Tutorial: Note Identification

Time Domain and Frequency Domain

In the real world, audio signals are continuous time signals. To record these signals in a computer, we sample this continuous time signal at a fixed pre-defined sampling frequency. We will denote this sampling frequency by f_s. For example a signal sampled at f_s = 44.1 KHz, captures 44,100 samples of continuous time signal every second at equal intervals of time. The signal thus obtained by sampling the continuous time audio signal is called a discrete time signal. Consider a continuous time signal y(t), the corresponding discrete time signal is given by:

$$y[n] = y(t) * \delta(t - nT)$$

where, $T = \frac{1}{f_s}$ is the time interval between two samples.

Time Domain

For capturing audio, we take samples at regular intervals of time. Such a signal in which the values are recorded against time is known as a time domain signal.

Frequency Domain

As we will study in the next section, every "well behaved" signal can be represented as a weighted sum of Complex Sinusoids of different frequencies where each of these Complex Sinusoids has a different weight. If the signal is represented in terms of weights of different Complex Sinusoids, then the signal is said to be represented in the Frequency Domain.

Time Domain and Frequency Domain representations can be obtained from each other as will be discussed in the next section.

Frequency Domain for Music Signals

As discussed in section 1, music is composed of notes, each of which has a pre-defined frequency. Thus Frequency Domain of a Music Signal will have higher weights for the Complex Sinusoids corresponding to the notes present and lower weights for other signals (Weights will not be zero for other signal because of the finite length of Music Signal as will be described later.) Thus, though we do not get any useful information to identify notes from the Time Domain representation of the signal, the Frequency Domain representation tells us about the Frequencies and hence notes that make up our music signal. Thus all we need to do is to convert the signal from Time Domain to Frequency Domain. Fourier transform will help us in doing this.

Fourier Transform

The Fourier Transform connects the Time Domain world to the Frequency Domain world. Given a continuous Time Domain signal $f(t)$, its representation in Frequency Domain $F(\omega)$ can be obtained from the Fourier Transform of $f(t)$ as given by:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) \, . \, e^{-2i\pi\omega t} dt$$

Equation 1. Fourier Transform

We can recover the Time Domain signal $f(t)$ from $F(\omega)$ using the Inverse Fourier Transform given by:

$$f(t) = \int_{-\infty}^{\infty} F(\omega) \, . \, e^{2i\pi\omega t} d\omega$$

Equation 2. Inverse Fourier Transform

Equation 1 is also called the **Analysis Equation** because it analysis what component frequencies are present in the signal and Equation 2 is also called the **Synthesis Equation** because it synthesizes the Time Domain signal from its Frequency Components.

There are two things to note here:

1. We have defined the Fourier Transform and its Inverse for Continuous Time signals but we need to define it for discrete time Music Signals. We will do this in the next section.
2. The integrals in both cases range from $-\infty$ to ∞, thus we assume that the signals $f(t)$and $F(\omega)$ are defined for all possible values of t and ω respectively. This is however not true in practice. Signals in real world have a finite length. Due to this reason, we do not get a zero weight for all Complex Sinusoids corresponding to the frequencies that are not present in the signal. This introduces noise in the Frequency Domain representation of our signal.

Discrete Fourier Transform

The Discrete Fourier Transform equations are similar to Continuous Time Fourier Transform equation except that they use summation instead of integration. They have been given below:

$$F[\omega] = \sum_{n=-\infty}^{\infty} f(n) \, . \, e^{-2\pi\omega n/k}$$

Equation 3. Discrete Fourier Transform Analysis Equation

$$f[n] = \frac{1}{2\pi} \sum_{\omega=-\infty}^{\infty} F[\omega] \cdot e^{2\pi\omega n/k}$$

Equation 4. Discrete Fourier Transform Synthesis Equation

A finite length Discrete Time signal can be represented as a vector. Let us have an input signal vector of length k. This signal can be represented in a k-dimensional vector space as a linear combination of k basis for that vector space. Discrete Fourier Transform can be seen as a change of basis operation. Thus, we want to represent the same vector but using different basis. Since our signal is k-dimensional we will need k basis. Let v_ω represent the ω^{th} basis vector, then our Time Domain signal $f[n]$ can be synthesized using Equation 5.

$$f[n] = \sum_{\omega=0}^{k-1} F(\omega) \cdot v_\omega[n]$$

Equation 5. Synthesizing Signal Vector from Basis Vectors

The question at this point is as to what should those basis vectors v_ω be.

Let us define ourω^{th} basis vector v_ω as follows:

$$v_\omega[n] = e^{-2\pi\omega n/k}, \ \omega = 0, 1, 2, \dots, k-1$$

Equation 6. Defining Basis Vector

It can be verified that these vectors are orthogonal and hence they form the basis for the k-dimensional vector space. It must be noted that each of these $v_\omega's$ is a Complex Sinusoid with frequency ω. Plugging this value of $v_\omega[n]$ into Equation 5, we obtain Equation 3 which is the Discrete Fourier Transform of our signal $f[n]$.
Since there are k basis vectors, the Discrete Fourier Transform of our original signal vector of length k will have k weights corresponding to each of the k basis vectors. Hence the number of elements in a DFT is equal to the number of elements in the original signal vector.

Ideally a Musical Note is a pure sine wave with a predefined frequency. If we find the DFT for a pure sine wave, we get two peaks (maximum value) in the DFT output. The first of these peaks is at the index corresponding to the frequency of the sine wave. Hence given the DFT of a pure sine wave with unknown frequency we look for peaks and find the index of the first peak (maximum value). Let the index of this peak be i_{max}, then the corresponding frequency in Hertz is given by the following formula:

$$f = \frac{i_{max} \cdot f_s}{l}$$

Equation 7. Calculating Frequency from DFT Output

Here l refers to the length (i.e. the number of samples) of the vector corresponding to the audio signal of the sine wave.

Python Implementation

Understanding the Input

The input given to our program will consist of a series of pure notes, each of some finite duration, separated by silence. The width of silence between any two notes may be different. The duration of each note may be different.

Process Flow

In order to identify the notes that are present in the file, we will implement the following steps in the same sequence:

1. Reading the sound file
2. Detecting silence in the file
3. Detecting the location of notes using data obtained from (2)
4. Calculating the frequency of each detected note by using DFT
5. Matching the calculated frequency to the standard frequencies of notes to identify the note that is being played.

Modules Used

We will use the following three Python modules in our implementation:

1. **wave:** To read audio file
2. **struct:** To decode audio file
3. **numpy:** For all numerical computations. Eg. Fourier Transform

Hence we need three import statements:

```
import wave
import struct
import numpy
```

Reading the Audio File

The following code snippet can be used to read the audio file:

```
1. sound_file = wave.open('/path/to/sound.wav', 'r')
2. file_length = sound_file.getnframes()
3. sound = np.zeros(file_length)

4. for i in range(file_length):
5.     data = sound_file.readframes(1)
6.     data = struct.unpack("<h", data)
7.     sound[i] = int(data[0])

8. sound = np.divide(sound, float(2**15))
```

wave.open(file, mode) - The open function from wave module takes the path to audio file and the mode as parameters. We are using mode = 'r', i.e. reading mode. In this case wave.open returns a Wave_read object.

Wave_read.getnframes() - It returns the number of audio frames contained in the file. Loosely speaking an audio frame is equivalent to one sample of our discrete audio signal.

Wave_read.readframes(i) - It reads the next i audio frames from the file as a string of bytes.

struct.unpack(fmt, string) - The unpack function from the struct module takes the format of byte string to be decoded and the byte string itself as parameters. We are using fmt = "<h" which means that byte string should be interpreted as an integer in little endian format. Hence this function will return the integer corresponding to the specified byte string.

Note that the integers we obtain from struct.unpack using fmt = "<h" are two bytes long. To normalize the data in the range [-1.0, 1.0] we need to divide all the number by 215. This is what line 8 of our snippet does.

At the end of this code snippet, the sound variable will be the vector corresponding to the audio signal where sound[i] contains the ith audio sample.

Detecting Silence

For the detection of silence, one of the approaches that can be implemented involves using a window of some fixed size. Let us assume that our window is 0.05 seconds long. In terms of the number of samples, this window will have the length equal to $0.05 * i_i$. Considering, $i_i = 44{,}100$, this window will of length 2205.

We can slide this window over the input signal and for each position of window, record the sum of squared elements of input falling within the window. This will roughly be equal to the mean square of amplitude of the signal under the window. If this value falls below a particular threshold, then we can classify that input within that window as silence.
Implementing this using a for loop will work, but it may be inefficient for large audio files. Interested reader might also try to implement this using convolve function from numpy.

Detecting Location of Notes

Once the silence has been detected, it is easy to infer the location of notes by keeping track of start and end index of the input falling within the window. Basically everything that is not silence can be considered as a note.

Calculating DFT and Identifying Notes

Once we have the location of each note, we can use Equation 7 to calculate the frequency of the note. To do so, we find i_{iii} by calculating the DFT of the portion of signal at the identified location of the note. The DFT of a signal can be calculated in Python using fft function from numpy.fft module.

numpy.fft.fft(signal) - This function computes the one dimensional Discrete Time Fourier Transform of the specified signal and returns a complex numpy ndarray.

Hint: You might want to use numpy.argsort() function to find i_{iii} from the calculated DFT.

Another thing that Equation 7 requires is l, which is easy of i_i calculate given the start and end indices of the note. Note that Type equation here. will be known to us in advance. All this information combined together will give us the value of f in Equation 7.

This obtained frequency f can be compared to the known frequencies of all notes. The note that has the closest matching frequency will be identified as the current note.