# Spectral Analysis and Compression of Audio

Justin Nichols, Capprin Bass, Victoria Marbois

Mathematical and Computational Modeling—MATH 484

Dr. Mike Nicholas

Due: May 6th, 2020

# Contents

# 1 Introduction

The field of spectral analysis is of particular interest when working with audio. The concept is centered around analyzing the frequency components of a signal, generally from superposition of sinusoidal functions. Our goal was to use spectral analysis to break down an audio input (a song) and remove a select subset of the frequencies to return a compressed file.

# 2 Fourier Transformation

## 2.1 Background

Fourier Transforms translate continuous time-domain signals into the frequency domain. In other words, they translate a signal into its *spectra*, and as a result, they are the primary tool used in spectral analysis. The continuous Fourier transform is given by

$$\mathcal{F}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi x \xi} dx \tag{1}$$

The Fourier transform is a logical step from the Fourier series, a superposition of periodic (sine, cosine) functions. It is useful when performing analysis on *continuous* functions, but not discrete data. Because audio files are collections of sampled amplitudes, a discrete version of the Fourier transform is used: the Discrete Fourier Transform (DFT).

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \tag{2}$$

Given a complex vector of amplitudes $\mathbf{x}$ in the time domain, the DFT produces another complex vector of amplitudes $\mathbf{X}$ in the frequency domain. For the DFT, instead of each index $n$ representing a discrete time, each index $k$ represents a discrete frequency. In this way, the DFT returns the frequency components of a signal and their respective amplitudes. The differences between time-space and frequency-space representations are shown in Figures 1 and 2.

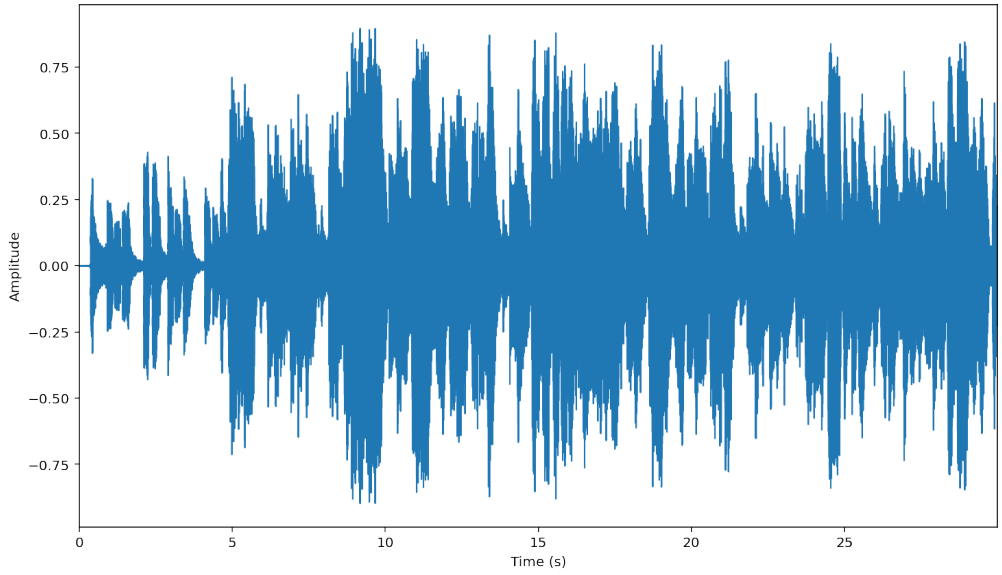


Figure 1: Amplitude vs. Time Plot of Discrete Signal

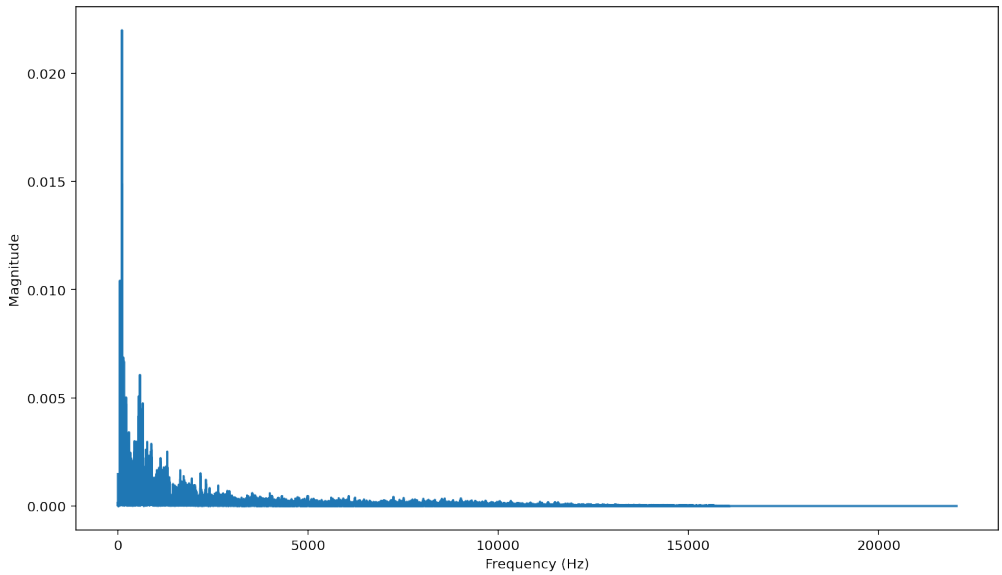


Figure 2: Amplitude vs. Frequency Plot of Discrete Signal

## 2.2 Short Time Fourier Transform

For our purposes, a single Fourier Transform of the input file does not provide enough information in order to construct meaningfully compressed output. In the case of audio, different frequencies ought to be considered "important" at different times over the duration of sound. For example, songs often begin with a heavy bassline, while the majority of the song is probably at a higher frequency. Simply applying a DFT to the whole song would likely remove important frequencies for entire sections.

The Short Time Fourier Transform (STFT) attempts to rectify this issue. Instead of applying a single DFT, we apply multiple DFT's over predefined time "windows." Mathematically, this is expressed as:

$$X_{k,w} = \sum_{n=0}^{N-1} x_n w_{n-k} e^{iwn} \tag{3}$$

Instead of producing a vector $\mathbf{X}$ as is done in Equation 2, the STFT produces a matrix $X$ of amplitudes, valued over some number of time windows. The input is composed of $x_n$ as before, but includes a vector of windows $\mathbf{w}$ informing time intervals.

When implemented, the STFT computes the Discrete Fourier Transform of a collection of time windows within a sound file. In our program, we used this to collect the frequency information of the file every 23 milliseconds. The function also requires an overlap parameter to be set, ensuring that no data is lost as the file is broken up into small intervals.

## 2.3 Spectrogram

Spectrograms are the visual representations of the matrix result of a STFT. By representing the matrix either as a discrete surface or an image, we can visually represent an audio file, and eventually its compressed counterpart. Figure 3 shows an example of the spectrogram, which includes the amplitude, frequency, *and* the time information of the input audio:

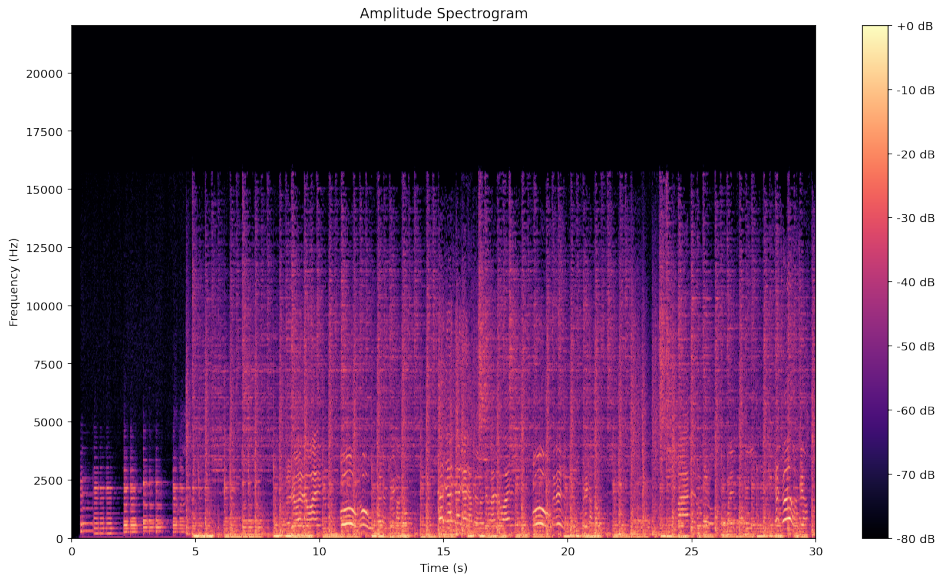


Figure 3: Amplitude Spectrogram

# 3 Mel Scaling

The Mel Scale (short for "melody") is a tool used to transform a regular Hertz-scaled data set into a form that has more discernible differences between frequencies. The motivation is the fact that humans can better perceive low frequencies than high frequencies. The scaling is accomplished by taking an elementwise log of the STFT matrix, although the form of the scale varies. A general representation is below, with $m$ as the mel-scaled frequency, $a, b, c$ as constants, and $X_{k,w}$ as the $(k, w)$th component of the STFT matrix.

$$m = a log_b(\frac{X_{k,w}}{c} + 1) \tag{4}$$

After mel scaling, the distance between low frequencies is increased while the distance between high frequencies is decreased. This step is performed within our program in order to magnify "important frequencies" in the input file by stretching the low frequency data and compressing the high frequency data. You can see this within Figure 4 below. Notice that high-frequency rows from Figure 3 are less represented, and that low-frequency rows have been expanded.

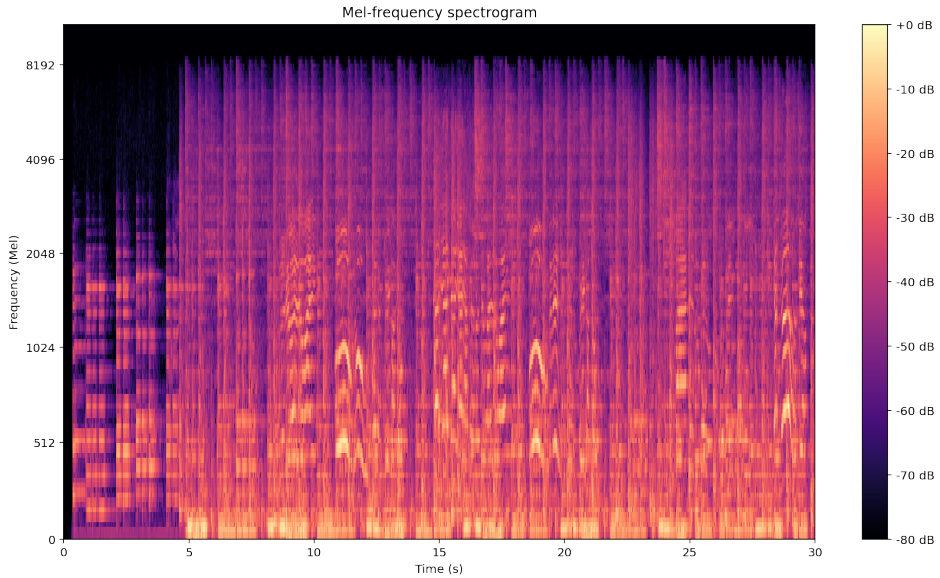


Figure 4: Mel Spectrogram

# 4 Compression

To compress the audio data and amount of frequencies, our program performed a variation of Principal Component Analysis. The importance of each frequency within its corresponding time window was calculated via a metric called explained variance. The *decompose* function from the libROSA library intakes a parameter called n_components, which is based off of a percentage chosen by the user. The program keeps this percentage of the frequencies sorted by importance and discards the rest. This portion of the frequencies is what builds the new compressed audio file.

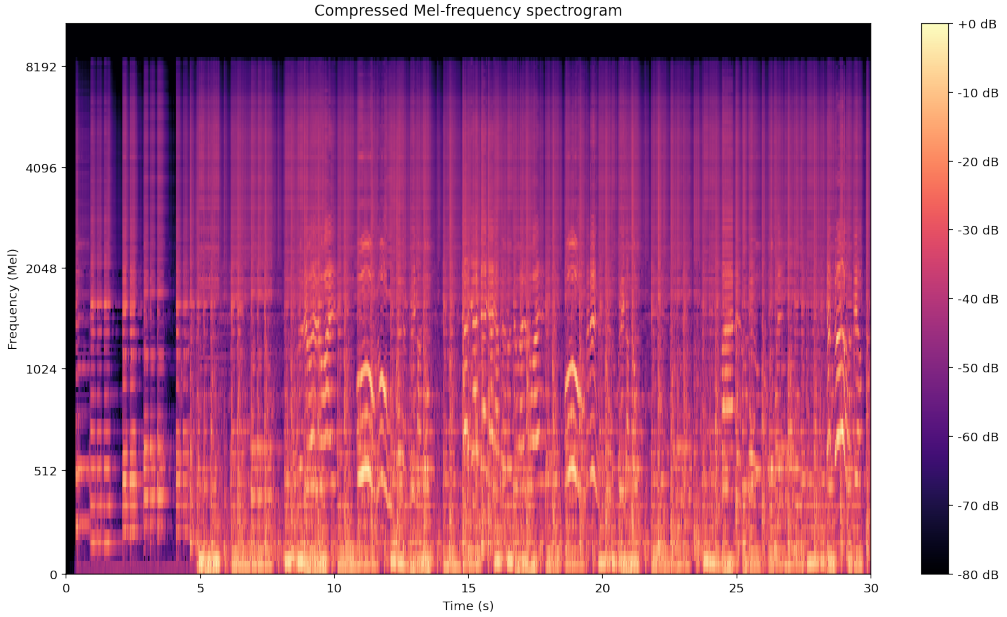


Figure 5: Compressed Spectrogram

The figures seen above and below were generated with the parameter n_components set to 40%. The noticeable visual differences between the original amplitude spectrogram and the compressed spectrogram are a loss of "texture", smoother tops of waves (seen around the 8192 Hz tick mark in Figure 5), and filled dark space between 0 and 5 seconds. Intuitively, if n_components was set to a higher number we would see less change between the two spectrograms after compression, and if n_components was lower we would see even more change occurring in the data.

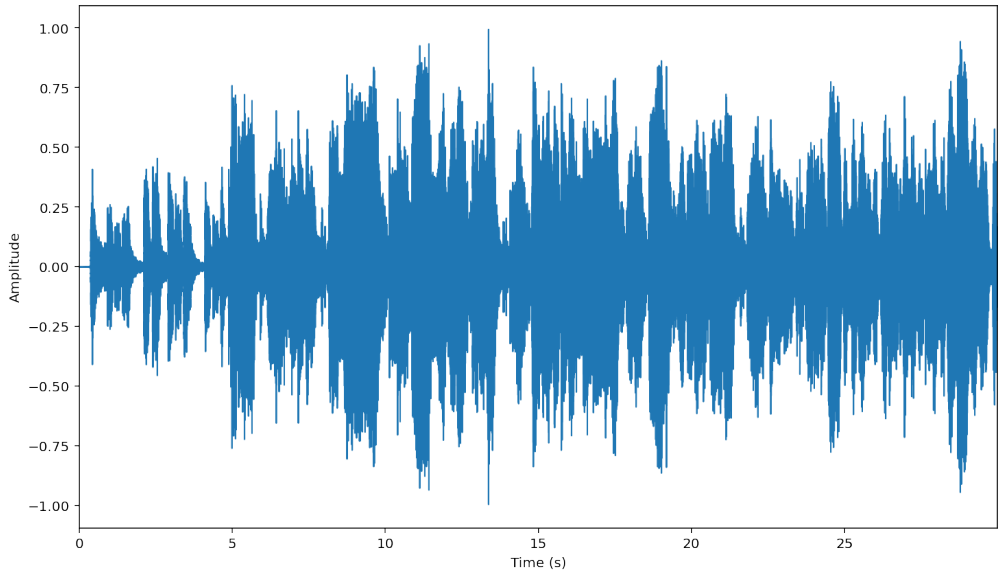


Figure 6: Amplitude vs. Time Plot of Compressed Audio

# 5 Analysis of Results

By utilizing the tools within the fields of spectral analysis and audio compression our program was able to successfully modify an original sound file into a more condensed format. Through the manipulation of a compression parameter, we were able to remove the less important frequencies present within the audio and returned a noticeably different version. In the example of maintaining only 40% of the original frequencies, we saw less texturing in the middle region of the original sound file. The compression also introduced unwanted static during the initial five seconds of the sound file. We suspect this is related to the removal of waveforms that were cancelling other frequencies during that time period.

Overall we feel as though we achieved the original goal of manipulating audio files by compressing the amount of frequencies discovered via Fourier Transform techniques. Future work could involve exploring why we saw the first five seconds of audio become populated with static after compression. Audio analysis is also a great jumping-off point into image processing, which uses many of the tools described previously from spectral analysis. This project helped each of us learn new skills regarding Fourier Transforms, file compression, and audio processing. By removing frequencies, the enjoyment of good tunes was ruined with high pitched racket. On the contrary, the excitement we had seeing math rip songs apart was ***glorious***.

# Appendix

## List of Figures



## A Code

Load File and Plot Sound Signal

```
import librosa as lib
from librosa import display
import matplotlib.pyplot as plt
from IPython.display import Audio

# load file
filename = "sound/louie.mp3"
samples, sampling_rate = lib.core.load(filename, duration=30, sr=None)

duration = len(samples)/sampling_rate
print(filename, "loaded; sampled at", sampling_rate, "Hz with a total length of", duration, "s.")

# output amplitude vs. time
plt.figure()
lib.display.waveplot(y=samples, sr=sampling_rate)
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.show()

# play sound (only runs in Jupyter classic)
Audio(filename)
```

Fourier Transform Performed

```
import numpy as np
import scipy as sp

# use fft for frequencies, magnitudes
n = len(samples)
T = 1/sampling_rate
freqs = np.linspace(0.0, 1.0/(2.0*T), n//2) #possible frequency space
freq_mag = sp.fft.fft(samples) #magnitudes (occurrences) of frequencies (complex)
real_freq_mag = 2.0/n*np.abs(freq_mag[:n//2]) #take positive-frequency terms

# output
plt.figure()
plt.plot(freqs,real_freq_mag)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.show()
```

Create Amplitude Spectrogram

```
# do stft

# window info
window_len_ms = 23
window_size = int(window_len_ms * sampling_rate * 0.001)
hop_len = int(window_size/4) #overlap, to be consistent between stft and specshow

# compute amplitude spectrogram; convert to decibels
S = np.abs(lib.core.stft(samples, n_fft=window_size, hop_length=hop_len, window='hann'))
S_db = lib.amplitude_to_db(S, ref=np.max) #convert to db

# output
lib.display.specshow(S_db, hop_length=hop_len, x_axis='time', y_axis='hz', sr=sampling_rate)
plt.colorbar(format='%+2.0f dB')
plt.title('Amplitude Spectrogram')
```

```
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.tight_layout()
plt.show()
```

Create Mel Spectrogram

```
# mel spectrogram
M = lib.feature.melspectrogram(samples, sr=sampling_rate, n_fft=window_size, hop_length=hop_len, window='hann')
M_db = lib.power_to_db(M, ref=np.max)

# output
lib.display.specshow(M_db, x_axis='time', y_axis='mel', hop_length=hop_len, sr=sampling_rate)
plt.colorbar(format='%+2.0f dB')
plt.title('Mel-frequency spectrogram')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Mel)')
plt.tight_layout()
plt.show()
```

Create Compressed Mel Spectrogram

```
# principal components (compression)
percent_comp = 0.4
n_comp = int(percent_comp*np.size(M,0))
print('Compressing file to', n_comp, 'samples out of', np.size(M,0))

comps, acts = lib.decompose.decompose(M, n_components=n_comp)
M_comp = comps.dot(acts)

M_comp_db = lib.power_to_db(M_comp, ref=np.max)

# output
display.specshow(M_comp_db, x_axis='time', y_axis='mel', sr=sampling_rate, hop_length=hop_len)
plt.colorbar(format='%+2.0f dB')
plt.title('Compressed Mel-frequency spectrogram')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Mel)')
plt.tight_layout()
plt.show()
```

Create Compressed Sound Signal

```
# file output
import soundfile as sf

S_comp = lib.feature.inverse.mel_to_stft(M_comp, sr=sampling_rate, n_fft=window_size)
y_comp = lib.griffinlim(S_comp)
sf.write('sound/pc40_louie30.wav', y_comp, sampling_rate)
sf.write('sound/louie30.wav', samples, sampling_rate)

# compressed amplitude plot

plt.figure()
display.waveplot(y=y_comp, sr=sampling_rate)
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.show()
```

# B Sources


Chaudhary, Kartik. "Understanding Audio Data, Fourier Transform, FFT, Spectrogram and Speech Recognition." Medium, Towards Data Science, 10 Mar. 2020, towardsdatascience.com/understanding-audio-data-fourier-transform-fft-spectrogram-and-speech-recognition-a4072d228520.

Gartzman, Dalya. "Getting to Know the Mel Spectrogram." Medium, Towards Data Science, 16 Jan. 2020, towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0.

LibROSA Python Library. https://librosa.github.io/librosa/.