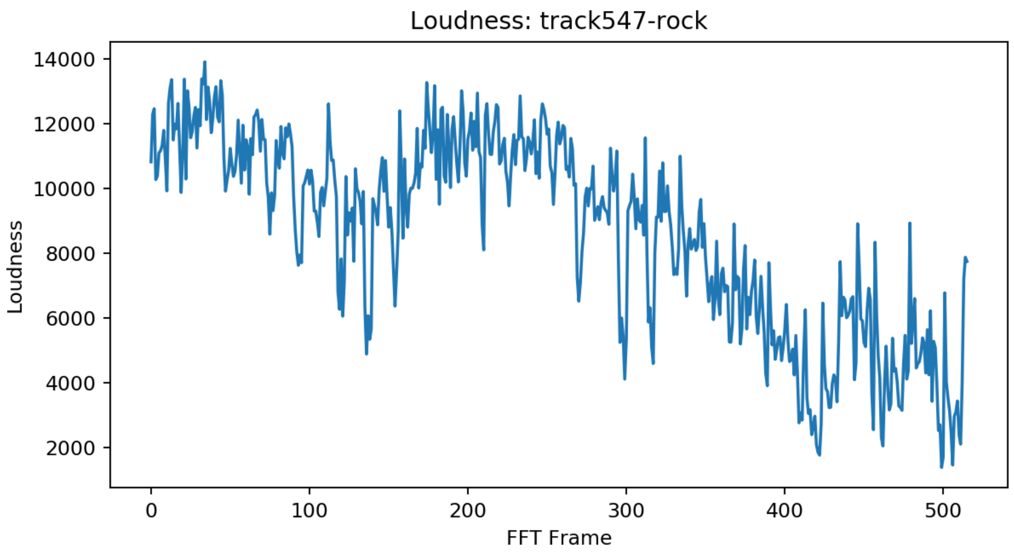
ECEN 4532 - Lab 1: Sound Processing in Python

Andrew Teta

January 26, 2019



Contents

Introduction

In this discussion, we will be exploring some techniques recently developed in the audio industry to organize, search, and classify large music collections. We are developing some statistical methods, along with frequency analysis to automatically characterize songs and genres.

0.1 Background

Fundamentally, we begin with a '.wav' file. This is one variety of many signal encodings, generated by sampling a physical sound wave using a microphone. The microphone detects pressure differences within a medium (sound) and converts them into electrical voltages. As the voltages are converted into a digital representation, they are quantized at a sampling frequency (fs). Furthermore, the Nyquist sampling theorem dictates that for a signal to be represented accurately in quantized form, it must be sampled at twice the rate of the highest frequency component. For example, a dolphin can only hear sound in the range $7kHz - 120kHz$, so if we want to record the sound that a dolphin would be able to hear, we have to sample at a rate $fs = 240kHz$ or higher.

In this lab, we consider a small collection of music, organized by genre, in '.wav' format. Each song is sampled at $fs = 11,025Hz$. Narrowing our data set even further, we will extract a 24 second sample from the middle of each song. We will then implement a short-time Fourier transform (STFT), which divides the sample into $N = 512$ samples, or $46ms$ and call each 46 ms interval a 'frame'. The STFT is a good way to obtain frequency spectrum data, while maintaining a level of time-domain relevance. For the purposes of this lab, the frames will be non-overlapping.

1 Time-domain Analysis

We begin by extracting a $24s$ sample from the middle of each song using the python function `scipy.io.wavfile.read`. See section 5.3 for Python implementation.

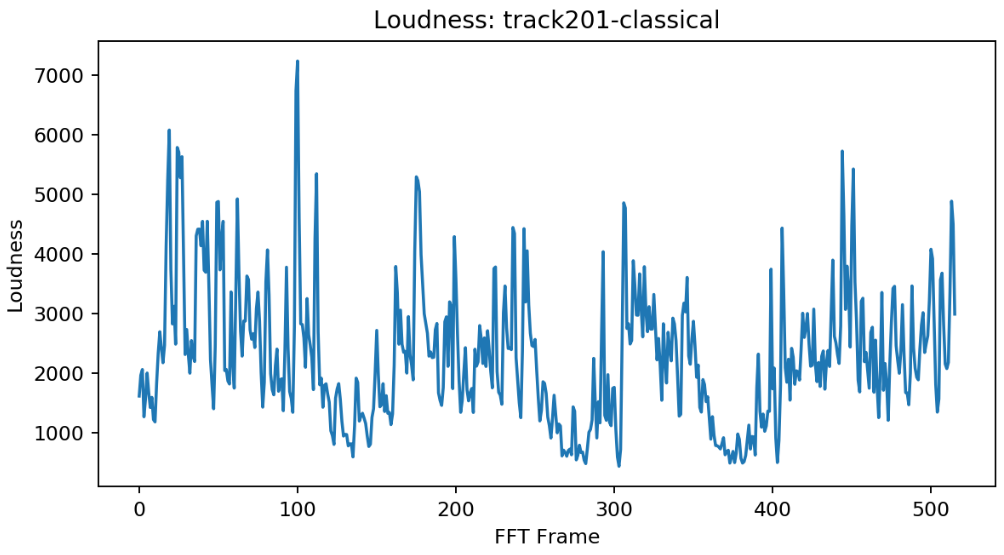
1.1 Loudness

To get a sense of 'loudness', we will compute the standard deviation over a 'frame' of size $N = 512$ defined as:

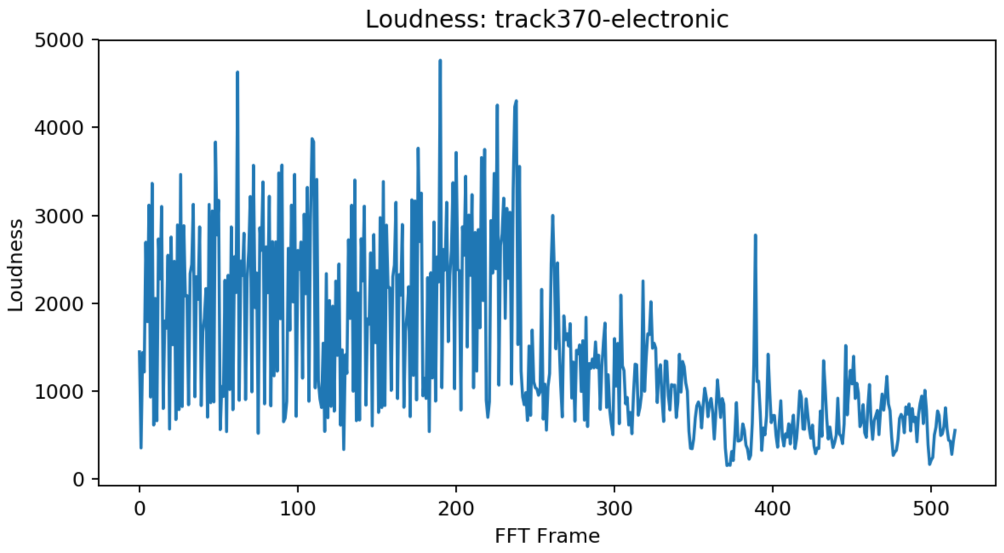
$$\sigma(n) = \sqrt{\frac{1}{N}\sum_{m=0}^{N-1}[x(nN+m) - E[x_n]]^2} \qquad \text{with} \qquad E[x_n] = \frac{1}{N}\sum_{m=0}^{N-1} x(nN+m) \tag{1}$$

See section 5.3.1 for Python implementation.

Results The output of the loudness calculation for one song of each genre is shown in the figures below.

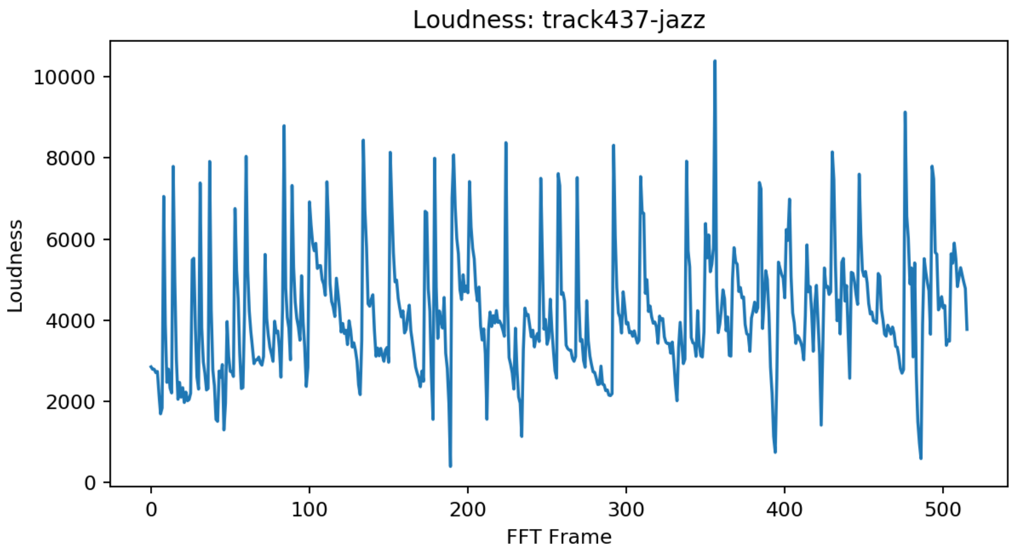


(a) 'track201-classical.wav'

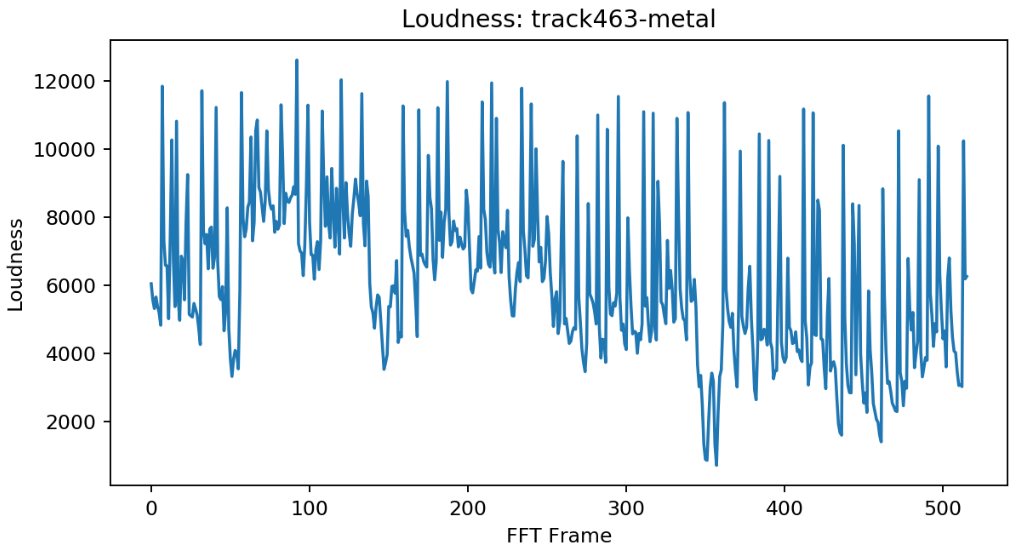


(b) 'track370-electronic.wav'

Figure 1: Loudness vs. frame

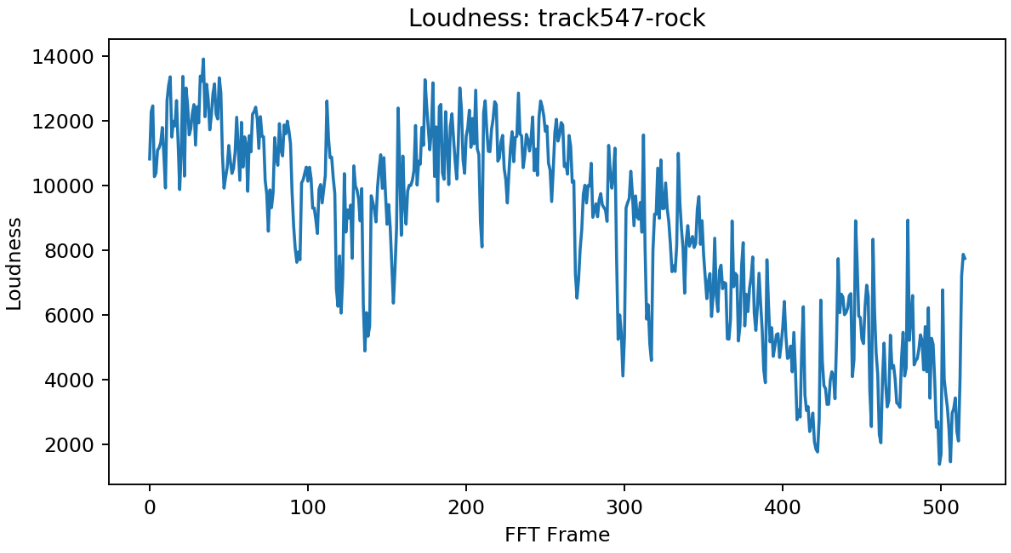


(c) 'track437-jazz.wav'

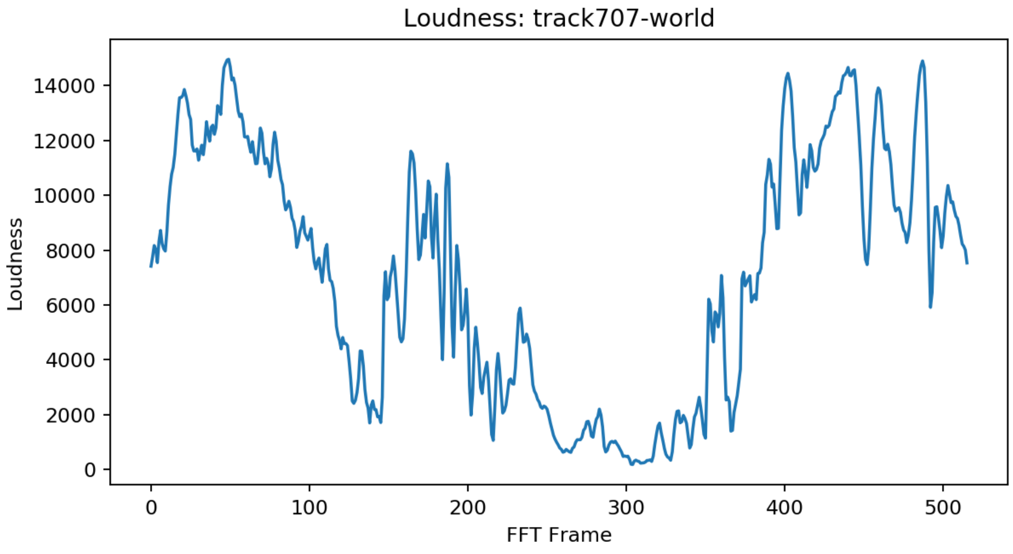


(d) 'track463-metal.wav'

Figure 1: Loudness vs. frame



(e) 'track547-rock.wav'



(f) 'track707-world.wav'

Figure 1: Loudness vs. frame

Comments While each of the plots in figure 1 has its own characteristics, loudness is not the best tool for characterizing genres. Figure 1b shows the uniformity of the track over time and contrasts with figure 1f as well as figure 1a, however it would be hard to distinguish between figure 1b and figure 1d. However, these plots do give a good sense of the progression of each song during the sample considered.

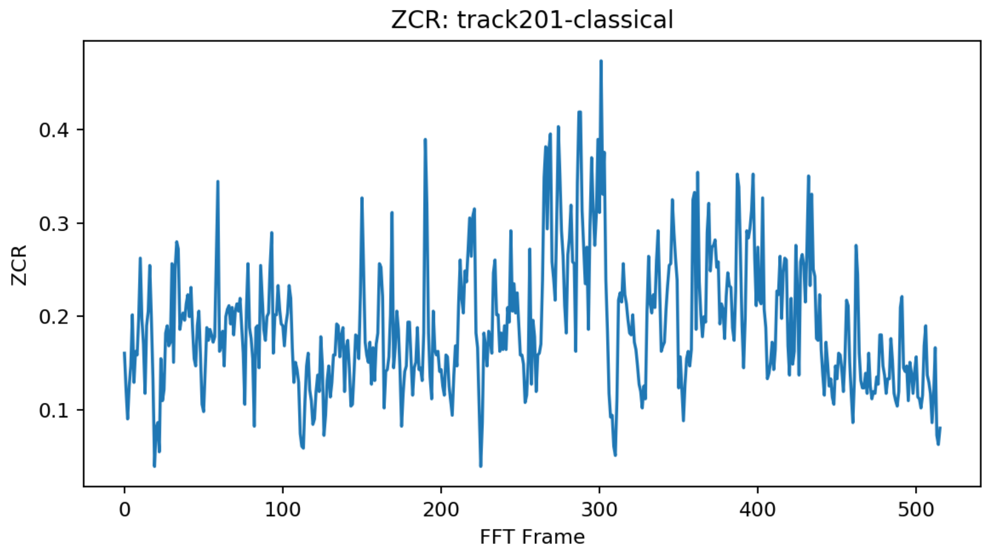
1.2 Zero Crossing Rate

The zero crossing rate (ZCR) is the average number of times the audio signal crosses the zero amplitude line per unit time. It is related to pitch height and correlated to the noise in the signal. For this lab, ZCR is defined as:

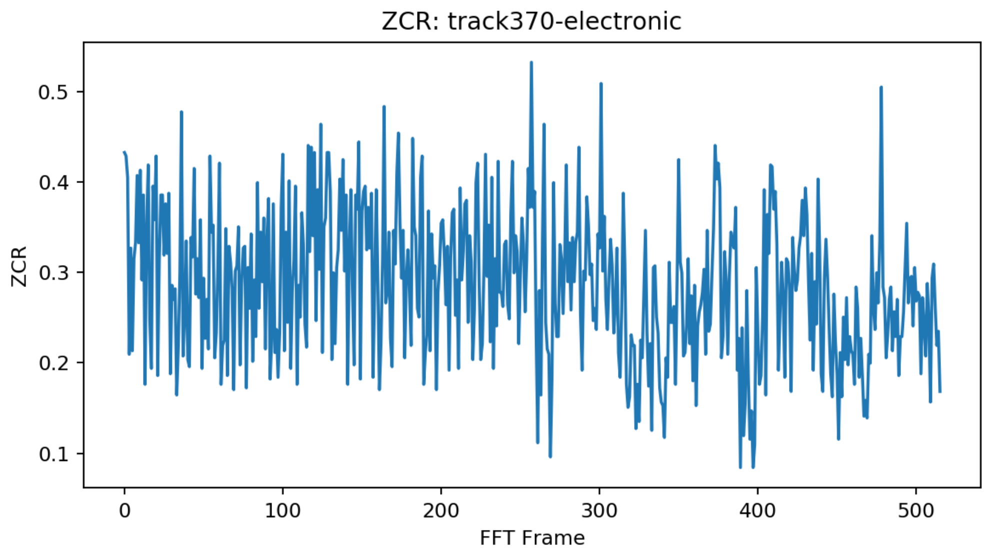
$$ZCR(n) = \frac{1}{N-1} \sum_{m=1}^{N-1} \frac{1}{2} |sgn(x(nN+m)) - sgn(x(nN+m-1))|. \tag{2}$$

See section 5.3.2 for Python implementation.

Results Again, we display the output of computed ZCR for one track of each genre in the figures below.

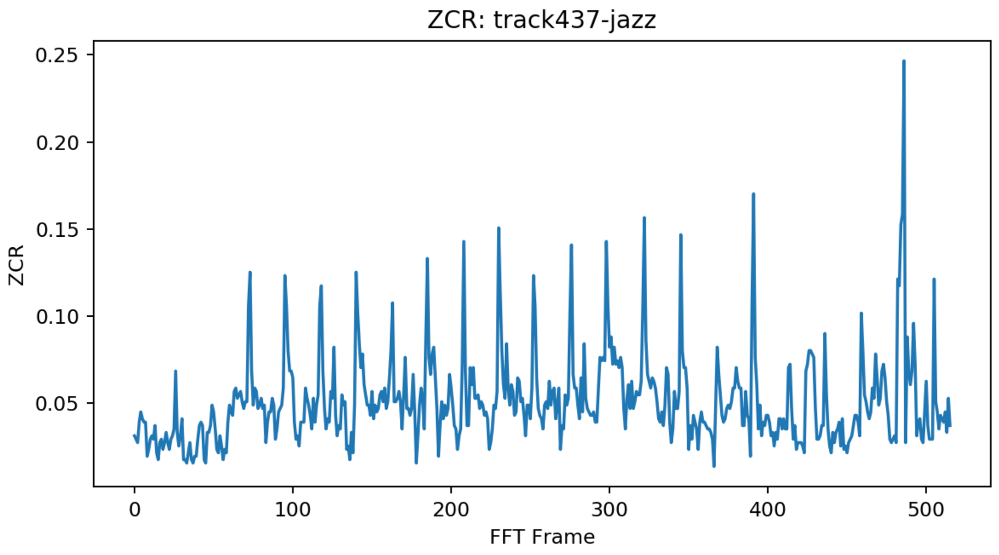


(a) 'track201-classical.wav'

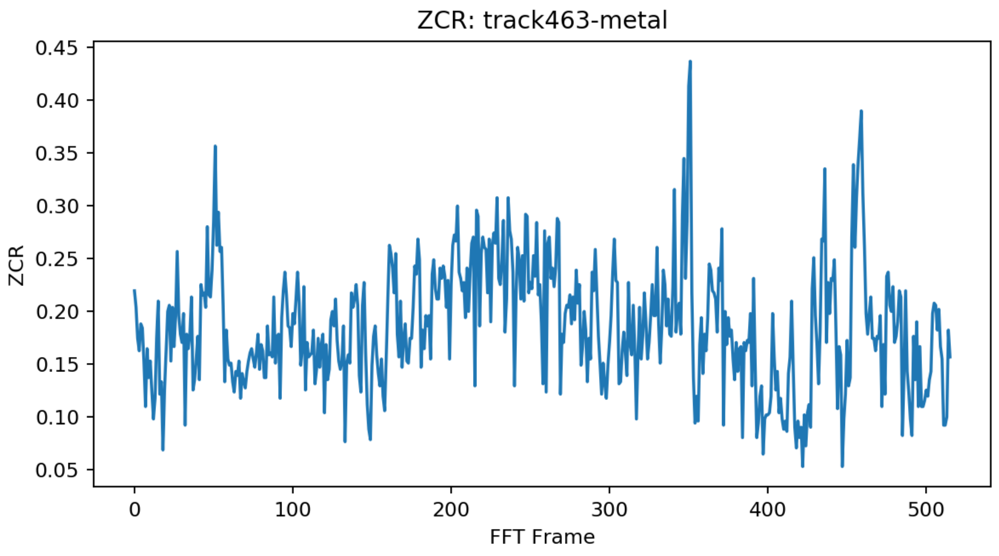


(b) 'track370-electronic.wav'

Figure 2: ZCR vs. frame

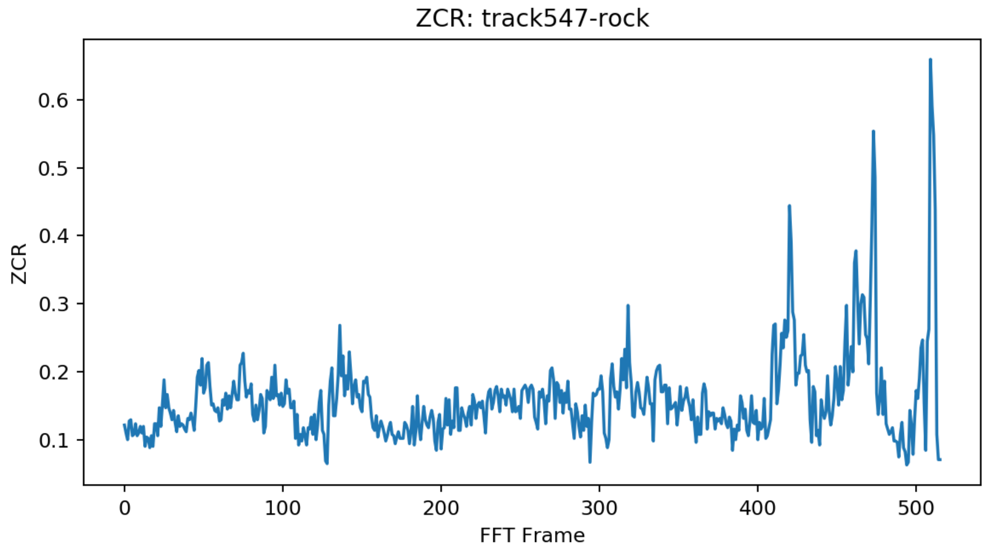


(c) 'track437-jazz.wav'

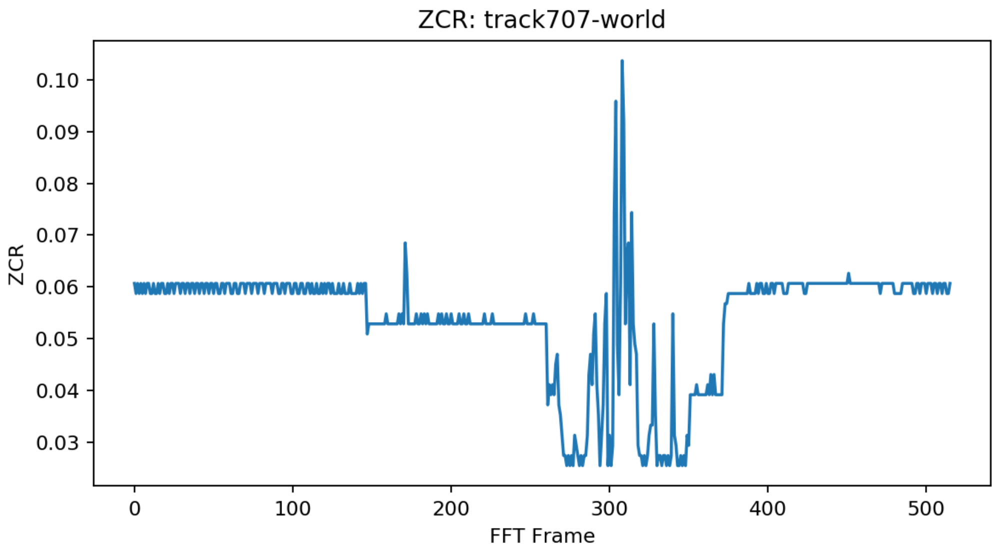


(d) 'track463-metal.wav'

Figure 2: ZCR vs. frame



(e) 'track547-rock.wav'



(f) 'track707-world.wav'

Figure 2: ZCR vs. frame

Comments From the plots in figure 2 we can definitely see some variability between genres. For example, the world track in 2f is easily distinguishable, however most of the other tracks are similar enough that this is also not likely a good measurement for classifying genre. This can be supported by the similarity between jazz (2c) and rock (2e).

2 Spectral (Frequency) Analysis

Sound, by nature is made up of many different frequencies or notes of different pitch. Thus it is natural to analyze audio in the spectral (frequency) domain.

Method As discussed in the introduction, we will accomplish our spectral analysis using a STFT, which decomposes short-time frames of a signal into discrete frequency bins, maintaining some time-domain relevance within the spectral domain. Then, we will perform some statistical analysis on the signal in an attempt to characterize some songs.

Rather than directly splitting each of our $N = 512$ frames, we will convolve (a fancy term for multiplication in the frequency domain) a taper window, w, with the signal and take the Fourier transform (FT). Since the signal is real, and we are concerned only with the magnitude, we will compute the FT of only half of the frequency spectrum to arrive at $N/2 = 256$ frequency bins for each of our $N = 512$ frames. Finally, for the purposes of this analysis, we are interested in the power spectrum of the original song, so we take the magnitude squared (i.e. $|X_n(k)|^2$, as a function of the frame index n and frequency index k). See section 5.4 for Python implementation.

To give some motivation for the taper window, w, we will derive the theoretical discrete-time Fourier transform (DTFT) for a given signal $x[n]$ where,

$$\begin{aligned} x[n] &= 1, & -N/2 \le n \le N/2 \\ x[n] &= 0, & \text{else} \end{aligned}$$

Recall the definition of the DTFT.

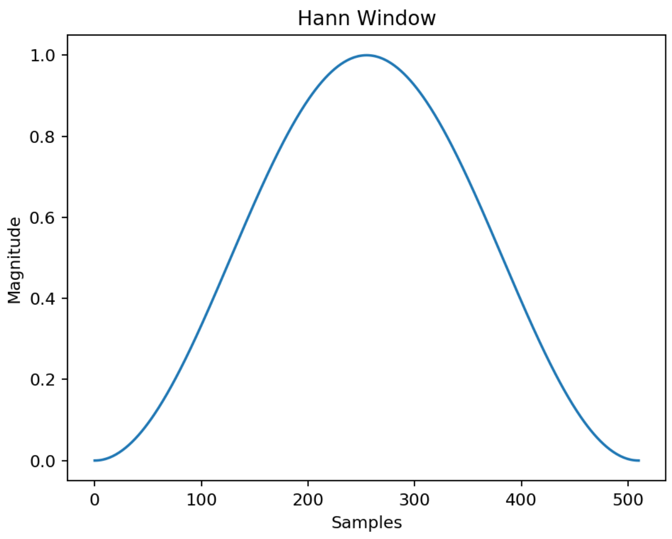
$$X(\omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}. \tag{3}$$

Since $x[n] \neq 0$ over only a subset of values in the DTFT range, equation (3) becomes,

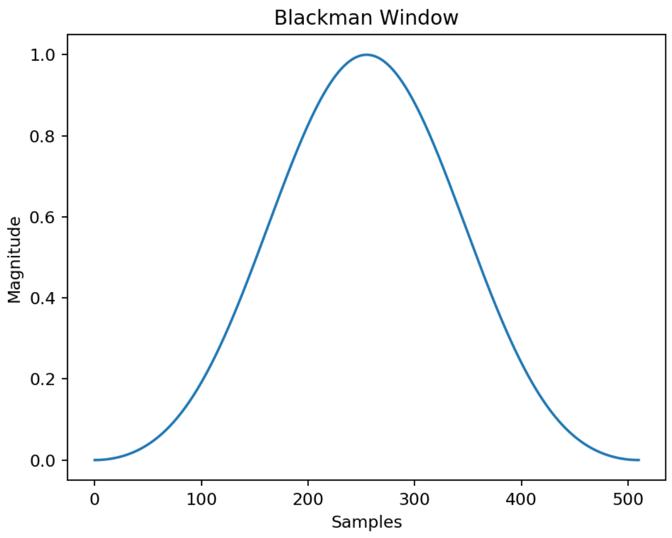
$$\begin{aligned} X(\omega) &= \sum_{n=-N/2}^{N/2} x[n]e^{-j\omega n} \\ &= \sum_{n=-N/2}^{N/2} e^{-j\omega n} \end{aligned}$$

2.1 Spectrogram

As promised, we will be implementing a STFT. In Python, we will use the `scipy.signal.spectrogram` function to automate this. However, we first need a window. For this, we will use the library function, `scipy.signal.get_window` which allows us to quickly generate taper windows of the same size as our STFT frames. There are many choices of windows, but we will look at the 'Hann' and 'Blackman' windows. See figure 3.



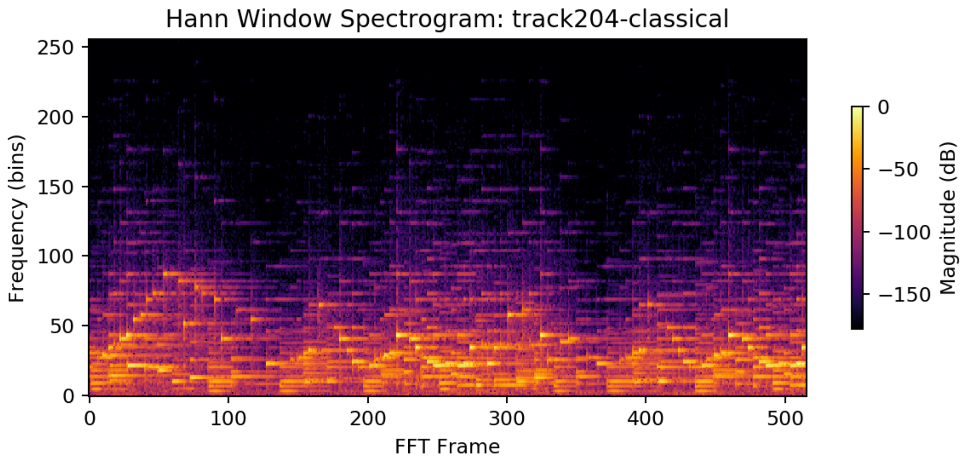
(a) Hann window



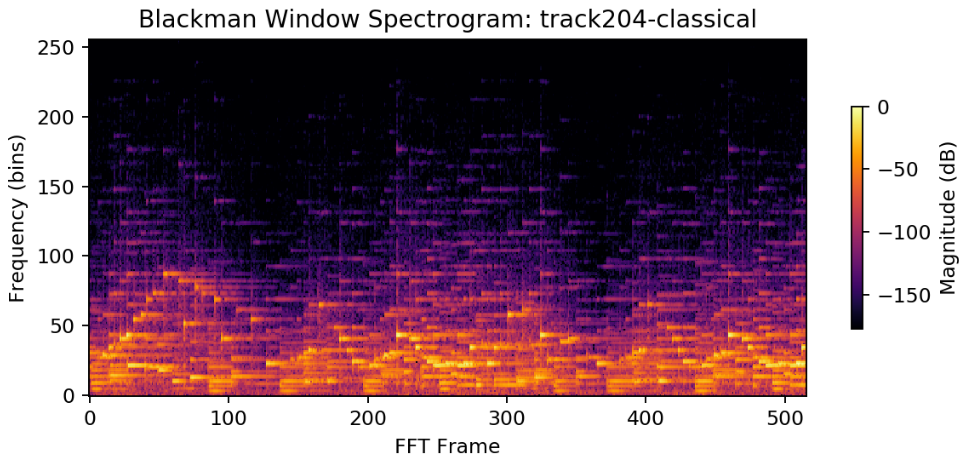
(b) Blackman window

Figure 3: Windows

As you can see, the Hann window (3a) tapers a little less steeply than the Blackman window (3b).

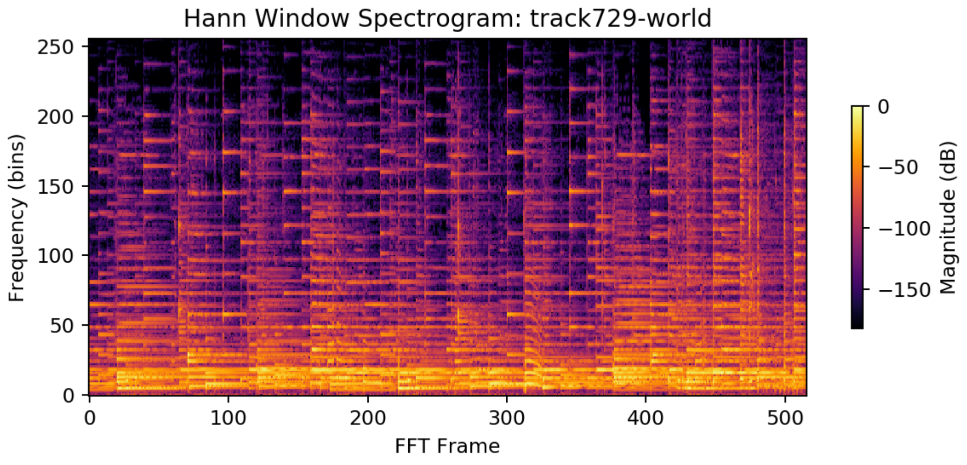


(a) Spectrogram of ’track204-classical’ using a Hann window

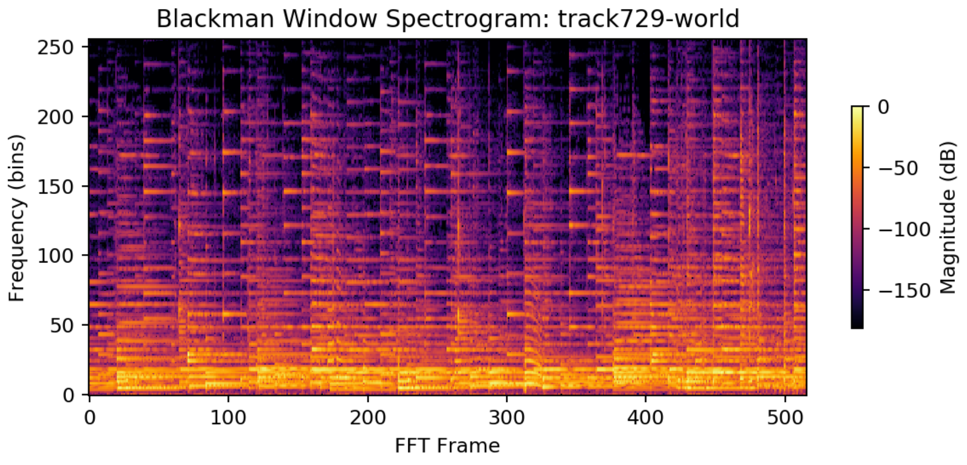


(b) Spectrogram of ’track204-classical’ using a Blackman window

Figure 4: Hann vs. Blackman windows for a classical music track



(a) Spectrogram of 'track729-world' using a Hann window



(b) Spectrogram of 'track729-world' using a Blackman window

Figure 5: Hann vs. Blackman windows for a classical music track

See section 5.4.1

Comments As far as I can tell, the two spectrograms produced by these two windows in figures 4 and 5 are identical. We will proceed with analysis using only the Blackman window.

2.2 Spectral Centroid and Spread

We wish to find the 'center of mass' of each frame in a spectrogram and will call this the spectral centroid. The centroid can be used to quantify sound sharpness or brightness. Additionally, we would like to find the spectral spread, or width of the spectrum around the centroid. Thus, we can then compare tone-like and noise-like sounds. For these concepts, we will treat the normalized magnitude of a spectral coefficient as if it were a 'probability' of that particular frequency. Then, for frame n, we have the 'probability' of frequency k

$$P_n(k) = \frac{|X_n(k)|}{\sum_{l=0}^{K} |X_n(l)|} \tag{4}$$

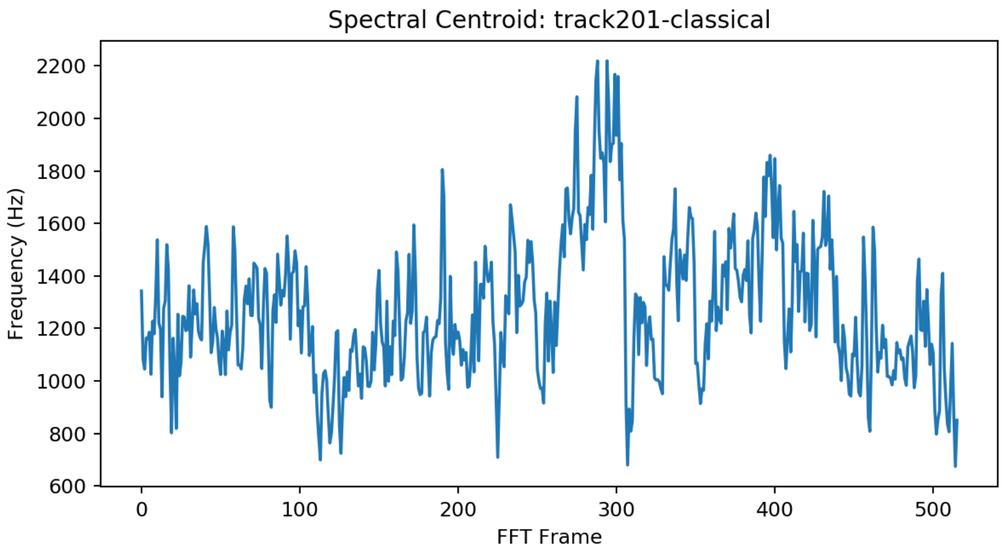
Which we can then use to define the spectral centroid as

$$\mu_n = \sum_{k=0}^{K} k P_n(k) \tag{5}$$

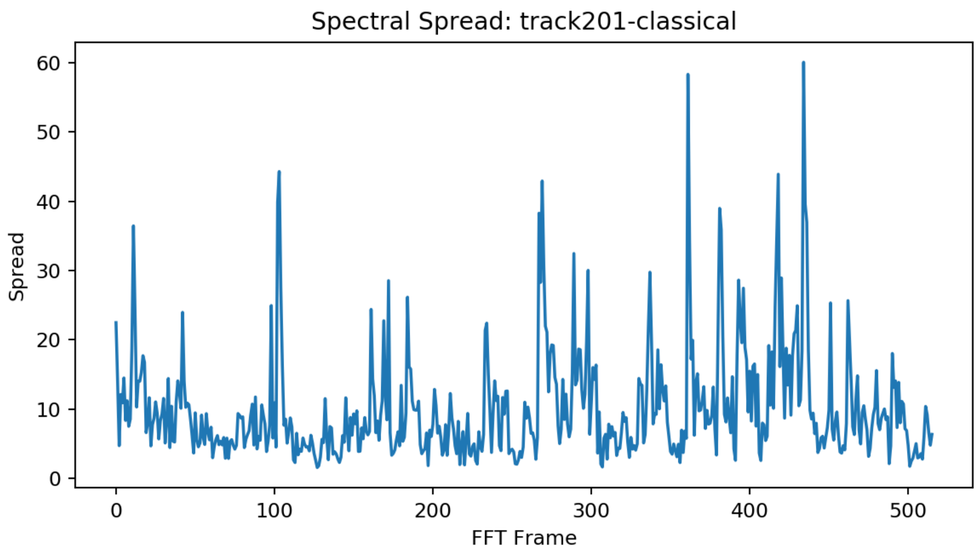
And the spectral spread for frame n is the standard deviation given by

$$\sigma_n = \sqrt{\sum_{k=0}^{K} [k - \mu_n]^2 P_n(k)} \tag{6}$$

The spectral centroid and spread were calculated for a classical, jazz, and metal song and a time-domain plot is shown in figure 6. See section 5.4.2 for Python implementation.

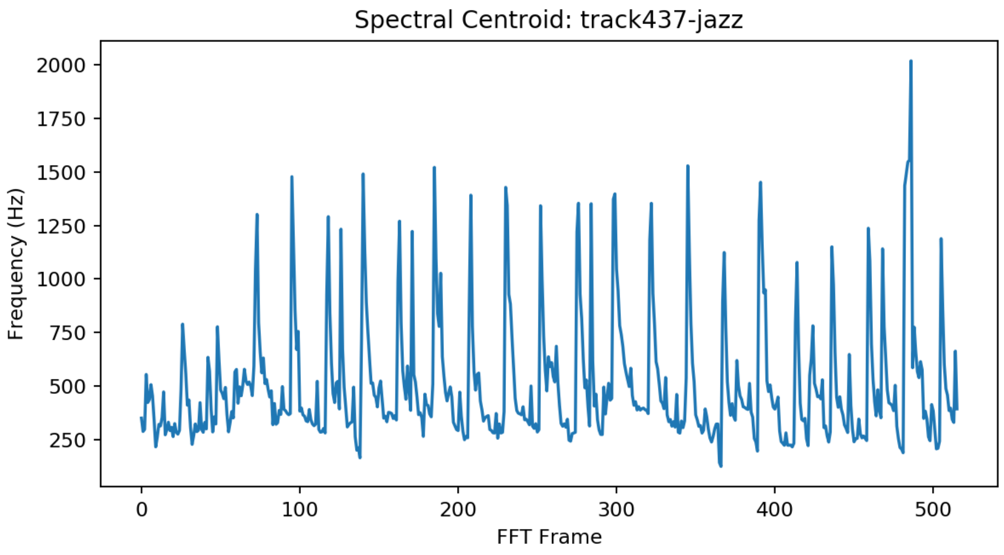


(a) Spectral centroid of 'track201-classical' as a function of frame number.

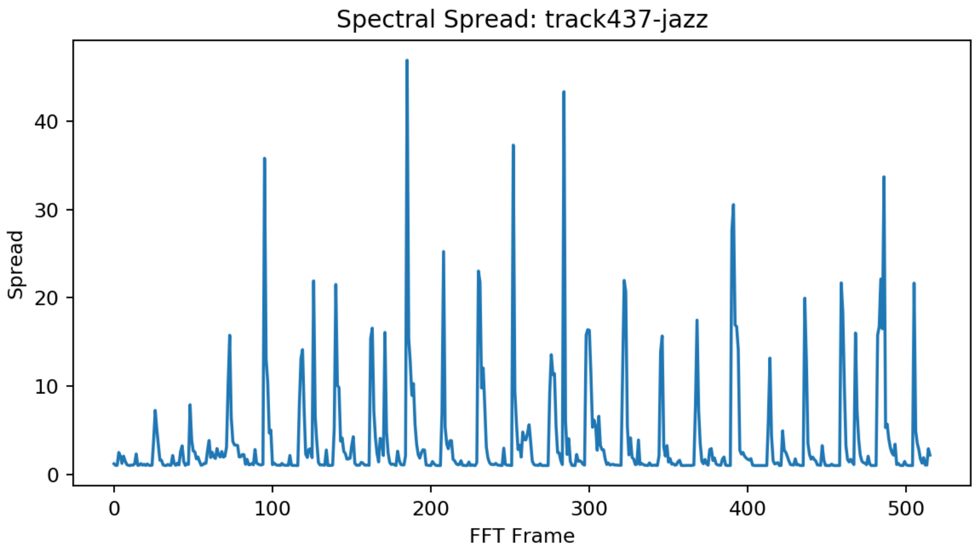


(b) Spectral spread of 'track201-classical' as a function of frame number.

Figure 6: Centroid and spread

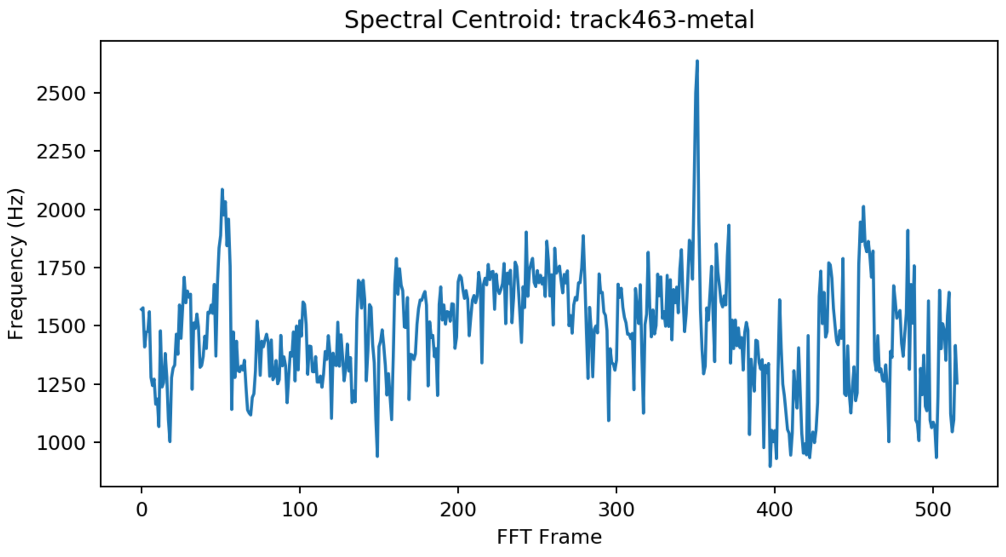


(c) Spectral centroid of 'track437-jazz' as a function of frame number.

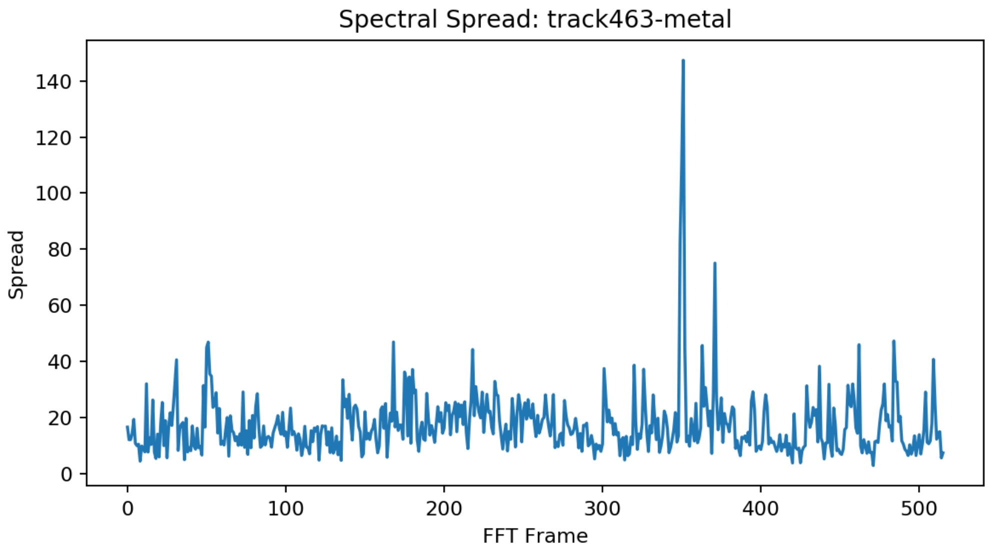


(d) Spectral spread of 'track437-jazz' as a function of frame number.

Figure 6: Centroid and spread



(e) Spectral centroid of 'track463-metal' as a function of frame number.



(f) Spectral spread of 'track463-metal' as a function of frame number.

Figure 6: Centroid and spread

Comments The centroids of this classical track, shown in figure 6a the most statistically significant frequency changes from frame to frame, sometimes drastically. This is most likely due to the violin's sharp transitions between tones. Then in the spread (figure 6b), we can see that there is relatively high spread for many frames, indicating the high tones in the song and their inherent noise (potentially from under-sampling).

Then, we see in the analysis of a jazz song (figures 6c and 6d) There is a contrast between the percussion and electric keyboard, which is apparent in the consistently alternating peaks and valleys. The spread shows that many of the peaks are also high-frequency and noisy.

In the metal track, we see a relatively uniform centroid (figure 6e) and spread (figure 6f) across frames. Notice also the different scales on the vertical axis of these plots, indicating that the metal track is fairly noisy compared to the other two songs, although it appears to have smaller magnitude on first glance. We see a large spike around frame 350 corresponding to a high-pitch yell from the vocalist. In the centroid plot, this can be seen as a spike in frequency with an associated spread due to the timbre of his voice and noise in the signal.

Comparing these between genres, notice that the classical and jazz have some similarities, which would make sense based on the similarity of genres, however we can see a contrast in the centroids, due to the electronic composition of the jazz track as opposed to the natural, dissonant composition of the classical. Furthermore, the small spread shown in figure 6d indicates that notes were generated electronically. The metal track stands out in a couple regards. Figure 6e is vastly different and shows the dense composition in the track, with the centroid in each frame nearing the middle of the frequency spectrum and indicating that the range of frequencies is large at any given point in the song.

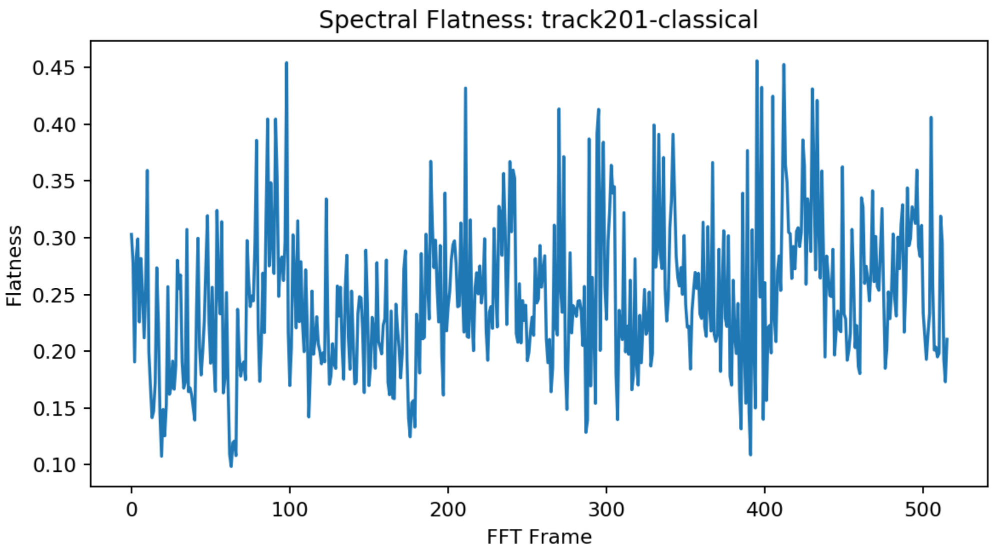
2.3 Spectral Flatness

Spectral flatness is the ratio between the geometric and arithmetic means of the magnitude of the Fourier transform,

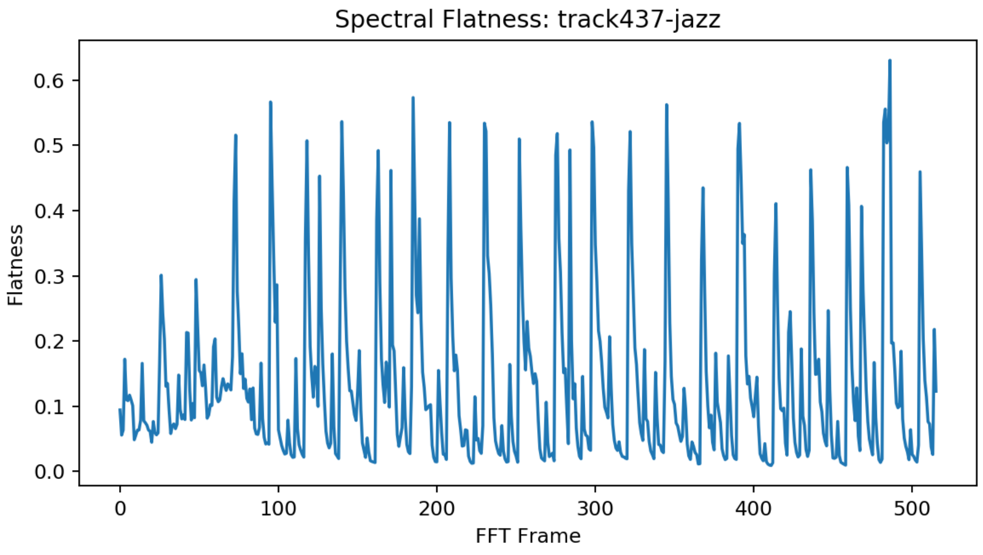
$$SF(n) = \frac{\left(\prod_{k=0}^{K} |X_n(k)|\right)^{1/K}}{\frac{1}{K}\sum_{k=0}^{K} |X_n(k)|} \tag{7}$$

A very small flatness corresponds to the presence of tonal components while a flatness equal to one corresponds to a very noisy signal. Thus, flatness is a measure of the noisiness of the spectrum.

Method This implementation was done using some low-level `scipy` and `numpy` methods, specifically `scipy.stats.mstats.gmean` and `numpy.mean` to calculate the geometric and arithmetic means, respectively. Again, this analysis was performed on classical, jazz, and metal samples. For the full Python implementation, see section 5.4.3.

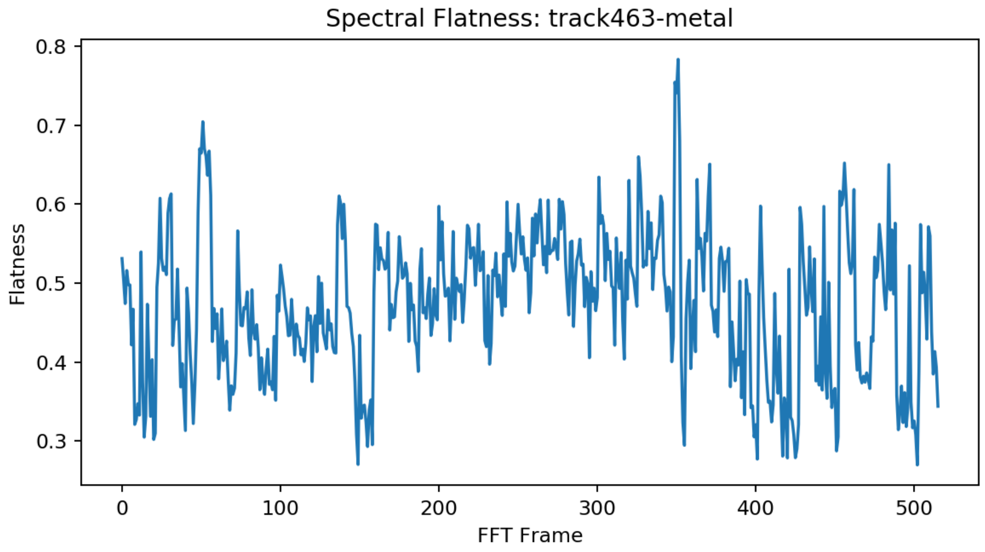


(a) Spectral flatness of 'track201-classical' as a function of frame number.



(b) Spectral flatness of 'track437-jazz' as a function of frame number.

Figure 7: Flatness



(c) Spectral flatness of 'track463-metal' as a function of frame number.

Figure 7: Flatness

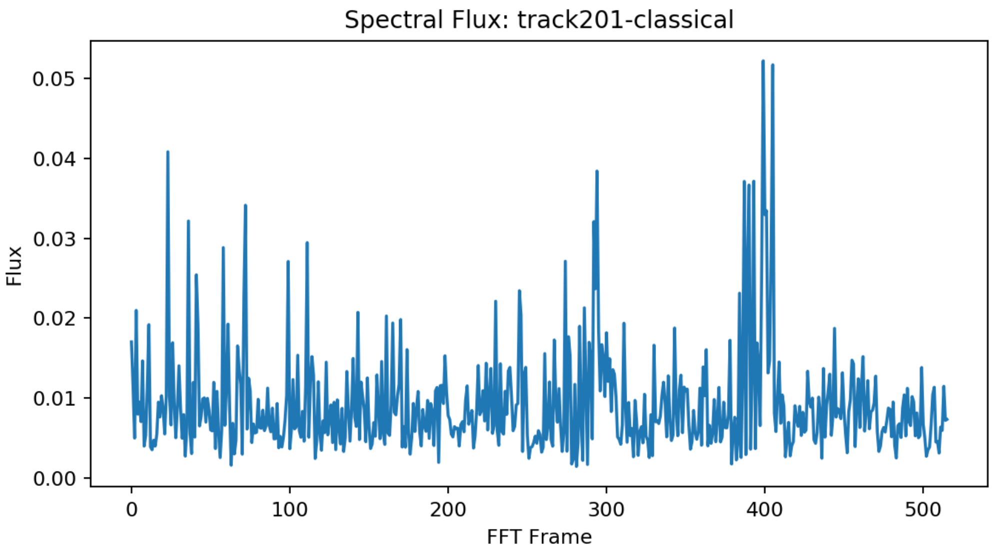
Comments Notice the magnitude of flatness across the entire metal sample (figure 7c) is almost double that of classical (figure 7a). This is a direct indication that the metal track is much noisier. Then comparing the flatness curve of this jazz sample, we see the same alternating magnitudes as in the centroid (figure 6c) and spread (figure 6d) curves, again indicating the noise inherent in the high-pitch components of this sample.

2.4 Spectral Flux

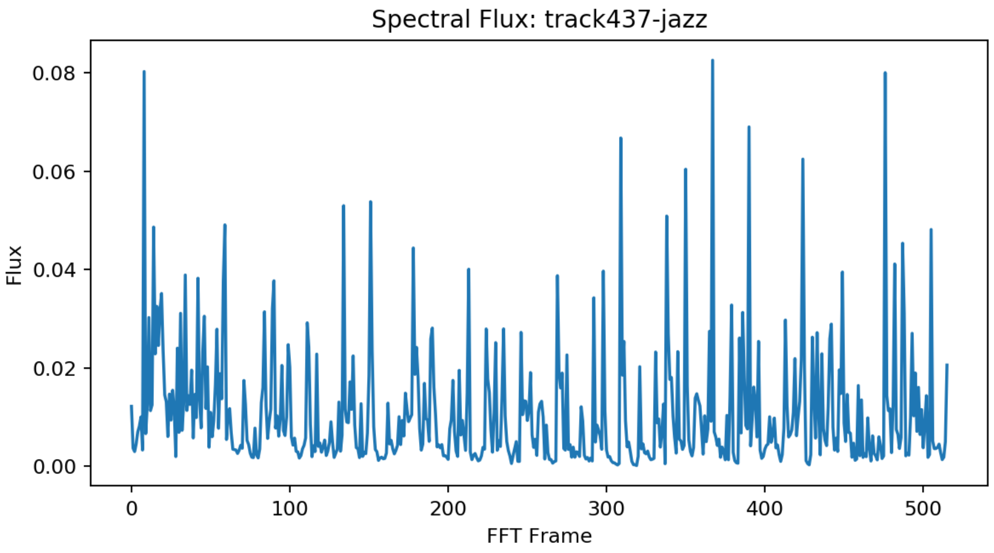
The spectral flux is a global measure of the spectral changes between two adjacent frames, $n-1$ and n,

$$F_n = \sum_{k=0}^{K} (P_n(k) - P_{n-1}(k))^2 \tag{8}$$

Where $P_n(k)$ is the normalized frequency distribution for frame n, given by 4. Applying 8 to the same three samples, the plots of figure 8 were produced. See section 5.4.4 fr Python implementation.

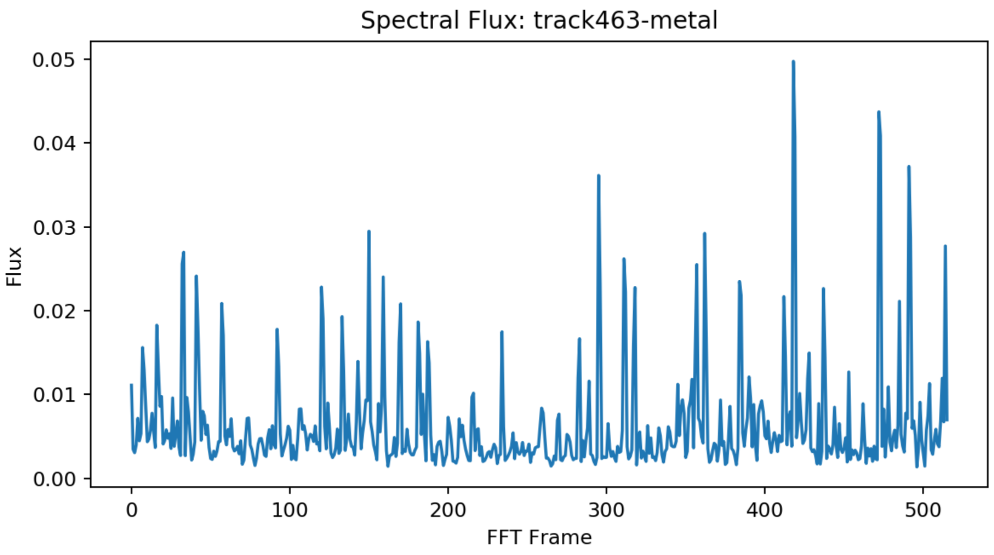


(a) Spectral flux of 'track201-classical' as a function of frame number.



(b) Spectral flux of 'track437-jazz' as a function of frame number.

Figure 8: Flux



(c) Spectral flux of 'track463-metal' as a function of frame number.

Figure 8: Flux

Comments Considering the similarity of figures 8b and 8c, spectral flux is not a good indication of genre.

3 Psychoacoustics

Background Psychoacoustics involves the study of the human auditory system and our perception of audio. In this lab, we will focus on some spectral content that can be defined mathematically. The human auditory system is sensitive to a range of frequencies with a logarithmic relationship. One very common model of human sound perception is called the mel/Bark scale.

3.1 The mel/Bark Scale

The Bark scale is defined as

$$z = 7arcsinh(f/650) = 7log\left(f/650 + \sqrt{1+(f/650)^2}\right),$$

where f is measured in Hz. However, in this lab, we will use a modified definition,

$$m = 1127.01048 * log(1 + f/700). \tag{9}$$

3.2 The Cochlear Filterbank

Additionally, the human auditory system behaves like a sequence of filters (filterbank) with overlapping frequency responses. Perception of pitch can be quantified using the total energy at the output of each filter, summing the spectral energy that falls into one critical band (the frequency band within which a second tone will interfere with the first). Our model is simple, with $N_B = 40$ logarithmically spaced triangle filters centered at frequencies Ω_p, which are implicitly defined as

$$mel_p = 1127.01048 * log(1 + \Omega_p/700). \tag{10}$$

The set of frequencies chosen to correspond to each Ω_p is chosen to be equally spaced the the mel scale. Letting the indexing of the center frequencies of the filters start with $p = 1$, we define

$$mel_p = p\frac{mel_max - mel_min}{N_B + 1} + mel_min \tag{11}$$

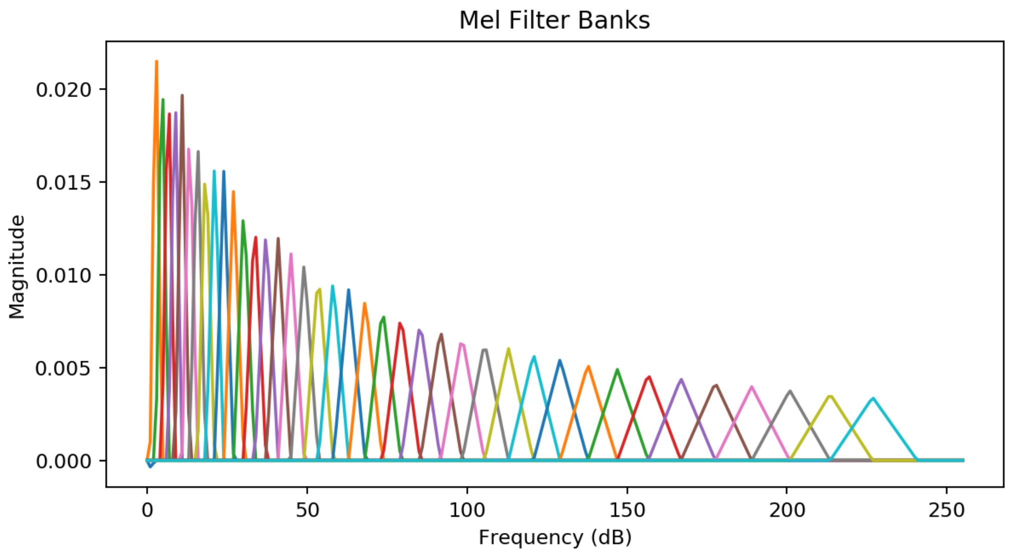


Figure 9: A collection of $N_B = 40$ filterbanks to be used in the calculation of mfcc coeffients when convolved with an audio sample.

where N_B is the number of filters in the filterbank and

$$mel_min = 1127.01048 * log(1 + 20/700),$$
$$mel_max = 1127.01048 * log(1 + 0.5 * f_s/700),$$

taking f_s to be the sampling frequency of the audio. This definition gives mel_max to be defined as the highest frequency preserved in the audio file and mel_min to be $20Hz$ (a good baseline for the limit of human hearing). We then define H_p, the hat filter corresponding to each filter to be centered around Ω_p:

$$H_p(f) = \begin{cases} \dfrac{2}{\Omega_{p+1} - \Omega_{p-1}} \dfrac{f - \Omega_{p-1}}{\Omega_p - \Omega_{p-1}} & \text{if } f \in [\Omega_{p-1}, \Omega_p), \\ \dfrac{2}{\Omega_{p+1} - \Omega_{p-1}} \dfrac{\Omega_{p+1} - f}{\Omega_{p+1} - \Omega_p} & \text{if } f \in [\Omega_p, \Omega_{p+1}). \end{cases} \tag{12}$$

Each triangular filter in 12 is normalized such that the integral of each filter is 1 and the filters overlap so that the frequency where H_p is maximum is at the starting frequency of the next filter, H_{p+1} and ending frequency of H_{p-1}.

Finally, the implementation of the above derivation produces the plot shown in figure 9.

Comments The triangle filters in figure 9 appear to decrease and increase on the left side of the x-axis, however this is an anomaly due to the quantization of values used to generate each filter. This can be seen in the offset peaks in a few of the filters. In reality, each filter maintains a unity integral over the range N_B. Python implementation for this can be found in section 5.5.

4 Conclusion

Spectral analysis of audio is definitively more useful than time-domain analysis, however this lab showed just how difficult it still can be to differentiate genres analytically. I think this was a good introduction to some of the useful statistical analysis techniques that can be used to programmatically sort signals based on characteristic features. Additionally, the script to run this analysis was relatively slow. With the digital age we live in, delay from input to output is very important, so some optimization would most likely help to make this sort of analysis even more useful. Next, I would like to apply this analysis to some of my favorite songs and see what I can gain from it. Eventually, I envision using this type of functionality to create a highly correlated audio visualizer or an advanced tool to equalize concert halls.

5 Appendix

5.1 Appendix A: Main

```python
"""
Created on Wed Jan 16 15:24:09 2019

@author: Andrew Teta
"""

import scipy.io.wavfile
import scipy.signal
import scipy.stats
import numpy as np
import math
import matplotlib.pyplot as plt
from tkinter import filedialog

# UI dialog to select files -> selection of multiple files will run all functions for
    each file
files = filedialog.askopenfilenames()
# Loop over files selected
for f in files:
    filename = str.split(f,'/')[-1]
    filename = str.split(filename,'.')[0]
    filepath = f

    # Extract 24s sample from center of 60s clip
    fs,songSample = extractSample(filepath,24,60)

    # Calculate loudness (energy function returned just in case)
    E,sigma,Nloudness = loudness(songSample,512)

    # Save figures
    plt.figure(figsize=(8,4),dpi=170)
    plt.plot(range(Nloudness),sigma)
    plt.xlabel('FFT Frame')
    plt.ylabel('Loudness')
    plt.title('Loudness: '+filename)
    plt.savefig('figs/loudness_'+filename)
    plt.close()

    # Calculate zero-crossing-rate
    Z,Nzcr = zcr(songSample,512)

    plt.figure(figsize=(8,4),dpi=170)
    plt.plot(range(Nzcr),Z)
    plt.xlabel('FFT Frame')
    plt.ylabel('ZCR')
    plt.title('ZCR: '+filename)
    plt.savefig('figs/zcr_'+filename)
    plt.close()

    # Calculate spectrograms for Hann and Blackman windows
    spectroHann,spectroBlack,freqAxis,timeAxis = spectrogram(songSample,fs,512)
    spectroHann = np.transpose(spectroHann)
    spectroBlack = np.transpose(spectroBlack)

    mel = mfcc(spectroBlack,freqAxis,1,fs,40)

    # save figures
    plt.figure(figsize=(8,4),dpi=170)
    plt.imshow(spectroHann, cmap='inferno')
    plt.gca().invert_yaxis()
    plt.colorbar(shrink=0.5).set_label('Magnitude (dB)')
    plt.title('Hann Window Spectrogram: '+filename)
    plt.xlabel('FFT Frame')
    plt.ylabel('Frequency (bins)')
    plt.savefig('figs/powerHann_'+filename)
    plt.close()

    plt.figure(figsize=(8,4),dpi=170)
    plt.imshow(spectroBlack, cmap='inferno')
    plt.gca().invert_yaxis()
    plt.colorbar(shrink=0.5).set_label('Magnitude (dB)')
    plt.title('Blackman Window Spectrogram: '+filename)
    plt.xlabel('FFT Frame')
    plt.ylabel('Frequency (bins)')
    plt.savefig('figs/powerBlack_'+filename)
    plt.close()

    # calculate statistical vectors, centroid and spread
    centroid,spread,P,nFrames = spectralCentroid(songSample,fs,512)

    # save figures
    plt.figure(figsize=(8,4),dpi=170)
    plt.plot(centroid)
    plt.title('Spectral Centroid: '+filename)
    plt.ylabel('Frequency (Hz)')
    plt.xlabel('FFT Frame')
    plt.savefig('figs/centroid_'+filename)
    plt.close()

    plt.figure(figsize=(8,4),dpi=170)
    plt.plot(spread)
    plt.title('Spectral Spread: '+filename)
    plt.ylabel('Spread')
    plt.xlabel('FFT Frame')
    plt.savefig('figs/spread_'+filename)
    plt.close()

    # calculate spectral flatness
    flat,nFrames = flatness(songSample,fs,512)

    # save figure
    plt.figure(figsize=(8,4),dpi=170)
    plt.plot(flat)
    plt.ylabel('Flatness')
    plt.xlabel('FFT Frame')
    plt.title('Spectral Flatness: '+filename)
    plt.savefig('figs/flatness_'+filename)
    plt.close()

    sflux = flux(P)

    plt.figure(figsize=(8,4),dpi=170)
    plt.plot(sflux)
    plt.ylabel('Flux')
    plt.xlabel('FFT Frame')
    plt.title('Spectral Flux: '+filename)
    plt.savefig('figs/flux_'+filename)
    plt.close()

    print('done')

```

5.2 Appendix B: Sample Extraction

```
# ================================================================================
# Input:
#   filepath = string containing absolute path to audio file
#   duration = length of sample to extract in seconds
#   location = start index of sample given in seconds
# Output:
#   arg1 = sampling frequency (Hz)
#   arg2 = sample of specified duration in seconds
# ================================================================================
def extractSample(filepath, duration, location):
    fs,data = scipy.io.wavfile.read(filepath)
    print('Read file: '+filepath+'\n')
    Tsample = 1/fs
    startIndex = int(location/Tsample)
    endIndex = startIndex + int(duration/Tsample)
    return fs,data[startIndex:endIndex]

```

5.3 Appendix C: Time-domain Analysis

5.3.1 Loudness

```
# ================================================================================
# Input:
#   sample = audio clip
#   N = frame size
# Output:
#   arg1 = energy vector
#   arg2 = standard deviation vector
#   arg3 = num frames (for accurate plotting)
# ================================================================================
def loudness(sample,N):
    nframes = int(len(sample)/N)
    E = np.zeros(nframes)
    sigma = np.zeros(nframes)
    for n in range(nframes):
        E[n] = (1/N)*sum(sample[n*N:n*N+(N-1)])
    for n in range(nframes):
        sigma[n] = np.sqrt((1/N)*sum((sample[n*N:n*N+(N-1)] - E[n])**2))
    return E,sigma,nframes

```

5.3.2 ZCR

```
# ================================================================================
# Input:
#   sample = audio clip
#   N = frame size
# Output:
#   arg1 = zero-crossing-rate
#   arg2 = num frames (for accurate plotting)
# ================================================================================
def zcr(sample,N):
    nframes = int(len(sample)/N)
    Z = np.zeros(nframes)
    for n in range(nframes):
        Z[n] = 1/(N-1) * sum(0.5*abs(np.sign(sample[n*N+1:n*N+(N-1)]) - np.sign(sample[
    n*N:n*N+(N-2)])))
    return Z, nframes

```

5.4 Appendix D: Frequency-domain Analysis

5.4.1 Spectrogram

```python
# ==============================================================================
# Input:
#   sample = audio clip
#   fs = sampling frequency
#   N = frame size
# Output:
#   arg1 = spectrogram using Hann window
#   arg2 = spectrogram using Blackman window
#   arg3 = vector of axis points in frequency (Hz)
#   arg4 = vector of sxis points in time (s)
# ==============================================================================
def spectrogram(sample,fs,N):
    # generate a Hann window
    wHann = scipy.signal.get_window('hann',N-1,False)
    # Generate Blackman window
    wBlack = scipy.signal.get_window('blackman',N-1,False)
    # Save figure
    plt.figure(dpi=170)
    plt.plot(wHann)
    plt.title('Hann Window')
    plt.xlabel('Samples')
    plt.ylabel('Magnitude')
    plt.savefig('figs/hann')
    plt.close()
    # save figure
    plt.figure(dpi=170)
    plt.plot(wBlack)
    plt.title('Blackman Window')
    plt.xlabel('Samples')
    plt.ylabel('Magnitude')
    plt.savefig('figs/blackman')
    plt.close()
    # Number of FFT frames
    nframes = int(len(sample)/N)
    SHann = np.zeros([nframes,int(N/2)])
    SBlack = np.zeros([nframes,int(N/2)])
    time = np.zeros(nframes)
    # Loop over FFT frames
    for n in range(nframes):
        # Calculate a frequency power spectrum for Hann window
        f,t,sH = np.transpose(scipy.signal.spectrogram(sample[n*N:n*N+(N-1)],fs,wHann,
    mode='magnitude'))
        # throw away values smaller than 10^-3 and square values
        sH = sH**2
        sH = np.where(sH < 10**-3, 10**-3, sH)
        # Repeat for Blackman window
        f,t,sB = np.transpose(scipy.signal.spectrogram(sample[n*N:n*N+(N-1)],fs,wBlack,
    mode='magnitude'))
        sB = sB**2
        sB = np.where(sB < 10**-3, 10**-3, sB)
        # Reshape a bit
        SHann[n,:] = np.transpose(sH*fs/N)
        SBlack[n,:] = np.transpose(sB*fs/N)
        time[n] = t
    # convert linear values to dB scale
    SHann = 20*np.log10(SHann/np.amax(SHann))
    SBlack = 20*np.log10(SBlack/np.amax(SBlack))
    return SHann,SBlack,f,time

```

5.4.2 Spectral Centroid and Spread

```python
# ====================================================================================
# Input:
#   sample = audio clip
#   fs = sampling frequency
#   N = frame size
# Output:
#   arg1 = vector of size nframes of spectral centroids
#   arg2 = vector of size nframes of spectral spreads
#   arg3 = length of output vectors = nframes
# ====================================================================================
def spectralCentroid(sample,fs,N):
    nframes = int(len(sample)/N)
    SP = np.zeros([nframes,int(N/2)])
    # Generate Blackman window
    wBlack = scipy.signal.get_window('blackman',N-1,False)
    # Generate a spectrogram matrix
    for n in range(nframes):
        f,t,sp = np.transpose(scipy.signal.spectrogram(sample[n*N:n*N+(N-1)],fs,wBlack,
    mode='magnitude'))
        SP[n,:] = np.transpose(sp)
    frange = np.shape(sp)[0]
    P = np.zeros([nframes,frange])
    mu = np.zeros(nframes)
    spread = np.zeros(nframes)
    # Loop over FFT frames
    for n in range(nframes):
        # calculate sum of all frequency components in this frame
        sfreq = sum(SP[n,0:frange])
        for k in range(frange):
            # calculate the relative 'probability' of each frequency bin
            P[n,k] = SP[n,k]/sfreq
            # calculate spectral centroid (center of mass) of each frame
            mu[n] = mu[n] + (k*fs/N)*P[n,k]
            # calculate spectral spread of each frame
            spread[n] = np.sqrt(spread[n] + P[n,k]*(k-mu[n])**2)
    return mu,spread,P,nframes

```

5.4.3 Spectral Flatness

```python
# ===========================================================================
# Input:
#   sample = audio clip
#   fs = sampling frequency
#   N = frame size
# Output:
#   arg1 = vector of length nframes of spectral flatness
#   arg2 = length of output vectors = nframes
# ===========================================================================
def flatness(sample,fs,N):
    nframes = int(len(sample)/N)
    K = int(N/2)
    SP = np.zeros([nframes,int(N/2)])
    wBlack = scipy.signal.get_window('blackman',N-1,False)
    geo = np.zeros(nframes)
    arith = np.zeros(nframes)
    SF = np.zeros(nframes)
    for n in range(nframes):
        f,t,sp = np.transpose(scipy.signal.spectrogram(sample[n*N:n*N+(N-1)],fs,wBlack,
    mode='magnitude'))
        SP[n,:] = np.transpose(sp)
        geo[n] = scipy.stats.mstats.gmean(SP[n,:])
        arith[n] = np.mean(SP[n,:])
    SF = geo/arith
    return SF,nframes

```

5.4.4 Spectral Flux

```python
# ===========================================================================
# Input:
#   P = probability matrix of dimension [nframes,N/2]
# Output:
#   arg1 = vector of length nframes of spectral flux
# ===========================================================================
def flux(P):
    nframes = np.shape(P)[0]
    frange = np.shape(P)[1]
    F = np.zeros(nframes)
    for n in range(nframes):
        F[n] = sum((P[n,:]-P[n-1,:])**2)
    return F

```

5.5 Psychoacoustics

5.5.1 Mel Filter Banks and Coefficients

```python
# ==============================================================================
# Input:
#   SP = audio sample spectrogram
#   f = list of frequency values with size = range of frequency bins in SP
#   n = frame index of SP
#   fs = sampling frequency of audio sample
#   nBanks = number of mel filter banks to generate
# Output:
#   arg1 = N_B x K (K = len(f)) matrix of filter values for H_p
#   arg2 = matrix of filter banks, size = nBnaks x len(f)
# ==============================================================================
def mfcc(SP,f,n,fs,nBanks):
    mel_min = 1127.01048*math.log(1 + 20/700)
    omega_min = (math.e**(mel_min/1127.01048) - 1) * 700
    mel_max = 1127.01048 * math.log(1 + 0.5*fs/700)
    omega_max = (math.e**(mel_max/1127.01048) - 1) * 700
    # Generate frequency array in Hz
    linfreq = np.arange(np.round(omega_min,0),np.round(omega_max,0)+1,1)
    # Map frequency array to mel array
    melfreq = 1127.01048*np.log(1+linfreq/700)
    # Generate nbanks+2 mel frequencies uniformly spaced between mel_min and mel_max, inclusive
    mel = np.linspace(mel_min,mel_max,nBanks+2)
    # Array of center frequencies of each filter
    omega = np.zeros(nBanks+2,np.float64)
    # Populate array
    for i in range(nBanks+2):
        idx = np.argmin(np.abs(melfreq-mel[i]))
        omega[i] = linfreq[idx]
    # Generate filters
    nframes = np.shape(SP)[1]
    frange = np.shape(SP)[0]
    mfcc = np.zeros([nBanks,frange])
    h = np.zeros([nBanks,frange])
    plt.figure(figsize=(8,4),dpi=170)
    plt.ylabel('Magnitude')
    plt.xlabel('Frequency (dB)')
    plt.title('Mel Filter Banks')
    for p in range(nBanks):
        omegaC = omega[p]
        omegaL = omega[p-1]
        omegaR = omega[p+1]
        for k in range(frange):
            freq = f[k]
            if (freq >= omegaL and freq < omegaC):
                h[p,k] = (2/(omegaR-omegaL)) * ((freq-omegaL)/(omegaC-omegaL))
            elif (freq >= omegaC and freq < omegaR):
                h[p,k] = (2/(omegaR-omegaL)) * ((omegaR-freq)/(omegaR-omegaC))
            else:
                pass
        plt.plot(range(len(f)),h[p])
    plt.savefig('figs/melBanks')
    #mfcc[p-1,:] = np.sum((np.abs(h*SP[:,n]))**2)
    return mfcc,h

```