Arda”. According to Wiktionary, these are the phonetic pronunciations for all the words in the sentence.

“Our” : aʊə “Group”: gɹuːp “is”: ɪz “Veer”: vɪɹ “and”: ænd

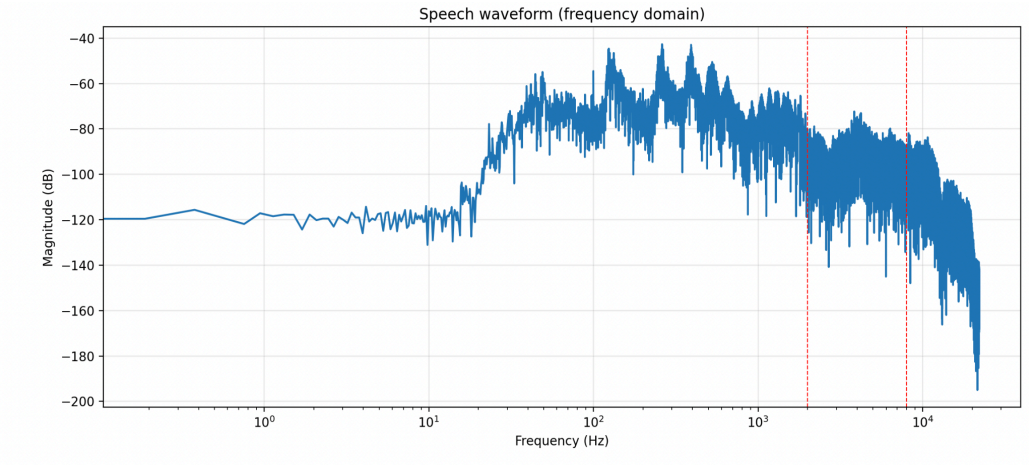
As can be seen on the plot, F1 for all of these vowels are between 250 Hz – 850 Hz. The peaks on the frequency domain plot to which these vowels correspond are shown below.

Plot 4: Frequency domain representation of the speech signal showing vowel formant peaks



Task 1.4: Mark up the frequency range which mainly contains the consonants up to the highest frequencies containing them.

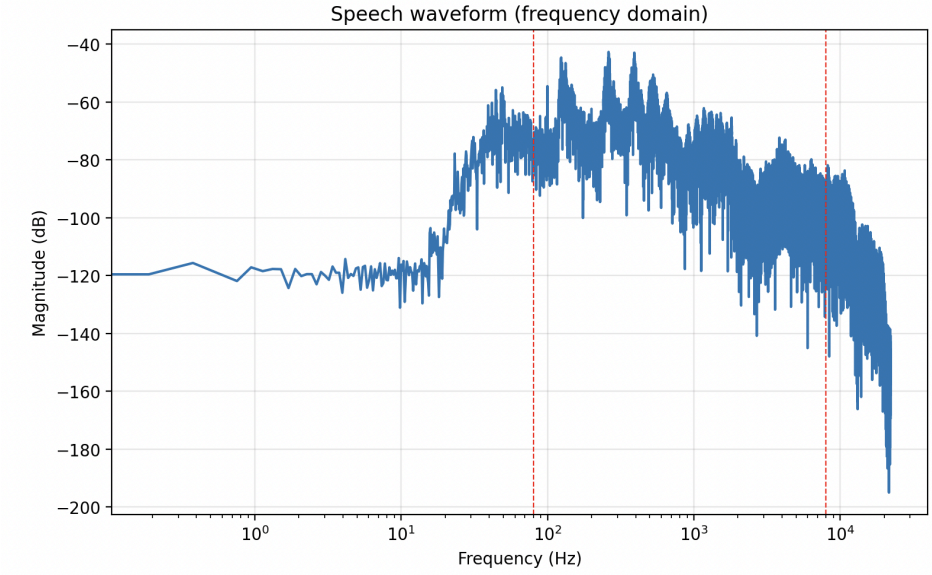
Plot 5: Marked Consonant Range (2000 Hz – 8000 Hz)



The range containing consonants (2000 Hz – 8000 Hz) is marked

Task 1.5: Mark up the whole speech spectrum containing the vowels, consonants and harmonics.

Plot 6: Marked Vocal Spectrum (80 Hz – 8000 Hz)



The range containing the vocal spectrum (80 Hz – 8000 Hz) is marked

Task 2: Low pass filtering and downsampling of the speech signal

In telephony, the standard sampling rate is 8kHz and the standard speech bandwidth is approximately from 300Hz to 3400Hz. To meet this standard the signal was low pass filtered at 3.4kHz to remove higher frequencies and then downsampled to 8kHz, satisfying the Nyquist criterion ($2 \times 3.4kHz = 6.8kHz < 8kHz$).

The downsampling was done using interpolation. After applying the low pass filter a new set of time points were generated at the target sampling rate covering the same duration as the original signal. The amplitudes of the signal at these new time points were obtained by estimating them with NumPy's linear interpolation.

During testing, we also experimented with lower sampling rates such as 4kHz and found that the speech was still recognizable but with significant loss of clarity particularly in high frequency consonants. The 8kHz version represented the best compromise between intelligibility and bandwidth reduction. Therefore we used a sampling rate of 8kHz and bandwidth of 3.4kHz.

Task 3: Touch Tone Detection

In this task, a DTMF (Dual Tone Multi Frequency) detection algorithm was implemented to identify which keys were pressed on a telephone keypad.

First, the DTMF row and column frequencies were defined according to the standard frequency table. Each combination of one row and one column frequency corresponds to a specific button on the phone dial, and these pairs were stored in a DTMF mapping table.

The process begins by loading the WAV file using wavfile.read() and converting the audio data into a NumPy array. The signal was then normalized to the range between -1 and 1 to standardize amplitude values. Next, the button_press_detector() function was used to detect button presses in the time domain. This function looks for the beginning of each button press by checking whether the amplitude of the normalized signal in the time domain exceeds a certain threshold value, then looks 400 samples ahead and uses this as the signal for each segment. To make this work, we checked that none of the samples within those 400 samples also exceeded the threshold value (since when the signal spikes high, it remains there for a couple of samples and we don't want to double count the same button press), and also checked that the average magnitude of values within those 400 samples was above a certain value (since the signal also spikes at the end of a button press, and we don't want to record this as the beginning of a new button press). This approach was able to correctly divide both .wav files into the correct segments.

After the segments were identified, the detect_which_button_is_pressed() function was called. This function takes the list of segments, the sample rate, and the full time domain signal as input. For each segment, it extracts the corresponding end and start indexes of the signal and performs a Fast Fourier Transform (FFT) to convert it from the time domain to the frequency domain. The resulting frequencies and magnitudes are stored in separate arrays. Only positive frequencies and magnitudes are kept, as the negative side of the spectrum is redundant for real valued signals.

Boolean values are then created to separate the frequency spectrum into two regions which are the low frequency row band (600–1000 Hz) and the high frequency column band (1150–1700 Hz). Within each band, the frequency with the highest magnitude is selected. The detected highest magnitude frequencies are then compared with the DTMF row and column frequencies, and the closest matching values are chosen. Finally, the matching (row, column) pair is looked up in the DTMF map to identify the corresponding button that has been pressed. Each detected key is appended to an array of digits, and the function outputs the full phone number detected from the WAV file as a sequence of digits.

The detected digits from 2.wav is 01413302000 which corresponds to University of Glasgow's phone number.

The detected digits from 3.wav is 01413399331 which corresponds to a restaurants phone number Suissi Vegan Kitchen | Glasgow

APPENDIX:

audioplot.py (for the 1st task):

```
from scipy.io import wavfile
import numpy as np
import matplotlib.pyplot as plt

#Task 1.1 - Reading waveform and plotting normalized amplitude vs time

sample_rate, data = wavfile.read("original_speech.wav")

#normalizing the signal to be in between -1 and 1 range
td_normalized = data.astype(np.float32) / np.iinfo(data.dtype).max

# creating time axis
num_samples = len(td_normalized)
t = np.arange(num_samples) / sample_rate #creating a time array for each sample
duration = num_samples / sample_rate # finding the total duration of the recording

#plotting normalised amplitude vs time
plt.figure(figsize=(10,3))
plt.plot(t, td_normalized, linewidth=0.8)
plt.xlabel("Time (s)")
plt.ylabel("Normalised amplitude")
plt.title("Speech waveform (time domain)")
plt.xlim(0, duration)
plt.ylim(-1.05, 1.05)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

#Task 1.2 - Plotting magnitude (dB) vs Frequency (decades)

fd_mag_raw = np.fft.fft(td_normalized)
k = np.arange(num_samples)
freq_raw = np.where(k < num_samples/2, k * sample_rate / num_samples, (k - num_samples) *
sample_rate / num_samples)
pos_mask = freq_raw >= 0
fd_db = 20 * np.log10(2/num_samples*np.abs(fd_mag_raw))[pos_mask]
freq = freq_raw[pos_mask]

plt.plot (freq, fd_db)
plt.xscale('log')
plt.ylabel('Magnitude (dB)')
plt.xlabel('Frequency (Hz)')
plt.title('Speech waveform (frequency domain)')
plt.grid(True, alpha=0.3)
```

```
plt.tight_layout()
plt.show()

#Task 1.3 On document

#Task 1.4 Plotting code

plt.plot (freq, fd_db)
plt.axvline(x=2000, color='red', linestyle='--', linewidth=0.8)
plt.axvline(x=8000, color='red', linestyle='--', linewidth=0.8)
plt.xscale('log')
plt.ylabel('Magnitude (dB)')
plt.xlabel('Frequency (Hz)')
plt.title('Speech waveform (frequency domain)')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

#Task 1.5

plt.plot (freq, fd_db)
plt.axvline(x=80, color='red', linestyle='--', linewidth=0.8)
plt.axvline(x=8000, color='red', linestyle='--', linewidth=0.8)
plt.xscale('log')
plt.ylabel('Magnitude (dB)')
plt.xlabel('Frequency (Hz)')
plt.title('Speech waveform (frequency domain)')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

audio4telephony.py (code for the 2nd task):

```
from scipy.io import wavfile
import numpy as np

# 1st load the wav file
sample_rate, data = wavfile.read("original_speech.wav")
x = data.astype(np.float32) / np.iinfo(data.dtype).max

# 2nd cutoff and target sampling rate
target_sr = 8000 # telecommunication voice standard 8kHz (8000 samples per second) human voice
approx 300-3400 Hz Nyquis criteria: 3400*2=6800 to be safe 8000Hz
cutoff_hz  = 3400.0

num_samples = len(x)
fd_mag_raw = np.fft.fft(x)
k = np.arange(num_samples)
```

```
freq_raw = np.where(k < num_samples/2, k * sample_rate / num_samples, (k - num_samples) *
sample_rate / num_samples)
fd_mag_raw[np.abs(freq_raw) >= cutoff_hz] = 0

# Back to time domain
x_lp = np.fft.ifft(fd_mag_raw).real

# 3rd downsample
N_old = len(x_lp)
t_old = np.arange(N_old) / sample_rate
T_end = (N_old - 1) / sample_rate
N_new = int(round(T_end * target_sr)) + 1
t_new = np.arange(N_new) / target_sr
y = np.interp(t_new, t_old, x_lp)

y_int16 = np.int16(np.clip(y, -1, 1) * 32767)
out_path = "telephone_speech.wav"
wavfile.write(out_path, target_sr, y_int16)
```

tt_detect.py (code for the 3rd task):

```
from scipy.io import wavfile
import numpy as np
import matplotlib.pyplot as plt

#touch tone detection DTMF constants
DTMF_ROWS = np.array([697, 770, 852, 941], dtype=float)
DTMF_COLS = np.array([1209, 1336, 1477, 1633], dtype=float)
DTMF_MAP = {
   (697,1209):'1', (697,1336):'2', (697,1477):'3', (697,1633):'A',
   (770,1209):'4', (770,1336):'5', (770,1477):'6', (770,1633):'B',
   (852,1209):'7', (852,1336):'8', (852,1477):'9', (852,1633):'C',
   (941,1209):'*', (941,1336):'0', (941,1477):'#', (941,1633):'D'
}

#this function shows which sections of the wav file have button press
# and returns an array containing tuples of start and end indexes of the numpy array

def button_press_segment_detector(x, start_threshold = 0.8, sampling_length = 400, noise_threshold =
0.05):
   segments = []
   abs_x = np.abs(x)
   num_peaks = 0
   for i in range(len(abs_x)):
      if abs_x[i] >= start_threshold:
         num_peaks += 1
         if (np.mean(abs_x[i:i+sampling_length]) >= noise_threshold):
            #print(i)
            if (np.max(abs_x[i+1:i+100]) < start_threshold):
```

```
                start = i
                end = i + sampling_length
                segments.append((start, end))
    return segments

#this function analyses each detected button press segment using FFT,
# identifies the two dominant DTMF frequencies (one low, one high)
# matches them to the DTMF frequency table, and returns the detected digits.
def detect_which_button_is_pressed(segments, sample_rate, x):
    digits = []
    for (start, end) in segments:
        segment = x[start:end]
        X = np.fft.fft(segment)
        num_samples = len(segment)
        k = np.arange(num_samples)
        freqs = np.where(k < num_samples/2, k * sample_rate / num_samples, (k - num_samples) *
sample_rate / num_samples)
        magnitudes = np.abs(X)

        #only keeping positive magnitudes and frequencies
        pos_mask = freqs > 0
        freqs = freqs[pos_mask]
        magnitudes = magnitudes[pos_mask]

        #creating boolean values to seperate the low frequencies (rows from DTMF table 600-1000Hz)
        # and high frequencies(columns from DTMF table 1150-1700Hz)
        row_band = (freqs >= 600) & (freqs <= 1000)
        col_band = (freqs >= 1150) & (freqs <= 1700)

        f_row = freqs[row_band][np.argmax(magnitudes[row_band])] #finds the strongest frequency in the
low frequency row range
        f_col = freqs[col_band][np.argmax(magnitudes[col_band])] #finds the strongest frequency in the high
frequency column range

        # matching the detected frequencies both in column and row to the closest DTMF frequencies by
finding the smallest absolute difference
        row = DTMF_ROWS[np.argmin(np.abs(DTMF_ROWS - f_row))]
        col = DTMF_COLS[np.argmin(np.abs(DTMF_COLS - f_col))]

        #look at the DTMF map and get the corresponding key and then append it to the array of detected
digits
        key = DTMF_MAP.get((row, col), '?')
        digits.append(key)

    return digits

if __name__ == "__main__":

    sample_rate, data = wavfile.read("2.wav")
```

```
td_normalized = (data.astype(np.float32)) / np.max(data)

segments = button_press_segment_detector(td_normalized)
digits = detect_which_button_is_pressed(segments, sample_rate, td_normalized)
print("The number in 2.wav is", ''.join(digits))

sample_rate, data = wavfile.read("3.wav")
td_normalized = (data.astype(np.float32)) / np.max(data)

segments = button_press_segment_detector(td_normalized)
digits = detect_which_button_is_pressed(segments, sample_rate, td_normalized)
print("The number in 3.wav is", ''.join(digits))
```