



# Understanding Audio data, Fourier Transform, FFT and Spectrogram features for a Speech Recognition System

An introduction to audio data analysis (sound analysis) using python



Kartik Chaudhary

Jan 19 · 12 min read



## Overview

A huge amount of audio data is being generated every day in almost every organization. Audio data yields substantial strategic insights when it is easily accessible to the data scientists for fuelling AI engines and analytics. Organizations that have already realized the power and importance of the information coming from the audio data are leveraging the AI (Artificial Intelligence) transcribed conversations to improve their staff training, customer services and enhancing overall customer experience.

On the other hand, there are organizations that are not able to put their audio data to

solutions (AI engines) they are going to implement. It is really important to capture all possible data and also in good quality.

This article provides a step-wise guide to start with audio data processing. Though this will help you get started with basic analysis, It is never a bad idea to get a basic understanding of the sound waves and basic signal processing techniques before jumping into this field. You can click here and check out my article on sound waves. That article provides a basic understanding of sound waves and also explains a bit about different audio codecs.

• • •

Before going further, let's list out the content we are going to cover in this article. Let's go through each of the following topics sequentially—

1. *Reading Audio Files*
2. *Fourier Transform (FT)*
3. *Fast Fourier Transform (FFT)*
4. *Spectrogram*
5. *Speech Recognition using Spectrogram Features*
6. *Conclusion*

• • •

## 1. Reading Audio Files

### LIBROSA

*LibROSA* is a python library that has almost every utility you are going to need while working on audio data. This rich library comes up with a large number of different functionalities. Here is a quick light on the features —

3. Feature extraction and Manipulation

4. Time-Frequency conversions

5. Temporal Segmentation

6. Sequential Modeling...etc

As this library is huge, we are not going to talk about all the features it carries. We are just going to use a few common features for our understanding.

Here is how you can install this library real quick —

```
pypi : pip install librosa  
conda : conda install -c conda-forge librosa
```

· · ·

## Loading Audio into Python

Librosa supports lots of audio codecs. Although .wav(lossless) is widely used when audio data analysis is concerned. Once you have successfully installed and imported libROSA in your jupyter notebook. You can read a given audio file by simply passing the file\_path to `librosa.load()` function.

`librosa.load()` —> function returns two things — 1. An array of amplitudes. 2. Sampling rate. The sampling rate refers to ‘sampling frequency’ used while recording the audio file. If you keep the argument `sr = None` , it will load your audio file in its original sampling rate. (**Note:** You can specify your custom sampling rate as per your requirement, libROSA can upsample or downsample the signal for you). Look at the following image —



`sampling_rate = 16k` says that this audio was recorded(sampled) with a sampling frequency of 16k. In other words, while recording this file we were capturing **16000 amplitudes every second**. Thus, If we want to know the **duration** of the audio, we can simply divide the number of samples (amplitudes) by the sampling-rate as shown below —

“Yes you can play the audio inside your jupyter-notebook.”

*I*Python gives us a widget to play audio files through notebook.

## Visualizing Audio

We have got amplitudes and sampling-rate from librosa. We can easily plot these amplitudes with time. *LibROSA* provides a utility function `waveplot()` as shown below



This visualization is called the **time-domain** representation of a given signal. This shows us the loudness (amplitude) of sound wave changing with time. Here **amplitude = 0** represents silence. (From the definition of sound waves — This amplitude is actually the amplitude of air particles which are oscillating because of the pressure change in the atmosphere due to sound).

These amplitudes are **not very informative**, as they only talk about the loudness of audio recording. To better understand the audio signal, it is necessary to transform it into the **frequency-domain**. The **frequency-domain** representation of a signal tells us what different frequencies are present in the signal. **Fourier Transform** is a mathematical concept that can convert a continuous signal from time-domain to frequency-domain. Let's learn more about Fourier Transform.

... . . .

## 2. Fourier Transform (FT)

An audio signal is a complex signal composed of multiple ‘single-frequency sound waves’ which travel together as a disturbance(pressure-change) in the medium. When sound is recorded we only capture the **resultant amplitudes** of those multiple waves.

**Inverse Fourier Transform** is just the opposite of the Fourier Transform. It takes the frequency-domain representation of a given signal as input and does mathematically synthesize the original signal.

Let's see how we can use Fourier transformation to convert our audio signal into its frequency components —

• • •

### 3. Fast Fourier Transform (FFT)

Fast Fourier Transformation(FFT) is a mathematical algorithm that calculates **Discrete Fourier Transform(DFT)** of a given sequence. The only difference between FT(Fourier Transform) and FFT is that FT considers a continuous signal while FFT takes a discrete signal as input. DFT converts a sequence (discrete signal) into its frequency constituents just like FT does for a continuous signal. In our case, we have a sequence of amplitudes that were sampled from a continuous audio signal. DFT or FFT algorithm can convert this time-domain discrete signal into a frequency-domain.



## FFT algorithm overview

### Simple Sine Wave to Understand FFT

To understand the output of FFT, let's create a simple sine wave. The following piece of code creates a sine wave with a **sampling rate = 100, amplitude = 1 and frequency = 3**. Amplitude values are calculated every **1/100th second (sampling rate)** and stored into a list called `y1`. We will pass these discrete amplitude values to calculate DFT of this signal using the FFT algorithm.



If you plot these discrete values(`v1`) keeping sample number on x-axis and amplitude

Now we have a sequence of amplitudes stored in list y1. We will pass this sequence to the FFT algorithm implemented by scipy. This algorithm returns a list yf of ***complex-valued amplitudes of the frequencies*** found in the signal. The first half of this list returns positive-frequency-terms, and the other half returns negative-frequency-terms which are similar to the positive ones. You can pick out any one half and calculate absolute values to represent the frequencies present in the signal. Following function takes samples as input and plots the frequency graph —

In the following graph, we have plotted the frequencies for our sine wave using the above fft\_plot function. You can see this plot clearly shows the single frequency value present in our sine wave, which is 3. Also, it shows amplitude related to this frequency which we kept 1 for our sine wave.

To check out the output of FFT for a signal having **more than one frequency**, Let's create another sine wave. This time we will keep **sampling rate = 100, amplitude = 2 and frequency value = 11**. Following code generates this signal and plots the sine wave —

Generated sine wave looks like the below graph. It would have been smoother if we had increased the sampling rate. We have kept the **sampling rate = 100** because later we are going to add this signal to our old sine wave.

Obviously FFT function will show a single spike with  $frequency = 11$  for this wave also. But we want to see what happens if we add these two signals of the same sampling rate but the different frequency and amplitude values. Here sequence **y3** will represent the resultant signal.

If we plot the signal y3, it looks something like this —

If we pass this sequence (y3) to our fft\_plot function. It generates the following frequency graph for us. It shows two spikes for the two frequencies present in our resultant signal. So the presence of one frequency does not affect the other frequency in the signal. Also, one thing to notice is that the **magnitudes of the frequencies are in line** with our generated sine waves.

## FFT on our Audio signal

Now that we have seen how this FFT algorithm gives us all the frequencies in a given signal. let's try to pass our original audio signal into this function. We are using the same audio clip we loaded earlier into the python with a *sampling rate = 16000*.



Now, look at the following frequency plot. This '**3-second long**' signal is composed of thousands of different frequencies. Magnitudes of frequency values  $> 2000$  are very small as most of these frequencies are probably due to the noise. We are plotting frequencies ranging from 0 to 8kHz because our signal was sampled at 16k sampling rate and according to the Nyquist sampling theorem, it should only posses frequencies  $\leq 8000\text{Hz}$  ( $16000/2$ ).

Strong frequencies are ranging from **0 to 1kHz** only because this audio clip was human speech. We know that in a typical human speech this range of frequencies dominates.

We got frequencies But where is the Time information?

... .

## 4. Spectrogram

### why spectrogram

Suppose you are working on a Speech Recognition task. You have an audio file in which someone is speaking a phrase (for example: How are you). Your recognition system should be able to predict these three words in the same order (1. 'how', 2. 'are', 3. 'you'). If you remember, in the previous exercise we broke our signal into its frequency values which will serve as features for our recognition system. But when we applied FFT to our signal, it gave us only frequency values and we lost the track of time information. Now our system won't be able to tell what was spoken first if we use these frequencies as features. We need to find a different way to calculate features for our system such that it has frequency values along with the time at which they were observed. Here **Spectrograms** come into the picture.

Visual representation of frequencies of a given signal with time is called **Spectrogram**. In a spectrogram representation plot — one axis represents the time, the second axis represents frequencies and the colors represent magnitude (amplitude) of the observed frequency at a particular time. The following screenshot represents the spectrogram of the same audio signal we discussed earlier. Bright colors represent strong frequencies. Similar to earlier FFT plot, smaller frequencies ranging from (0–1kHz) are strong(bright).

## **Creating and Plotting the spectrogram**

Idea is to break the audio signal into smaller frames(windows) and calculate DFT (or FFT) for each window. This way we will be getting frequencies for each window and window number will represent the time. As window 1 comes first, window 2 next...and so on. It's a good practice to keep these windows overlapping otherwise we might lose a few frequencies. Window size depends upon the problem you are solving.

For a typical speech recognition task, a window of **20 to 30ms** long is recommended. A human can't possibly speak more than one phoneme in this time window. So keeping the window this much smaller we won't lose any phoneme while classifying. The frame (window) overlap can vary from 25% to 75% as per your need, generally, it is kept 50% for speech recognition.

In our spectrogram calculation, we will keep the window duration 20ms and an overlap of 50% among the windows. Because our signal is sampled at 16k frequency, each window is going to have  $(16000 * 20 * 0.001) = 320$  amplitudes. For an overlap of 50%, we need to go forward by  $(320/2) = 160$  amplitude values to get to the next window. Thus our stride value is 160.

Have a look at the spectrogram function in the following image. In line-18 we are making a weighting window( Hanning ) and multiplying it with amplitudes before

the DFT algorithm. To learn more about why the weighting window is necessary — [click here](#).

---

*A python function to calculate spectrogram features —*

---

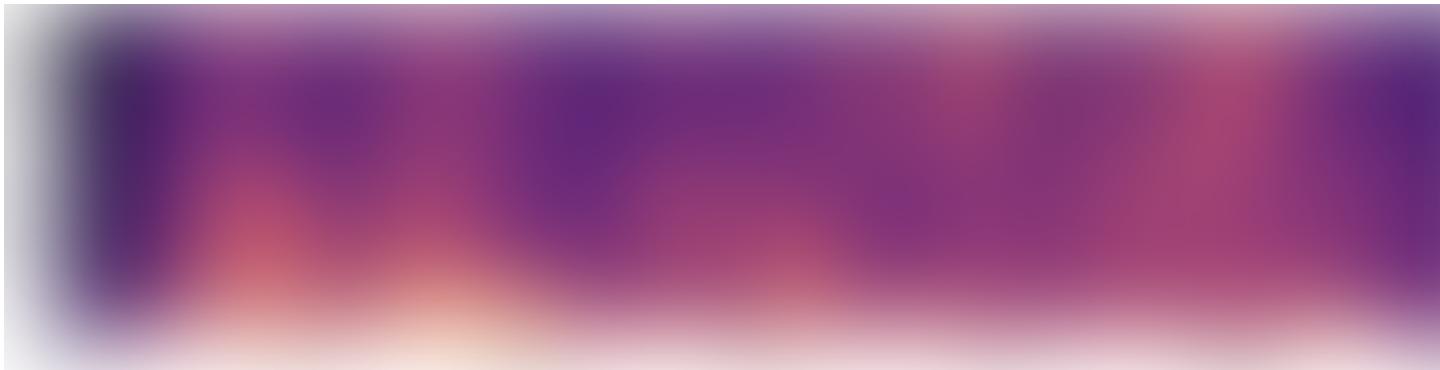
The output of the FFT algorithm is a list of complex numbers ( $size = window\_size / 2$ ) which represent amplitudes of different frequencies within the window. For our window of size 320, we will get a list of 160 amplitudes of **frequency bins** which represent frequencies from **0 Hz — 8kHz** (as our sampling rate is 16k) in our case.

Going forward, Absolute values of those complex-valued amplitudes are calculated and normalized. The resulting 2D matrix is your spectrogram. In this matrix rows and columns represent window frame number and frequency bin while values represent the strength of the frequencies.

• • •

## 5. Speech Recognition using Spectrogram Features

We know how to generate a spectrogram now, which is a 2D matrix representing the frequency magnitudes along with time for a given signal. Now think of this spectrogram as an image. You have converted your audio file into the following image.



This reduces it to an *image classification problem*. This image represents your spoken phrase from left to right in a timely manner. Or consider this as an image where your phrase is written from left to right, and all you need to do is identify those hidden

to try out —

Popular open source datasets –

1. LibriSpeech ASR corpus
2. Common Voice Massively-Multilingual Speech Corpus

Popular choices of deep learning architectures can be understood from the following nice research papers —

1. Wave2Letter (Facebook Research)
2. Deep Speech, Deep Speech 2 and Deep Speech 3 (Baidu Research)
3. Listen, Attend and Spell (Google Brain)
4. JASPER (NVIDIA)

· · ·

## 6. Conclusion

This article shows how to deal with audio data and a few audio analysis techniques from scratch. Also, it gives a starting point for building speech recognition systems. Although, Above research shows very promising results for the recognition systems, still many don't see speech recognition as a solved problem because of the following pitfalls —

1. Speech recognition models presented by the researchers are really big (complex), which makes them hard to train and deploy.
2. These systems don't work well when multiple people are talking.
3. These systems don't work well when the quality of the audio is bad.
4. They are really sensitive to the accent of the speaker thus requires training for every different accent.

model architecture point of view (by presenting a more robust and scalable deep learning architecture).

Thanks for reading, please let me know your comments/feedback.

[Data Science](#)

[Speech Recognition](#)

[Voice Recognition](#)

[Audio](#)

[Voice Assistant](#)

[About](#)   [Help](#)   [Legal](#)