Signals and Systems
Simulation Phase 1
DTMF

Alireza Moradi

April 25, 2020

For this project we had to implement a DTMF decoder using python.

Dual-tone multi-frequency signaling (DTMF) is a telecommunication signaling system using the voice-frequency band over telephone lines between telephone equipment and other communications devices and switching centers.

DTMF Keypad Frequencies

	1209 Hz	**1336 Hz**	**1477 Hz**	**1633 Hz**
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

The **DTMF** telephone keypad is laid out as a matrix of push buttons in which each row represents the low frequency component and each column represents the high frequency component of the DTMF signal.

The commonly used keypad has four rows and three columns, but a fourth column is present for some applications. Pressing a key sends a combination of the row and column frequencies.

For example, the 1 key produces a superimposition of a **697 Hz** low tone and a **1209 Hz** high tone. Initial pushbutton designs employed levers, enabling each button to activate one row and one column contact. The tones are decoded by the switching center to determine the keys pressed by the user.

In this phase we had to decode DTMF signal provided in wav files, Each **200 ms** long with different sampling rates. The result should be saved into a **csv** file and uploaded to Kaggle for scoring.

I only used **numpy** for this project.

[1]:
```
import numpy as np
```

Below is a python dictionary used to match the frequencies to keypad buttons.

[2]:
```
decode = {
    '1': [1209, 697],
    '2': [1336, 697],
```

```
    '3': [1477, 697],
    'A': [1633, 697],

    '4': [1209, 770],
    '5': [1336, 770],
    '6': [1477, 770],
    'B': [1633, 770],

    '7': [1209, 852],
    '8': [1336, 852],
    '9': [1477, 852],
    'C': [1633, 852],

    '*': [1209, 941],
    '0': [1336, 941],
    '#': [1477, 941],
    'D': [1633, 941],
}
```

Below function checks if a signal contains DTMF tones and matches them using the above dictionary. What it does is basically iterate over the dictionary above and checks if any of the signals are present in the sound file, there is an `offset` provided on order to increase accuracy.

Also if there are multiple signals detected there the ones with the highest energy are chosen. At last the detected tone is returned, if there is no tone detected an empty string is returned.

[3]:
```
def check(f, freqs, offset):
    energy_low = 0
    energy_high = 0
    res = ''
    for char, freq in decode.items():
        r0 = range(freq[0] - offset, freq[0] + offset + 1)
        r1 = range(freq[1] - offset, freq[1] + offset + 1)
        for i in range(len(r0)):
            if r0[i] in freqs and r1[i] in freqs and (f[int(r0[i])] >␣
↪energy_low or f[int(r1[i])] > energy_high):
                res = char
                energy_low = f[int(r0[i])]
                energy_high = f[int(r1[i])]

    return res
```

The two inputs of the function below, `signal` and `rate`, Are the wav file loaded into the program and its sampling rate respectively.

The first thing to do is to calculate the **fourier transform** of the input signal, Inorder to match the frequencies in the fourier transform to the original signal easily and also to increase accuracy, the length of the fourier transform is equal to the sampling rate which is the maximum frequency of the signal.

As we don't need the complex part of the fourier transform of the signal we use `np.absolute` take the real part.

Next we shoud calculate a **lower bound** to filter the frequencies below the lower bound and limit the choice.

- The lower bound is calculated using the formula below:
  - `lower_bound = np.average(f) * 11`

The formula is achieved by Trial and error in the test data.

Next we apply the filter to the fourier transform array and find the indices of the points above the lower bound.

Because the signal fourier transform of the signal is real and even, we can use this to check if a detected frequency is just a noise, and we can remove it.

The we call the `check` function with `f`,fourier transform, `freqs`, detected frequencies after applying the filters, and `offset`. We then return the final result.

[4]:
```
def DTMF(signal, rate):

    result = ''

    f = np.fft.fft(signal, rate)

    for i in range(len(f)):
        f[i] = int(np.absolute(f[i]))

    lower_bound = np.average(f) * 11

    freqs = np.argwhere(f > lower_bound)

    offset = 0

    for i in range(freqs.size):
        r = range(freqs[i][0] - offset, freqs[i][0] + offset + 1)
        flag = False
        for j in r:
            if len(f) - j in freqs:
                flag = True
                continue
        if not flag:
            np.delete(freqs, i)

    offset = 2
    result = check(f, freqs, offset)

    return result
```