Lab 1

Michael Eller

February 1, 2016

Contents

1 Introduction

The goal of this lab is to explore different methods of categorizing music into specific genres. Several methods of automated music analysis exist already, including **score following**, **automatic music transcription**, **music recommendation**, and **machine listening**. Unfortunately, all of these tools are still rudimentary. Faster, more efficient, methods still need to be developed in order to allow automated music analysis to be implemented on portable music players. The goal of this lab is to develop tools to automatically extract some defining features of music that will help us to categorize them more easily.

2 Sampling Rates

Current high-quality audio standards have a 24-bit depth and is sampled at 96 kHz. This 24-bit depth means there are 16,777,216 possible values for the audio signal at any given instance. It also means we are able to replicate frequencies up to 48 kHz. While this is far above the 20 kHz limit of human hearing, DVD audio is simply not high enough quality enough for a dolphin to listen to.

Assignment

1. Dolphins can only hear sounds over the frequency range [7 - 120] kHz. At what sampling frequency f_s should we sample digital audio signals for dolphins?

Nyquist theorem states that in order to accurately recreate a signal with maximum frequency $f_s/2$, we must sample at a minimum frequency of f_s. Therefore, our sampling frequency for digital audio signals for dolphins should be at minimum 240 kHz. Currently, one of the highest portable audio standards of BD-ROM LPCM (lossless) allow for 24 bit/sample and a maximum sampling frequency of only 192 kHz. Less common standards do exist though: Digital eXtreme Definition at 352.8 kHz using for recording and editing Super Audio CDs, SACD at 2,822.4 kHz known as Direct Stream Digital, and Double-Rate DSD at 5,644.8 kHz used in some professional DSD recorders. Therefore, as it stands, we might want to stay away from producing audio for dolphins (at least until better recording standards are developed.

3 Audio signal

Assignment

2. Write a MATLAB function that extract T seconds of music from a given track. You will use the MATLAB function ~~waveread~~ audioread to read a track and the function play to listen to the track.

   In the lab you will use T =**24 seconds** from the middle of each track to compare the different algorithms. Download the files, and test your function.

Listing 1: extractSound.m

```
function [ soundExtract,p ] = extractSound( filename, time )
%extractSound Extracts time (in seconds) from the middle of the song
%   Write a MATLAB function that extract T seconds of music from a
%   given track. You will use the MATLAB function audioread to
%   read a track and the function play to listen to the track.
info = audioinfo(filename);
[song,~]=audioread(filename);
if time >= info.Duration
    soundExtract=song;
```

```
    p=audioplayer(soundExtract,info.SampleRate);
    return;
elseif time<= 1/info.SampleRate
    error('Too small of a time to sample');
end
samples=time*info.SampleRate;
soundExtract=song(floor(info.TotalSamples/2)-floor(samples/2):1: ...
    floor(info.TotalSamples/2)+floor(samples/2));
p=audioplayer(soundExtract,info.SampleRate);
end
```

This MATLAB function is fairly straightforward. MATLAB's built-in function *audioinfo* provided all the necessary attributes to allow my function to parse any .wav file and extract the number of samples needed.

4 Low Level Features: Time-Domain Analysis

The bulk of music analysis is done in the frequency spectrum, however, there are some low level features that can be found from the time-domain analysis of music as well.

> **Assignment**
>
> 3. Implement the loudness and ZCR and evaluate these features on the different music tracks. Your MATLAB function should display each feature as a time series in a separate figure.
> 4. Comment on the specificity of the feature and its ability to separate different musical genre.

4.1 Loudness

There is not an easy way to describe *loudness*, but one way to mathematically quantize it is by finding it standard deviation, $\sigma(n)$.

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

The following is an excerpt from my *loudness* function. My *extractSound* function extracts 24 seconds around the middle of the song. The song excerpt is then split into frames of size 255 overlapped (at least) halfway with the previous frame.

Listing 2: loudness.m

```
[y,~]=extractSound( filename, time );
frames_data = buffer(y,frameSize,ceil(frameSize/2));
loudness_data=std(frames_data,0,1);
```

Figure 1 shows an example of the function applied to "track201-classical.wav". The figure would imply that, for the selection of the song, the beginning is quite loud, then quiets down until two loud moments, and finishes at a moderately quiet section. If one listens to the song selection in question, the results of my loudness function can be confirmed. The other loudness plots can be found in Appendix A.1

The **loudness** of a track appears to be most helpful in identifying *classical* and *jazz* music. Classical is very fluid and will not transition quickly between a high vs a low loudness. Jazz is a very periodic music and that is evident by the loudness plots where there are periodic spikes with a lull in between.

4.2 Zero-Crossing Rate (ZCR)

Another low-level feature that can be gathered from a time-domain analysis is the Zero-Cross Rate (ZCR). This feature has been used heavily in both speech recognition and music information retrieval, being a key feature to classify percussive sounds, where a frequency analysis would not be able to identify such features.

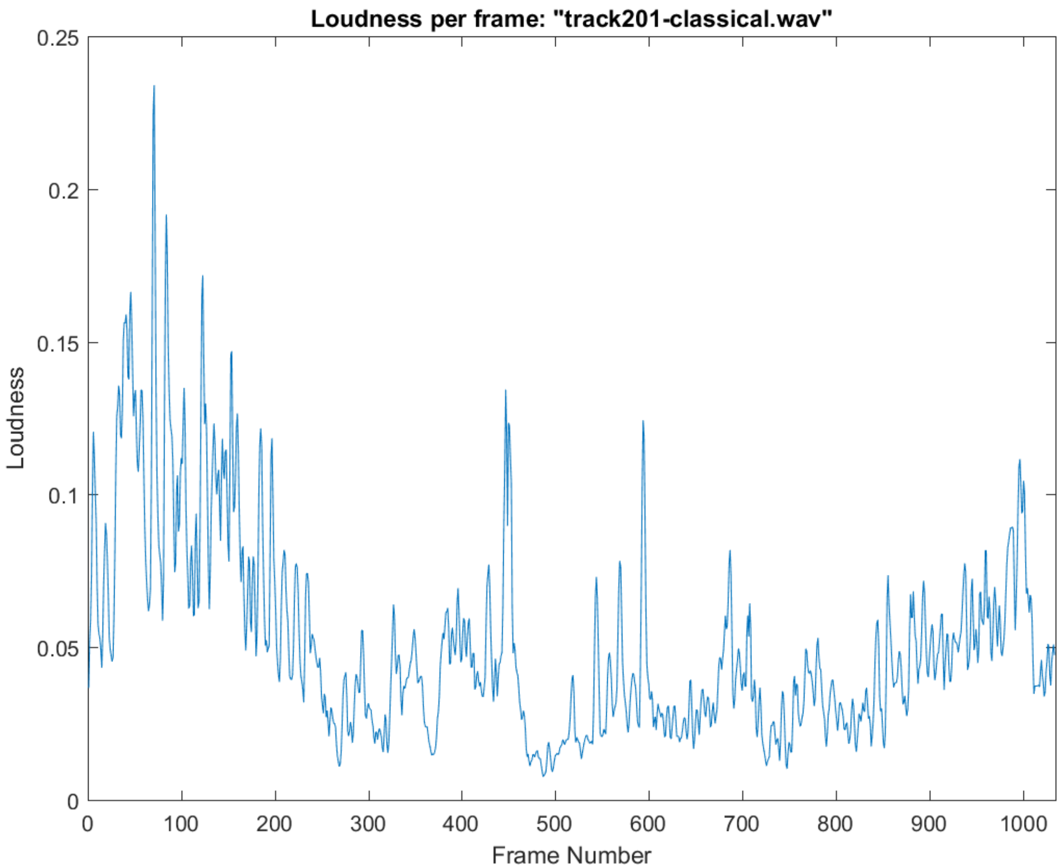


Figure 1: Loudness value per frame

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

The following is an excerpt from my *zeroCross* function. Once again, I extract the middle 24 seconds of audio from my track and store it into frames of size 255. Once **ZCR_data** has been allocated, each adjacent sample is compared to each other to check if they have crossed the zero-axis.

Listing 3: zeroCross.m

```
[y,~]=extractSound( filename, time ); % Operate on middle 24 seconds
frames_data = buffer(y,frameSize,ceil(frameSize/2));
ZCR_data=zeros(1,length(frames_data));
ZCR_data(1:end)=sum(abs(diff(frames_data(1:end,:))))/length(frames_data);
```

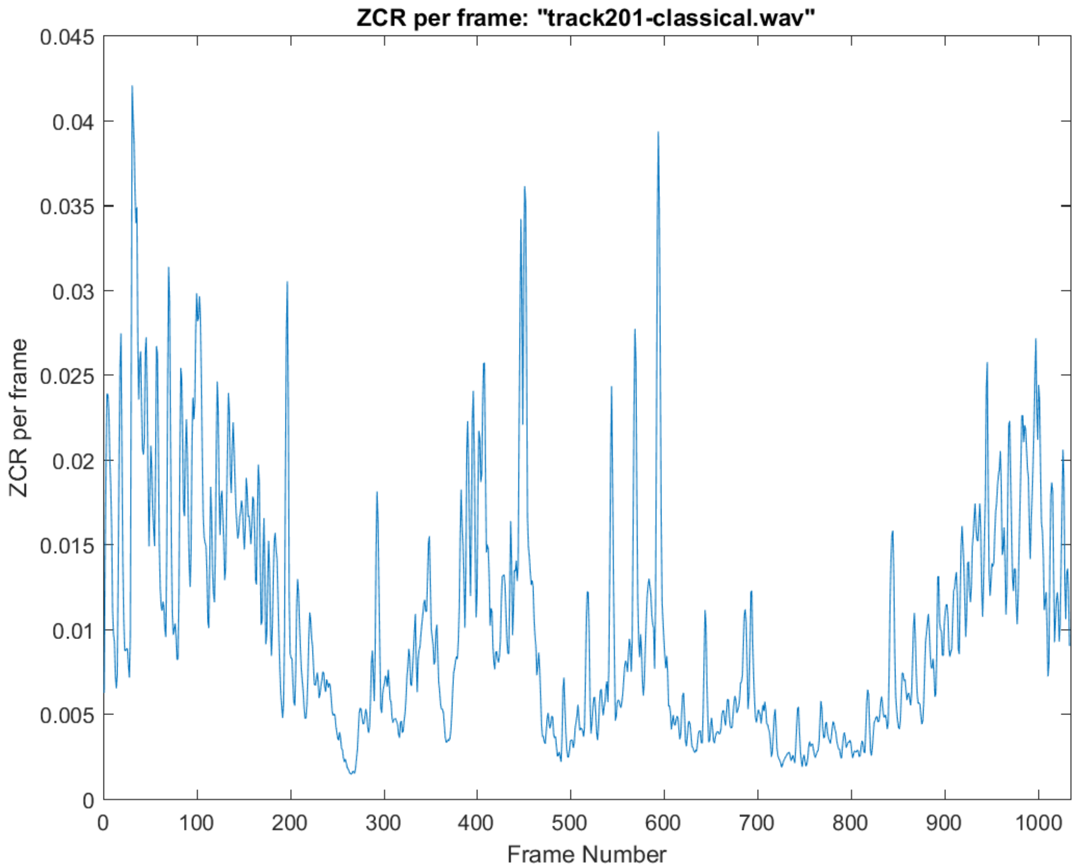


Figure 2: Average ZCR per frame

Figure 2 shows an example of the ZCR function applied to "track201-classical.wav". The plot produced by the ZCR function appears similar in shape to the plot from the previous loudness function. The ZCR function is more closely representative of how **noisy** a function is though. The other ZCR plots can be found in Appendix A.2.

Just as the **loudness** function, the **ZCR** function appears to be most helpful in identifying *classical* and *jazz* music. There doesn't seem to be any advantage in using **loudness** vs **ZCR** in identifying music genres. One method may prove to be faster on a large-scale integration, but such results cannot be determined on such a small unit test such as this one.

5 Low Level Features: Spectral Analysis

In this section, our goal is to reconstruct a musical score based on the spectral analysis of a piece. If we can apply the proper window function to each frame, we might be able to ascertain which note(s) are being played at that specific moment.

To properly evaluate the frequency domain of a frame, we need to perform the following operations:

$$\begin{aligned} Y &= \mathrm{FFT}(w. * x_n); \\ K &= N/2 + 1; \\ X_n &= Y(1:K); \end{aligned} \tag{3}$$

where w is our windowing function and N is the number of samples per frame.

Assignment

5. Let

$$x[n] = \cos(\omega_0 n), n \in \mathbb{Z} \tag{4}$$

Derive the theoretical expression of the discrete time Fourier transform of x, given by

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{5}$$

The Fourier transform of a cosine function is trivial at this point. A cosine function can be regarded as a sum of two complex exponentials. And the Fourier transform of a complex exponential is simply the dirac delta function offset by frequency ω. Therefore, the Fourier transform of $x[n] = \cos(\omega_0 n), n \in \mathbb{Z}$ is:

$$\sqrt{\frac{\pi}{2}}\delta(\omega - \omega_0) + \sqrt{\frac{\pi}{2}}\delta(\omega + \omega_0)$$

Assignment

6. In practice, we work with a finite signal, and we multiply the signal $x[n]$ by a window $w[n]$. We assume that the window w is non zero at times $n = -N/2, \dots, N/2$, and we define

$$y[n] = x[n]w[n - N/2] \tag{6}$$

Derive the expression of the Fourier transform $y[n]$ in terms of the Fourier transform of x and the Fourier transform of the window w.

From prior knowledge, we know that the Fourier transform multiplication of two functions in time domain is equal to the periodic convolution of two functions in the Fourier domain, as follows:

$$\text{if } y[n] = x[n] \cdot h[n]$$
$$\text{then, } Y(e^{j\omega}) = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{j\theta}) \cdot H(e^{j(\omega-\theta)})d\theta$$

We also know that a shift in time equates to a scaling in the frequency domain:

$$\mathcal{F}(x[n-k]) = X(e^{j\omega})e^{-j\omega k}$$

From these two properties, we can equate the DTFT of $y[n]$.

$$\text{if } y[n] = x[n]w[n - N],$$
$$\text{then } Y(e^{j\omega}) = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{j\theta}) \cdot (H(e^{j(\omega-\theta)}) \cdot e^{-j\omega N})d\theta$$

Assignment

7. Implement the computation of the windowed Fourier transform of y, given by Equation 3. Evaluate its performance with pure sinusoidal signals and different windows:

- Bartlett
- Hann
- Kaiser

8. Compute the spectrogram of an audio track as follows:

   (a) Decompose a track into a sequence of N_f overlapping *frames* of size N. The overlap between two frames should be $N/2$.

   (b) Compute the magnitude squared of the Fourier transform, $|X(k)|^2, k = 1, \ldots, K$ over each frame n.

   (c) Display the Fourier transform of all the frames in a matrix of size $K \times N_f$.

   You will experiment with different audio tracks, as well as pure sinusoidal tones. Do the spectrograms like what you hear?

In the rest of the lab, we will be using a Kaiser window to compute the Fourier transform, as explained in Equation 3.

In order to nicely bridge the gap between our continuous time perception of sound and a computer's discrete time processing, a windowing function is often used. This way, we are able to recreate the desired signal as close as possible without generating any high frequency components that result from abruptly stopping a time sample. In Figure 3 below, we are able to see the performance differences between each window. Overall, the Bartmann and Hamming windows seemed to perform the fastest, albeit by only a few milliseconds.

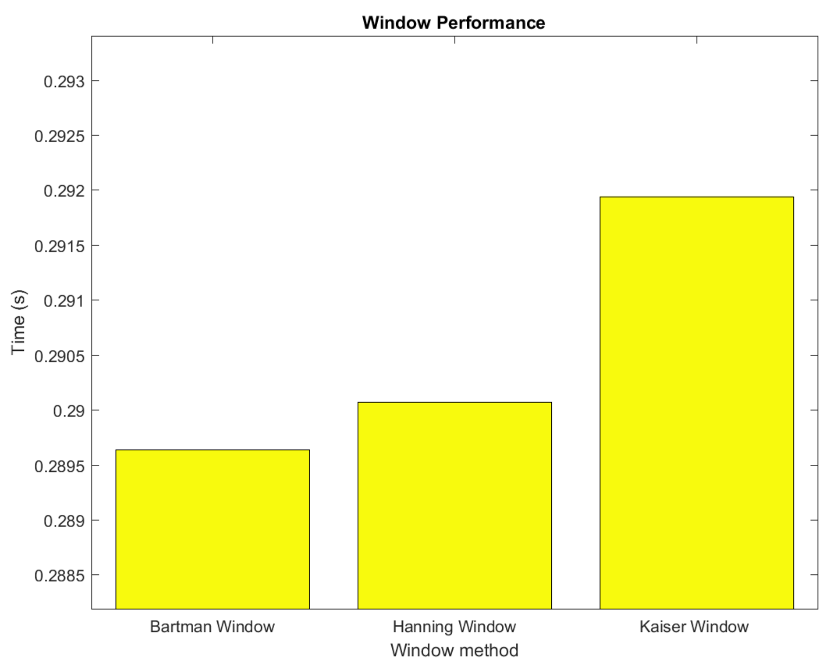


Figure 3: Timing comparison of different windows

Listing 4: windows.m

```
xn=buffer(x,frameSize);
Y=zeros(size(xn));
for i=1:length(xn)
    Y(:,i)=fft(xn(:,i).*w);
end
K=frameSize/2+1;
Xn=size(Y);
for i=1:length(xn)
    Xn(1:K,i)=Y(1:K,i);
end
end
```

In addition to how quickly each window is able to operate, we can also see how effectively each window extracts our signal. This can be seen in figures 4, 5, and 6.

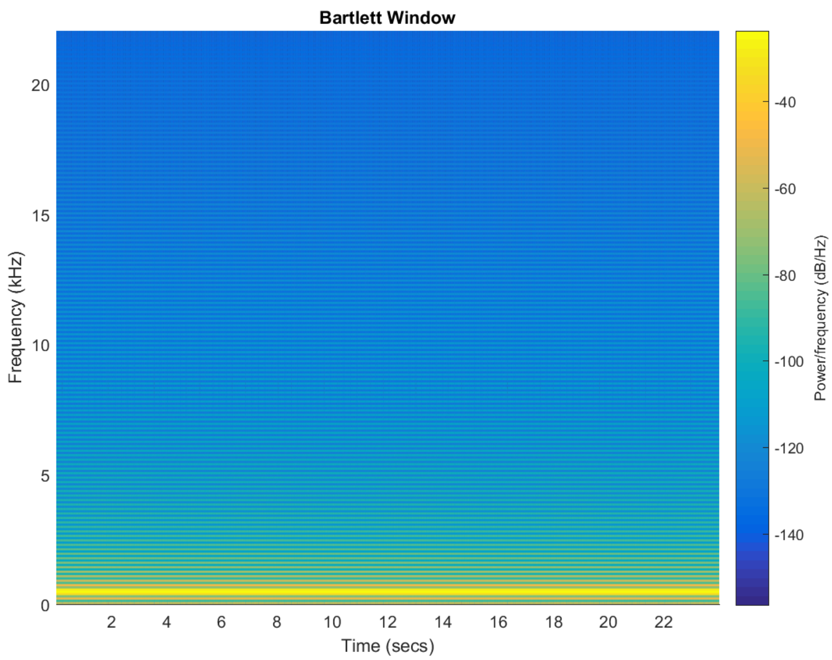


Figure 4: Spectrogram of Bartlett Window

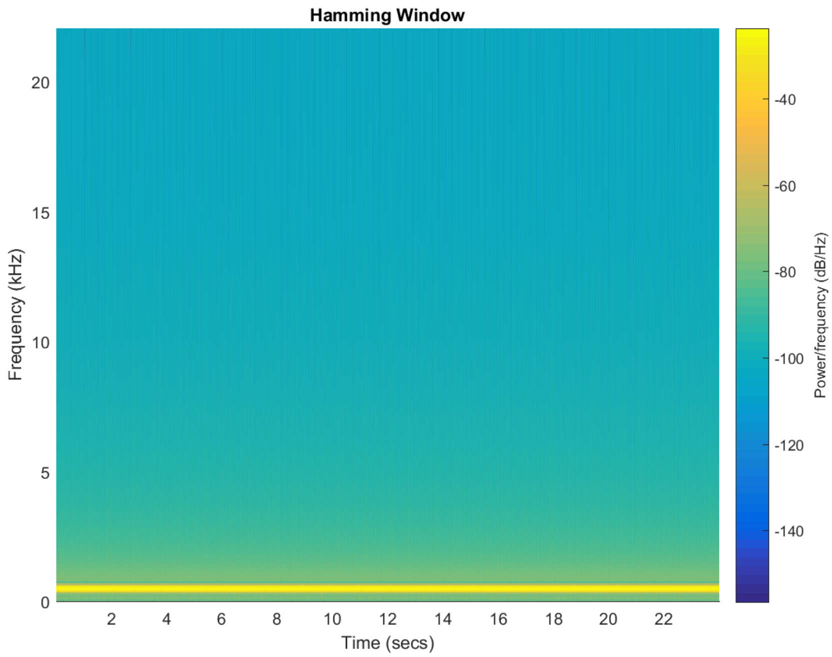


Figure 5: Spectrogram of Hamming Window

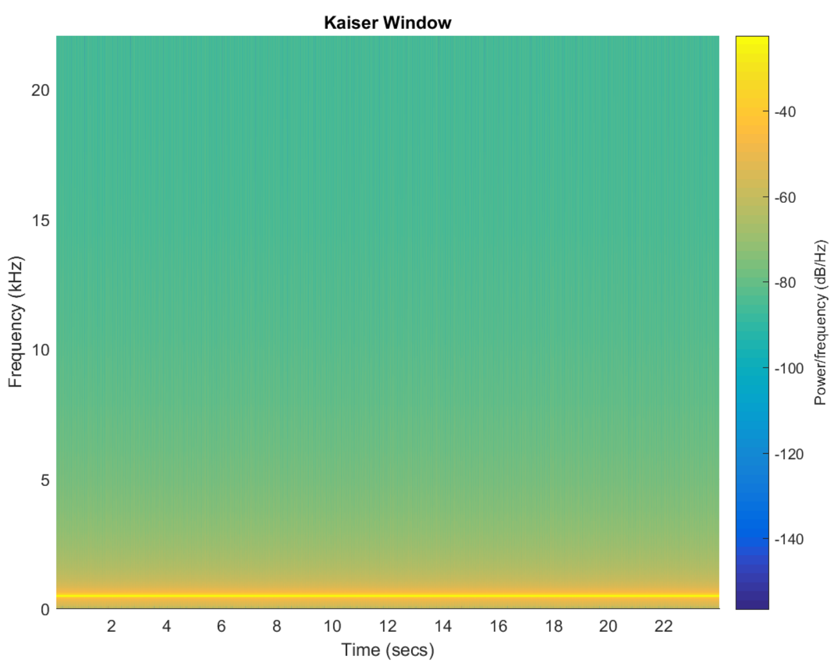


Figure 6: Spectrogram of Kaiser Window

In addition to being fairly fast, the Bartlett window seemed to be the most effective window in that its power to frequency ratio was very low away from the target frequency of 440 Hz.

Assignment

9. Implement all the low level spectral features them on the different tracks. Your MATLAB function should display each feature as a time series in separate figure.

10. Comment on the specificity of the feature, and its ability to separate different musical genres.

In addition to a song having low-level time-domain characteristics, low-level frequency-domain characteristics exist as well. With modern DSP chips, computing an FFT is no longer that big of an issue for DSP engineers. More plot for frequency centroids and frequency spreads can be found in Appendices A.4 and A.5, respectively.

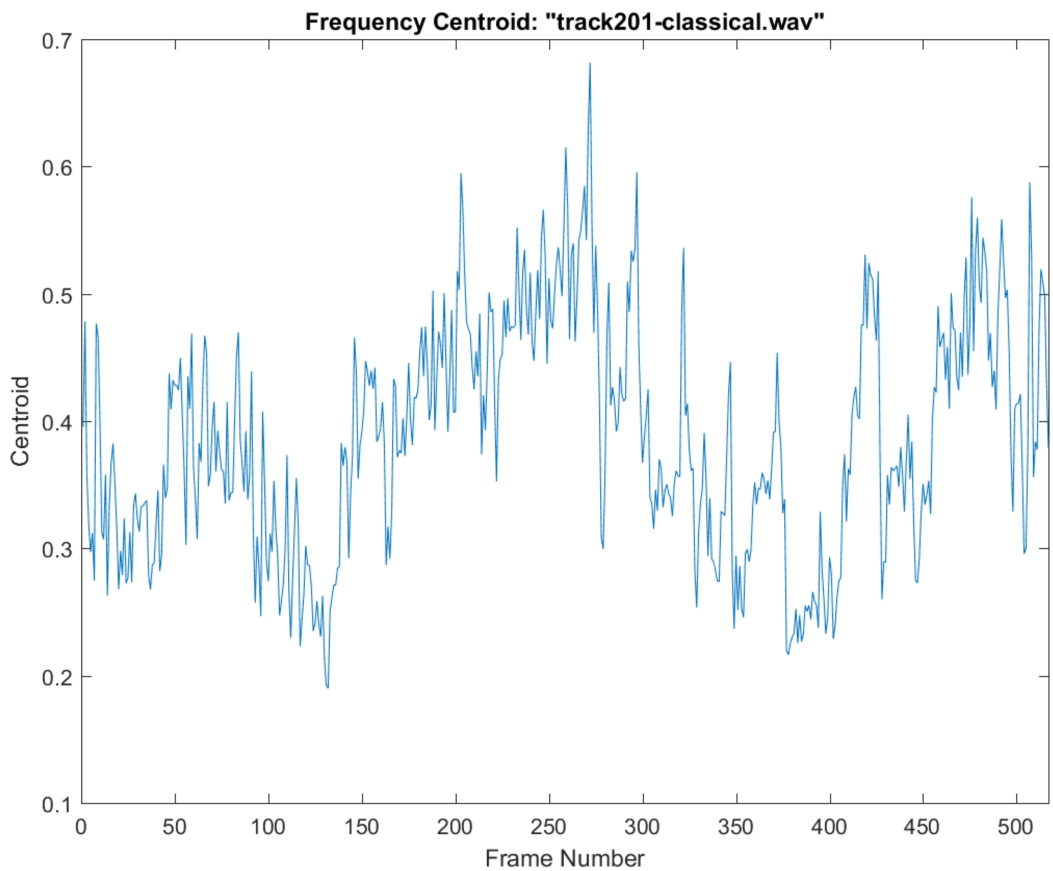


Figure 7: Frequency Centroid

Just as the in the time-domain analyses, the frequency-domain functions appear to be most helpful in identifying *classical* and *jazz* music. There doesn't seem to be any advantage in using **centroid** vs **spread** in identifying music genres. One method may eventually be better, but it is unclear at the moment.

Also, despite a more in depth analysis, these lower-level spectrum analysis tools seem to provide the same (if not less) help than the simple time-domain analyses.

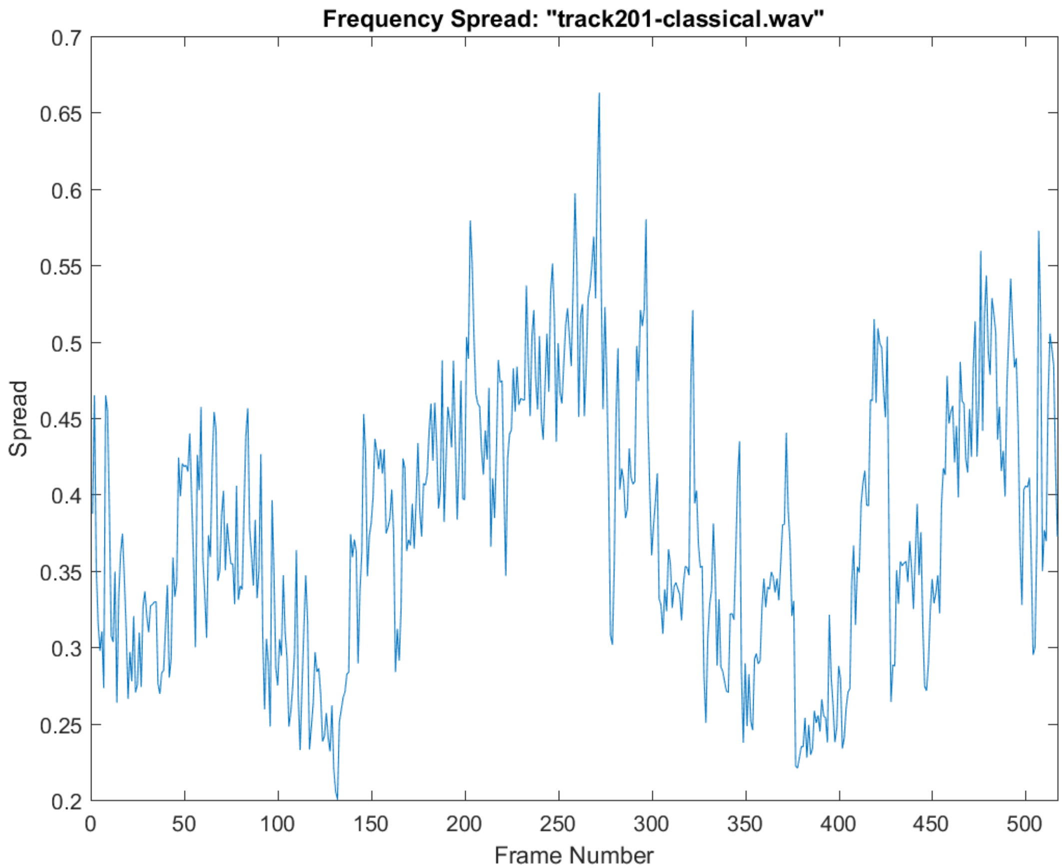
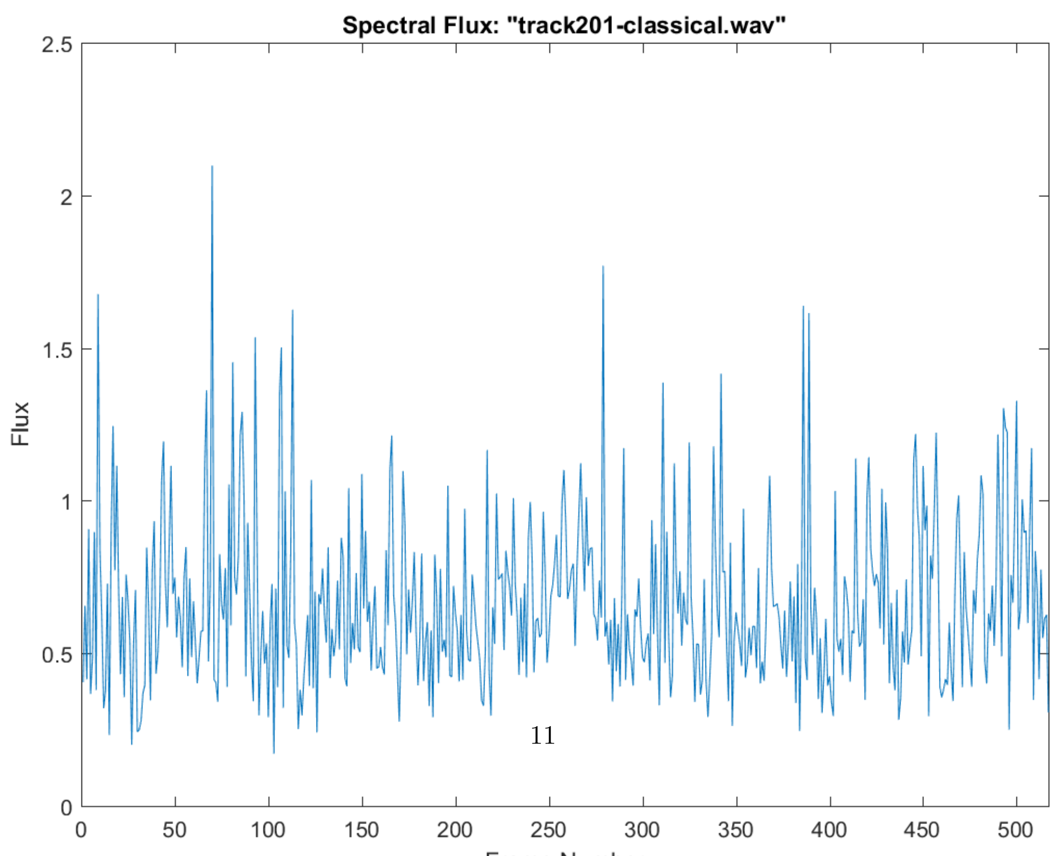


Figure 8: Frequency Spread



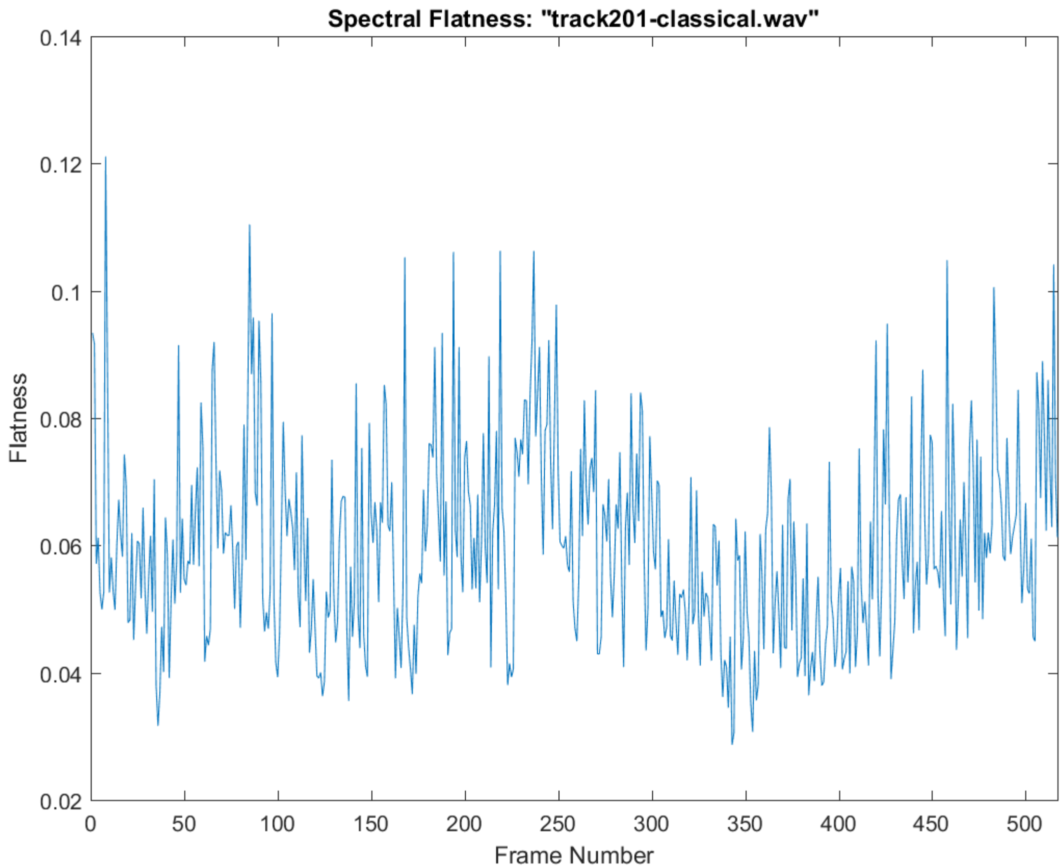


Figure 10: Spectral Flatness

6 Basic Psychoacoustic Quantities

Humans are very good at identifying music genres. Our problem is that we are too slow at it though. Humans generally identify a specific genre of music with its subjective features such as timbre, melody, harmony, rhythm, tempo, mood, lyrics, etc. In this section, we will focus solely on the aspects that are mathematically quantifiable.

Assignment

11. Implement the computation of the triangular filterbanks $H_p, p = 1, \ldots, N_B$. Your function will return an array **fbank** of size $N_B \times K$ such that $\mathbf{fbank(p,:)}$ contains the filter bank H_p.

12. Implement the computation of the mfcc coefficients, as defined by:

$$\text{mfcc}[p] = \sum_{k=1}^{K} |H_p(k) X_n(k)|^2 \tag{7}$$

With a proper filter bank created, we are one step closer to categorizing music into genres. Now that we are using a Mel scale, hopefully we will be able to categorize the pitches more closely to how a human ear would hear them. As opposed to a computer simply processing raw data.

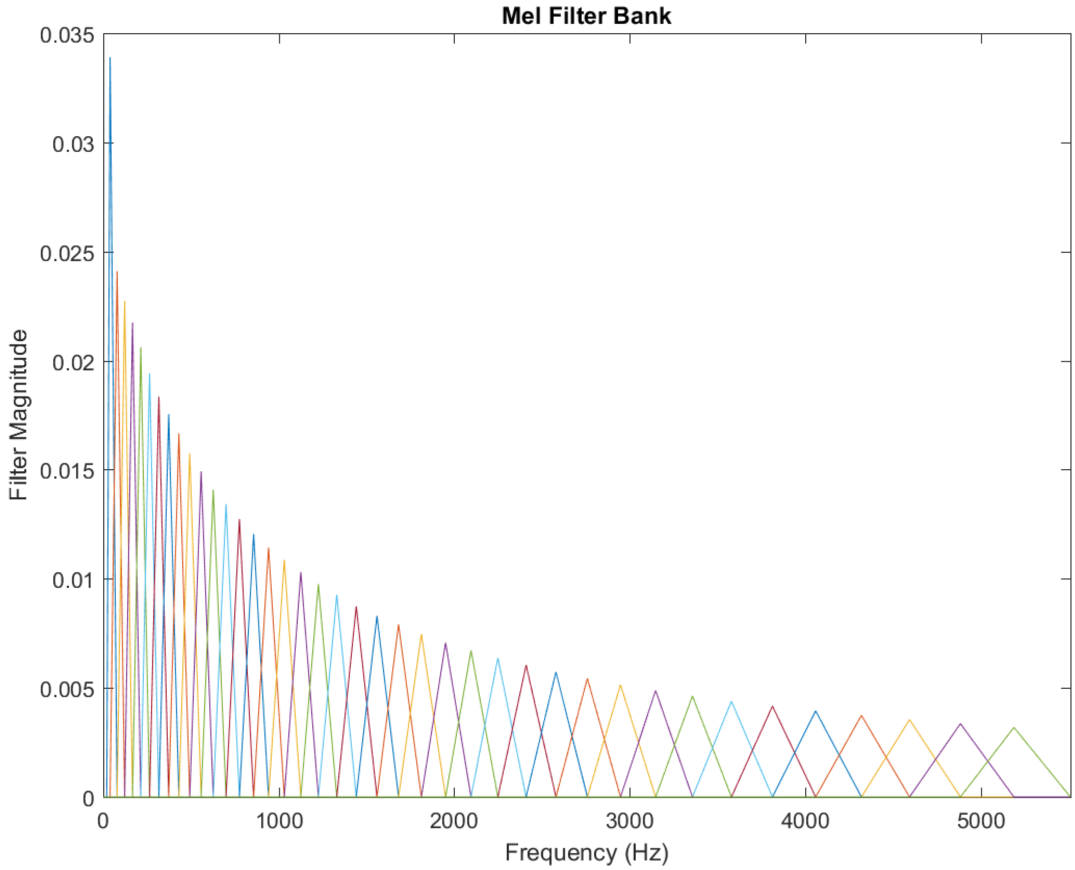


Figure 11: Mel Filter Bank

A Figures

A.1 Loudness

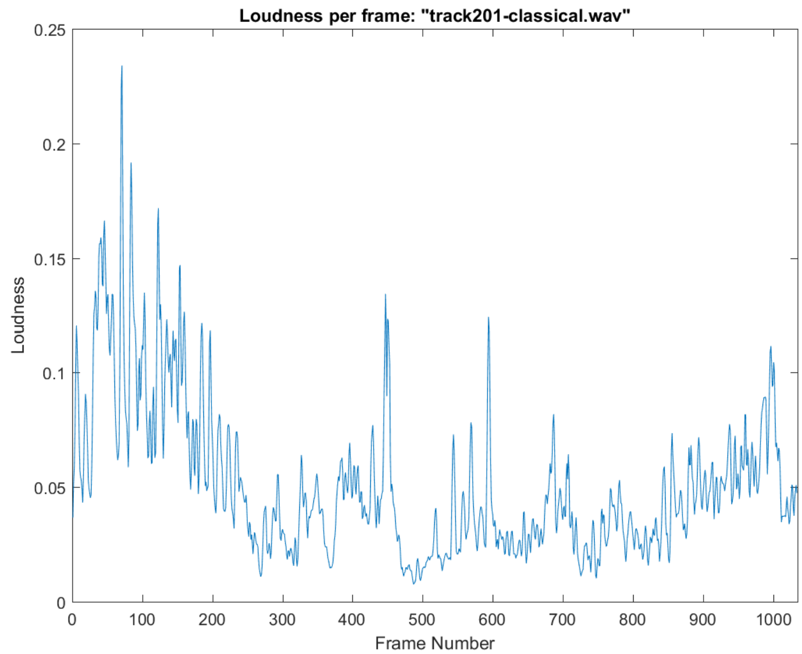


Figure 12: Loudness value per frame, classical201

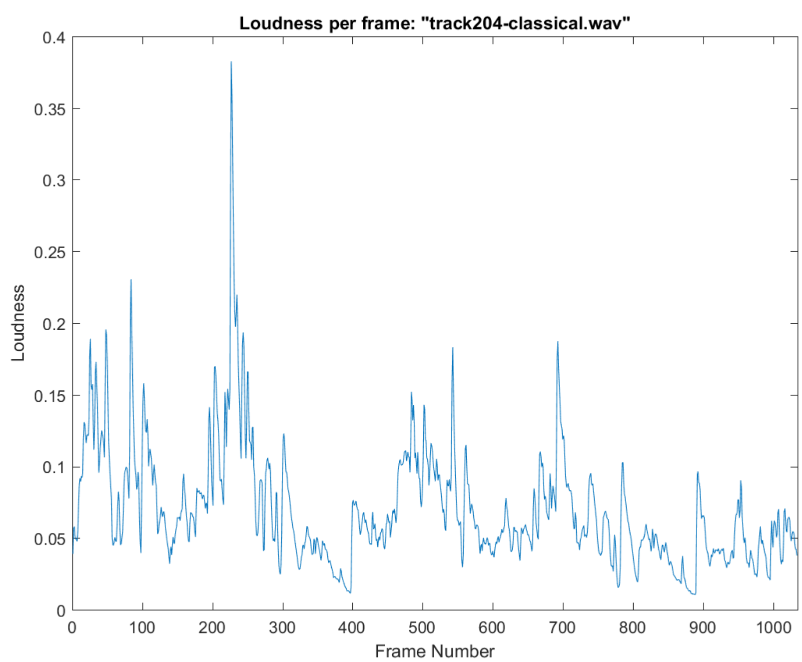


Figure 13: Loudness value per frame, classical204

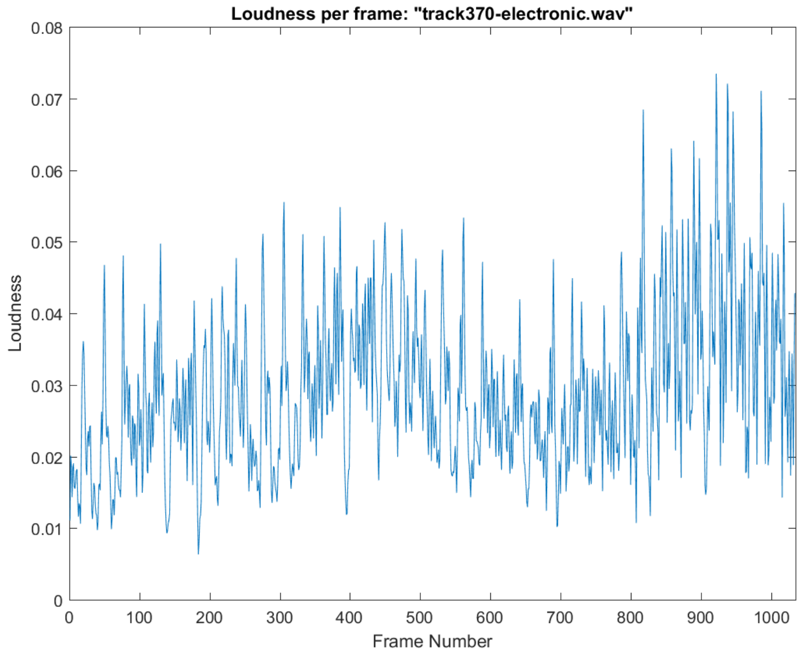


Figure 14: Loudness value per frame, electronic370

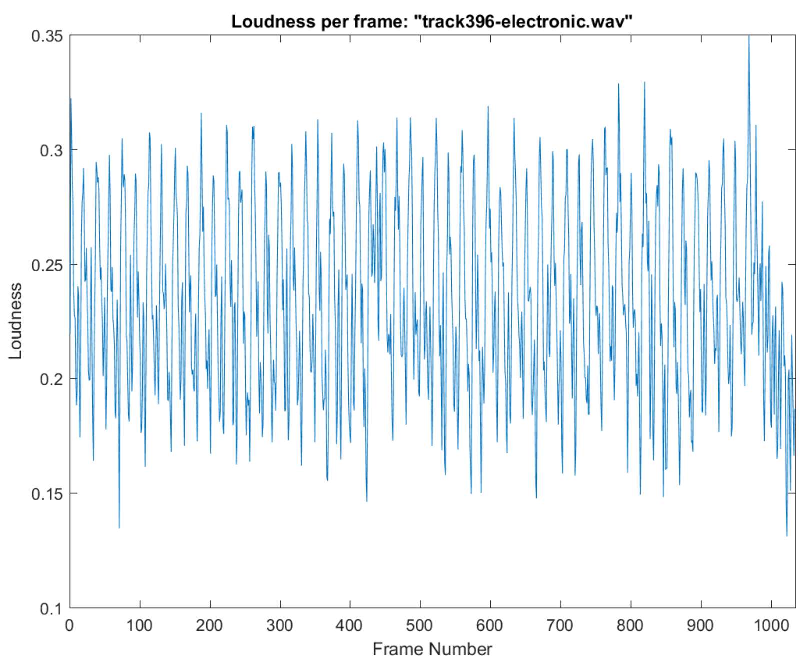


Figure 15: Loudness value per frame, electronic396

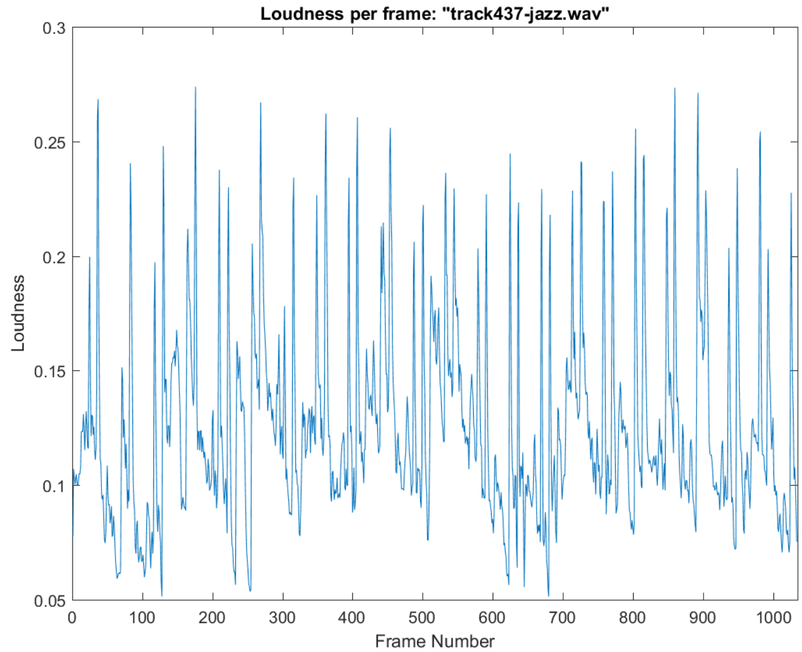


Figure 16: Loudness value per frame, jazz437

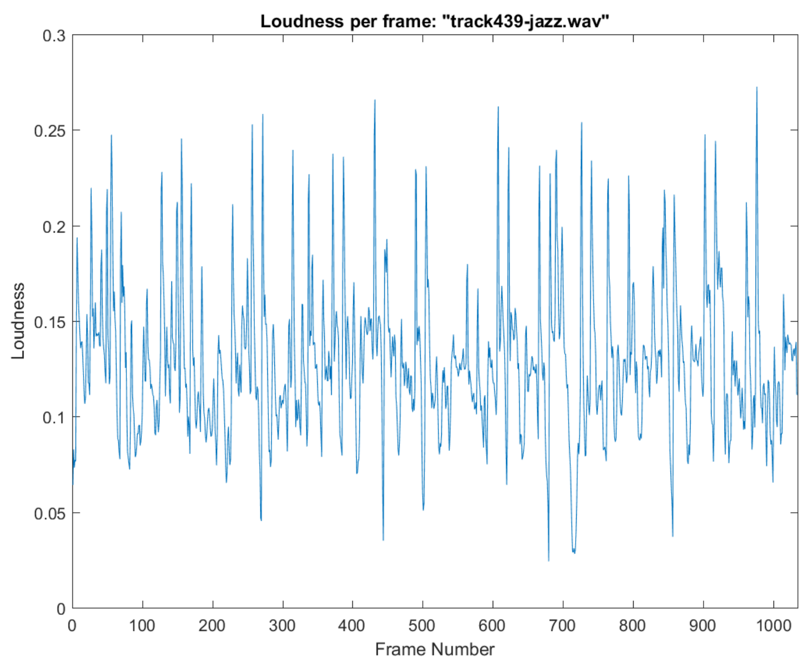


Figure 17: Loudness value per frame, jazz439

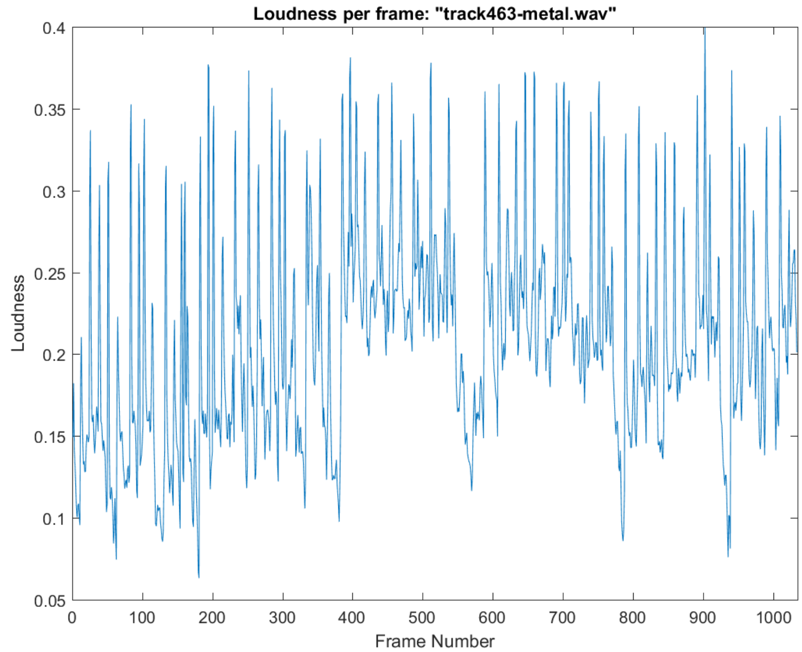


Figure 18: Loudness value per frame, metal463

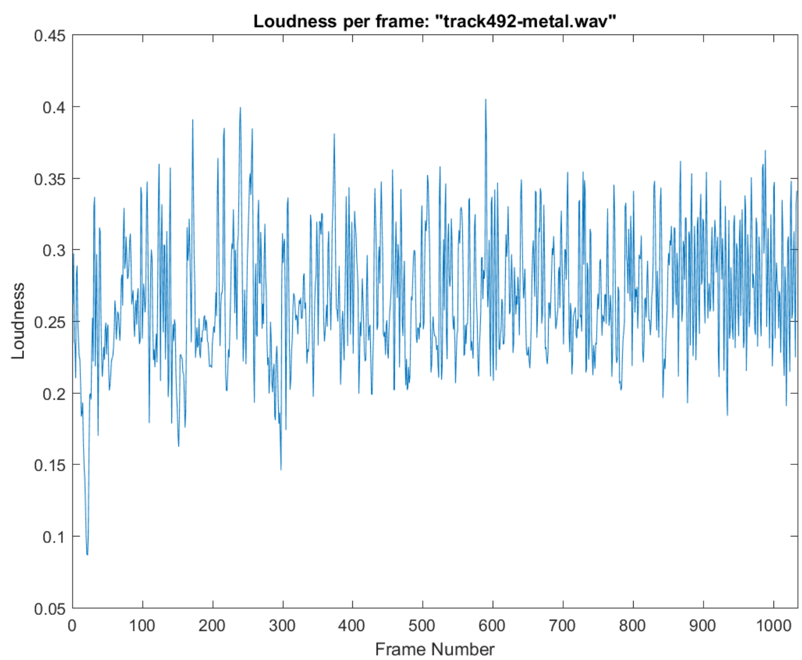


Figure 19: Loudness value per frame, metal492

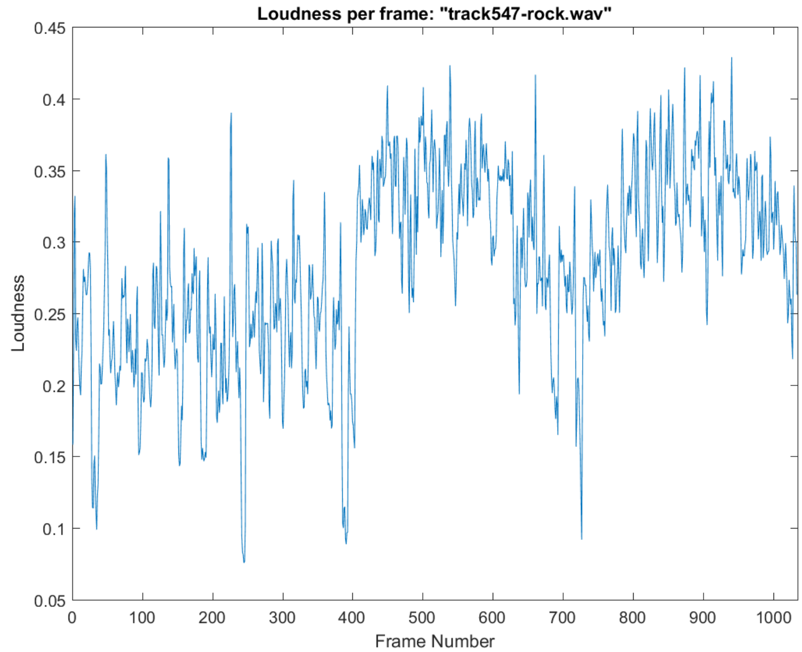


Figure 20: Loudness value per frame, rock547

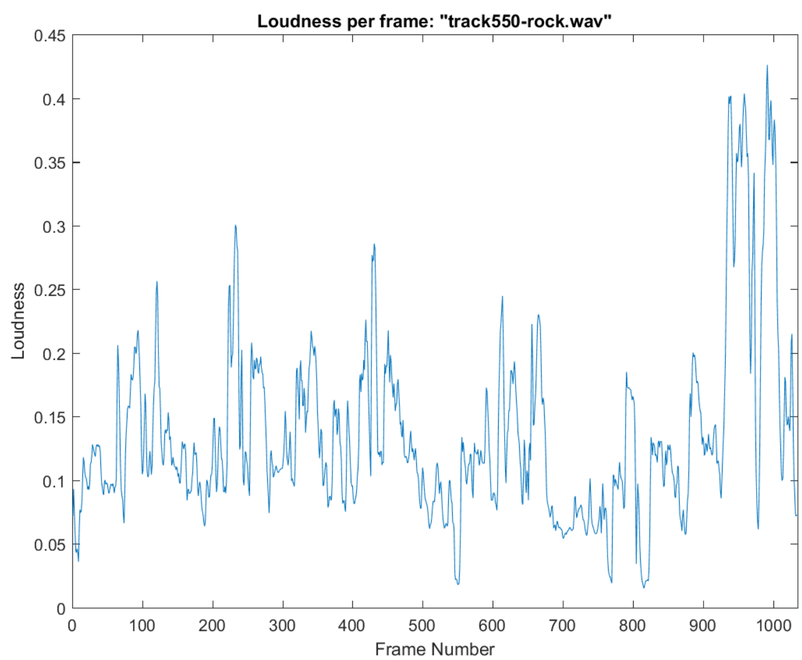


Figure 21: Loudness value per frame, rock550

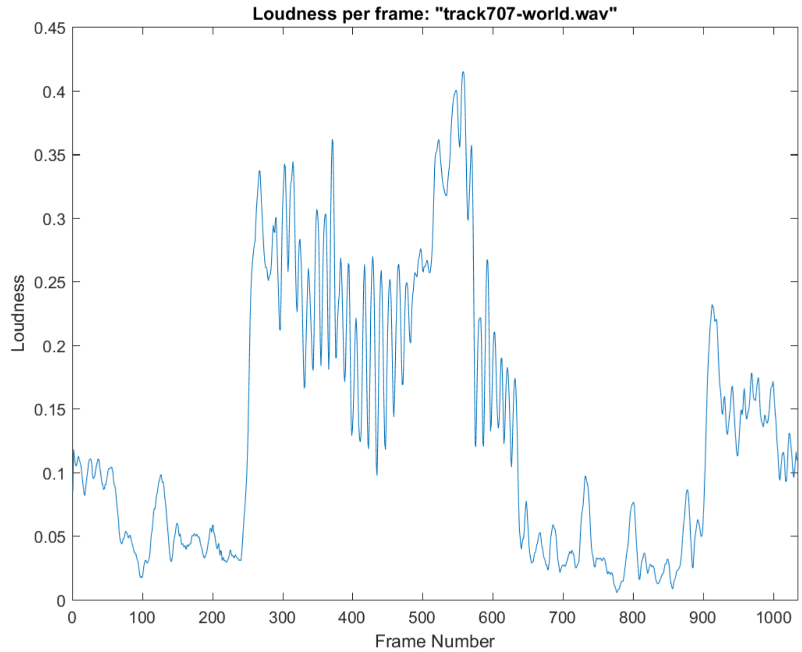


Figure 22: Loudness value per frame, world707

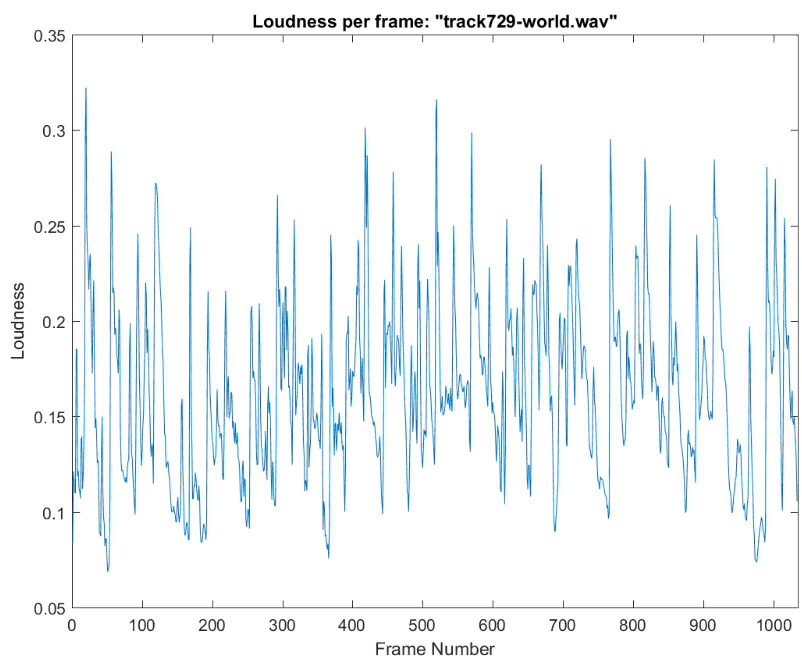


Figure 23: Loudness value per frame, world729

A.2 Zero-Cross Rate

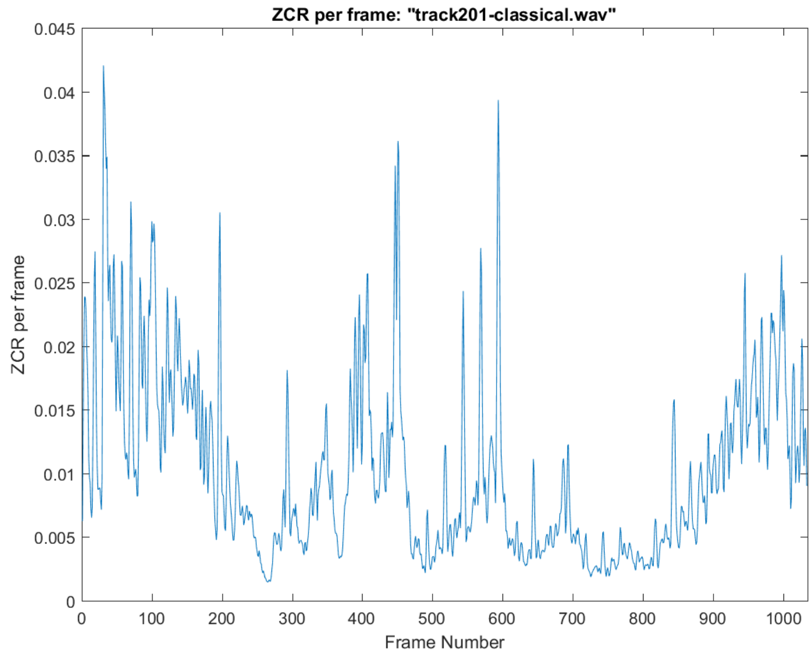


Figure 24: ZCR value per frame, classical201

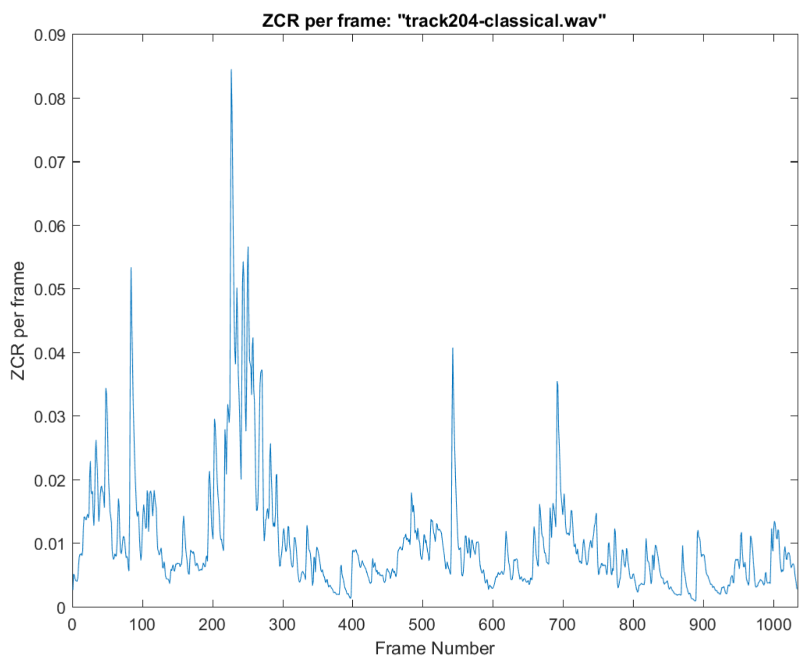


Figure 25: ZCR value per frame, classical204

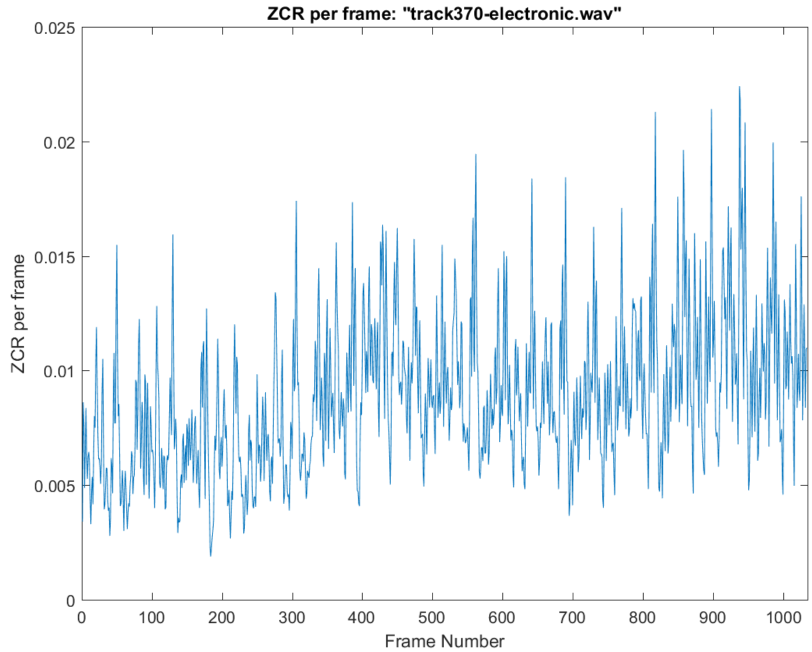


Figure 26: ZCR value per frame, electronic370

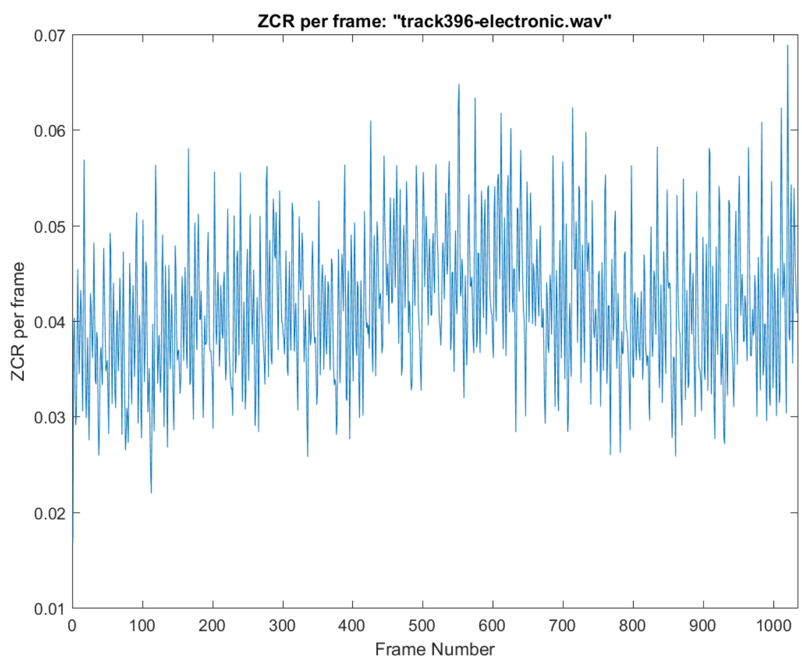


Figure 27: ZCR value per frame, electronic396

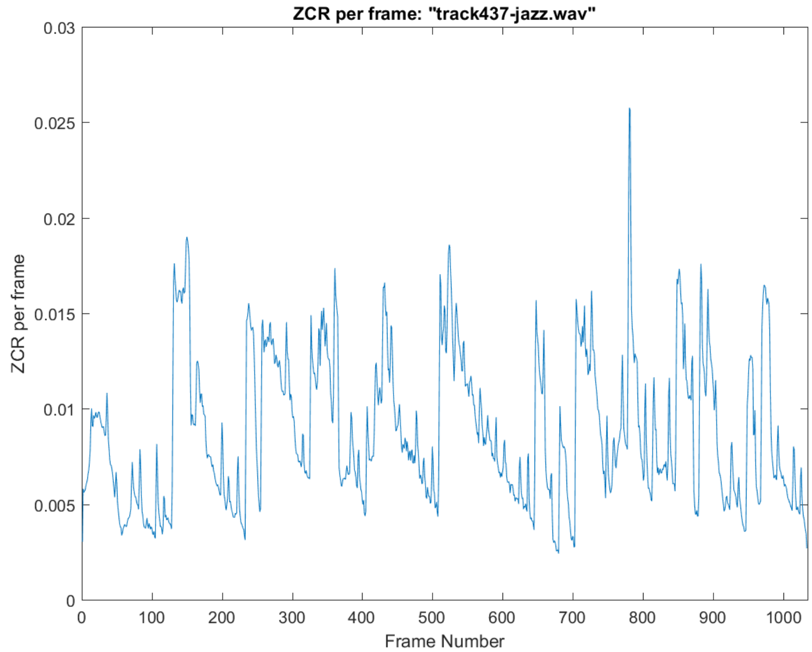


Figure 28: ZCR value per frame, jazz437

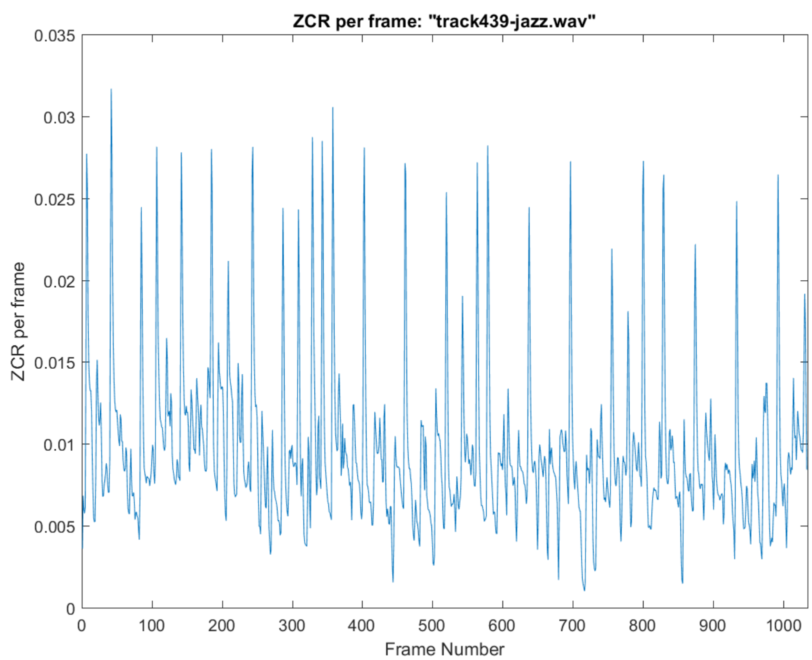


Figure 29: ZCR value per frame, jazz439

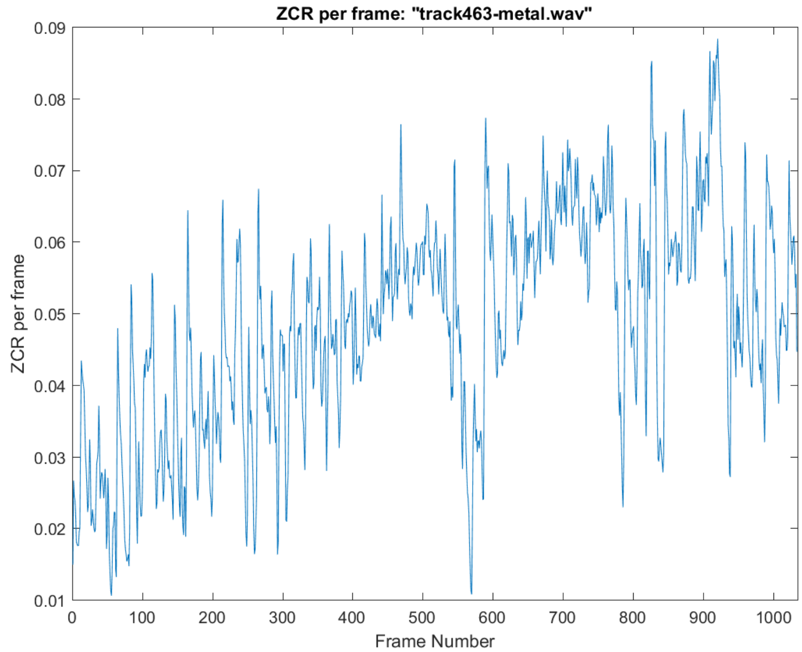


Figure 30: ZCR value per frame, metal463

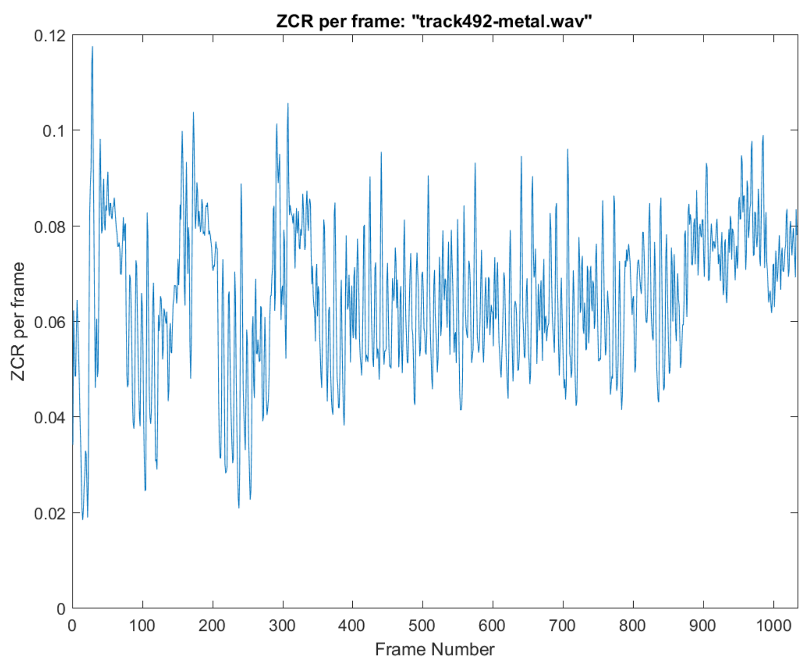


Figure 31: ZCR value per frame, metal492

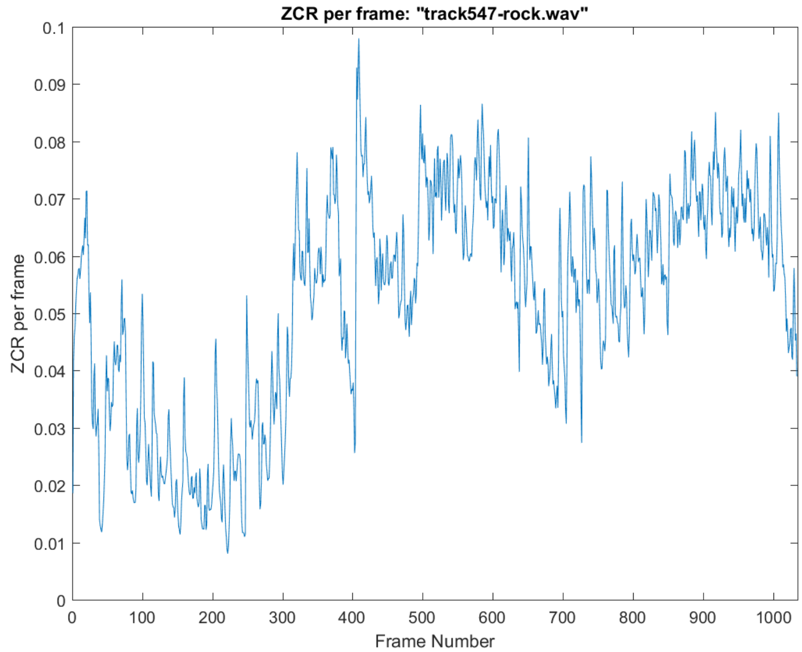


Figure 32: ZCR value per frame, rock547

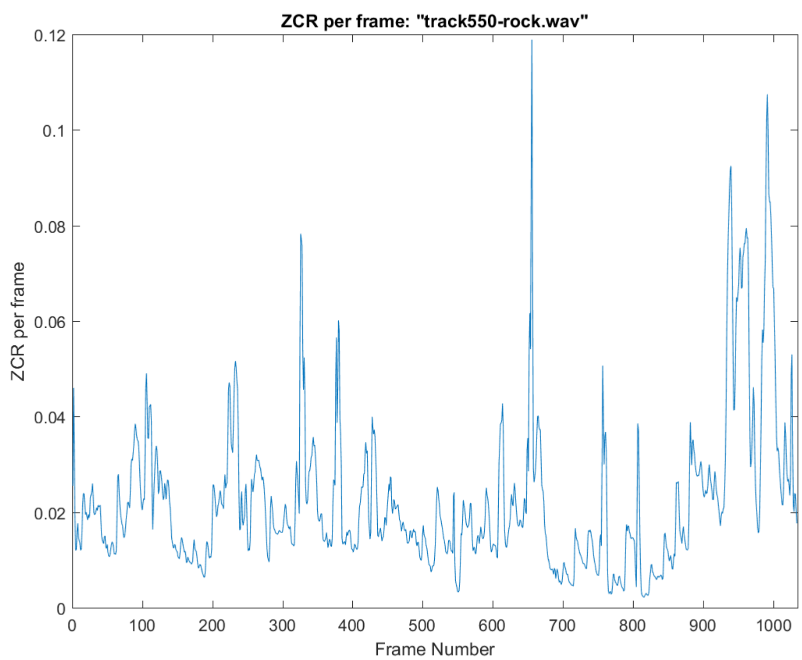


Figure 33: ZCR value per frame, rock550

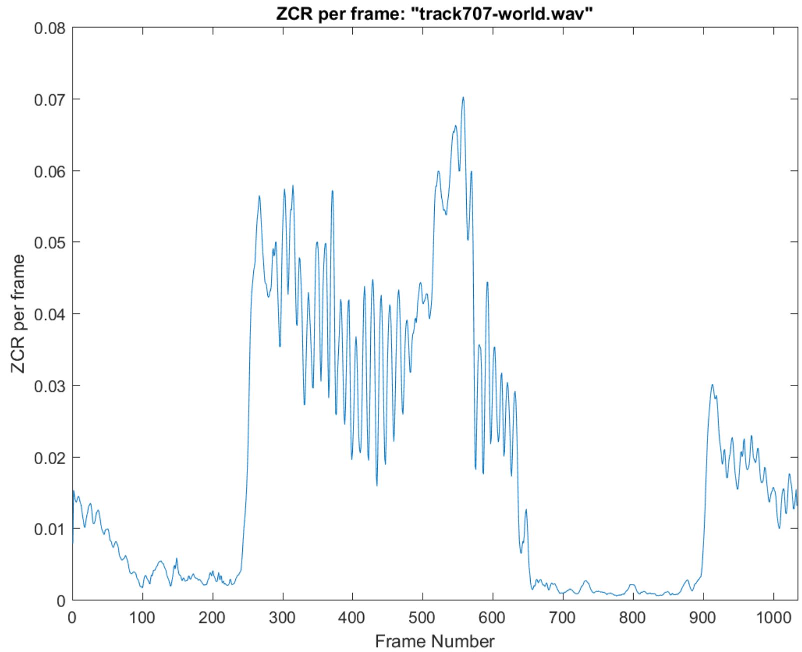


Figure 34: ZCR value per frame, world707

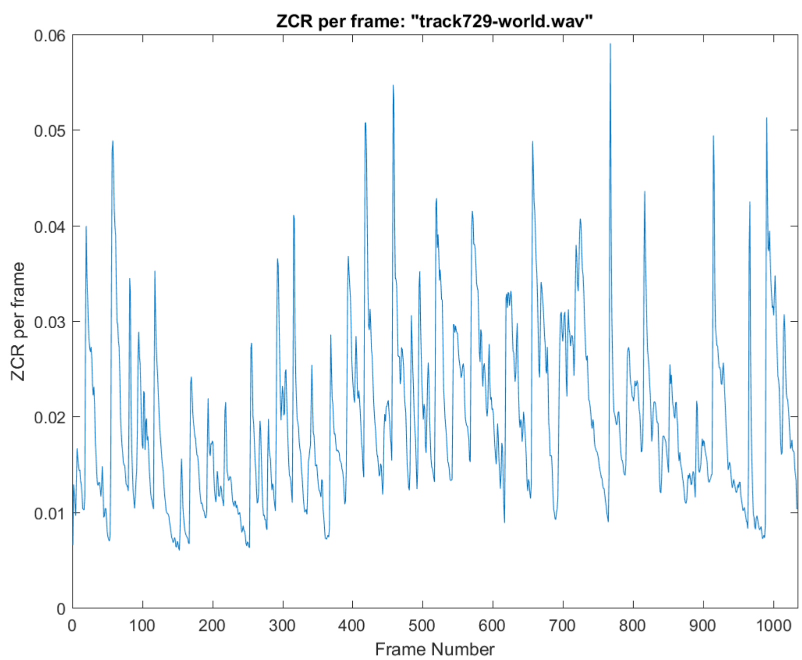


Figure 35: ZCR value per frame, world729

A.3 Windows

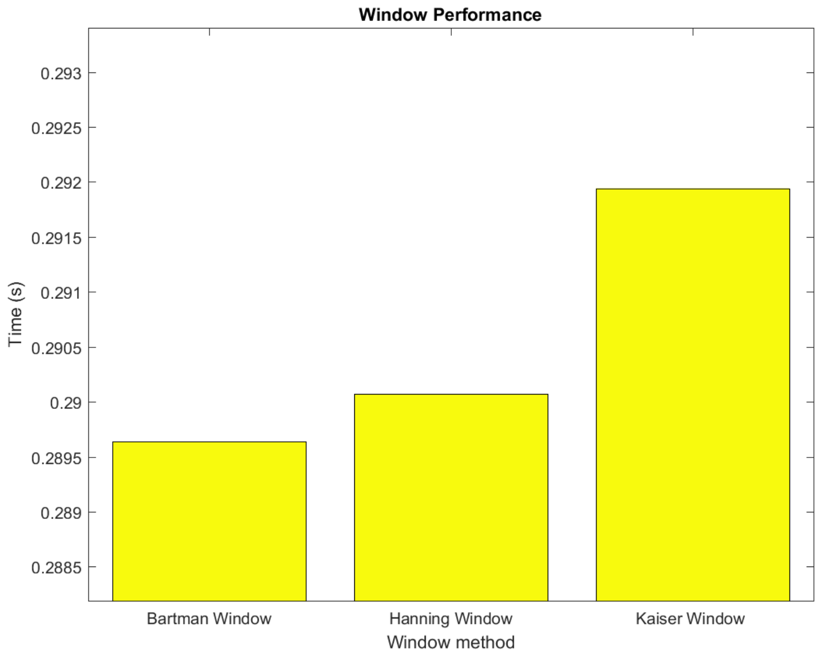


Figure 36: Timing comparison of different windows

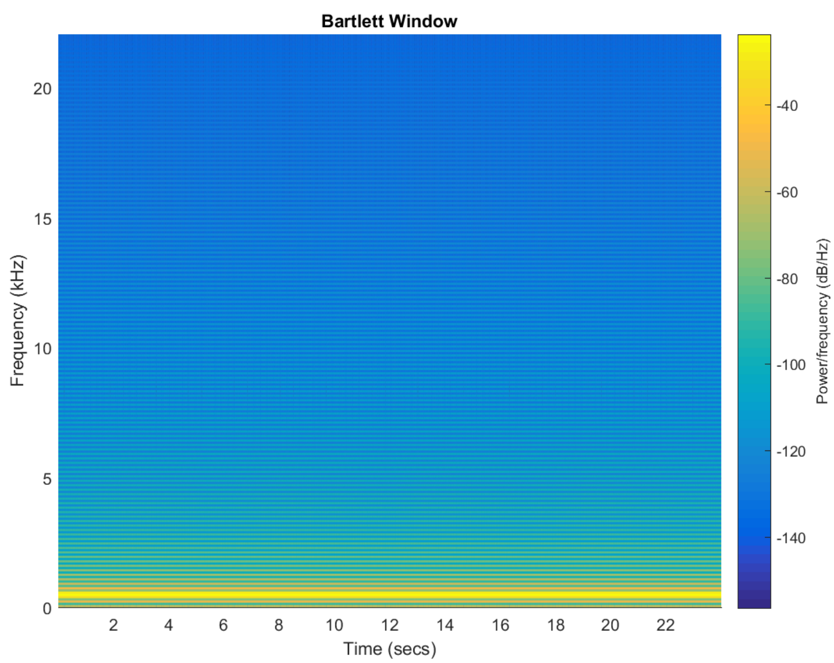


Figure 37: Spectrogram of Bartlett Window

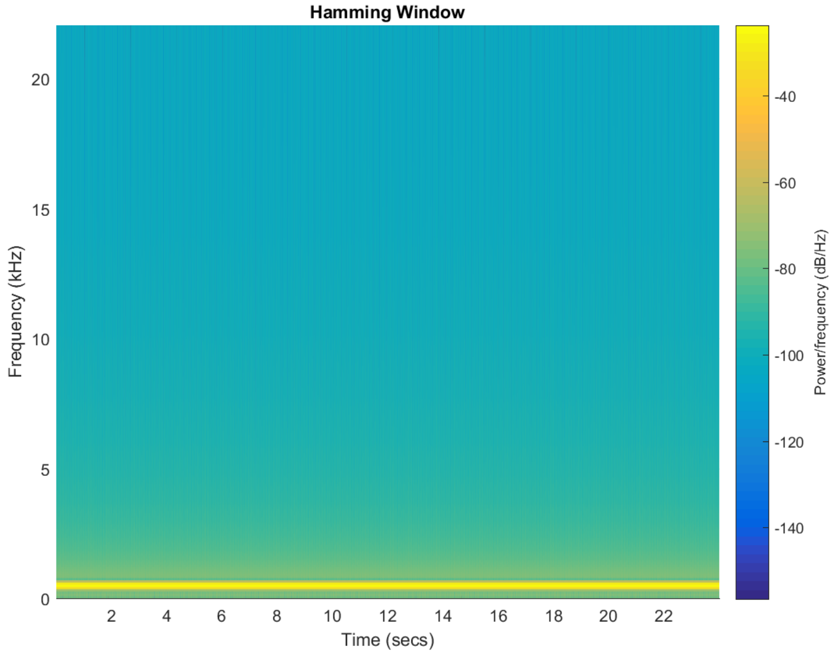


Figure 38: Spectrogram of Hamming Window

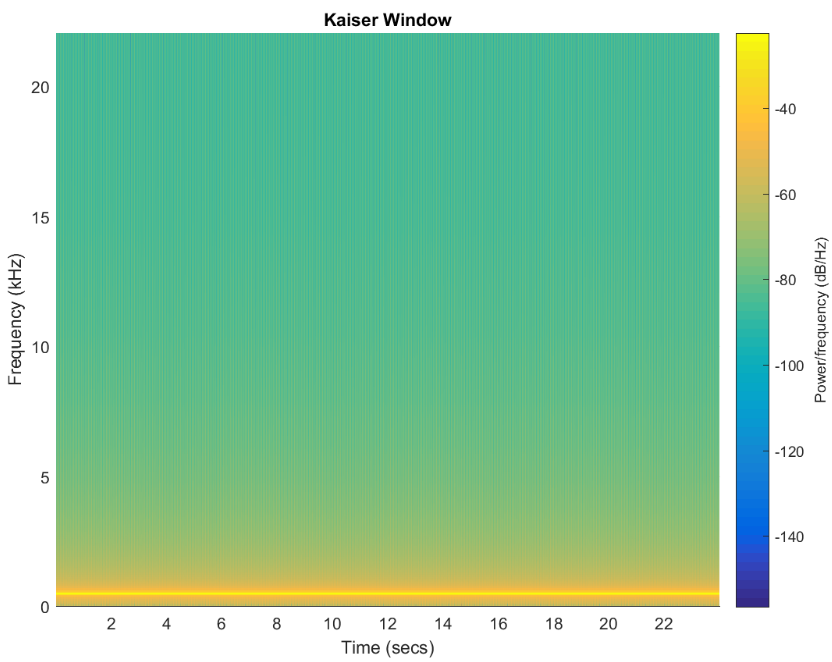


Figure 39: Spectrogram of Kaiser Window

A.4 Centroid

;

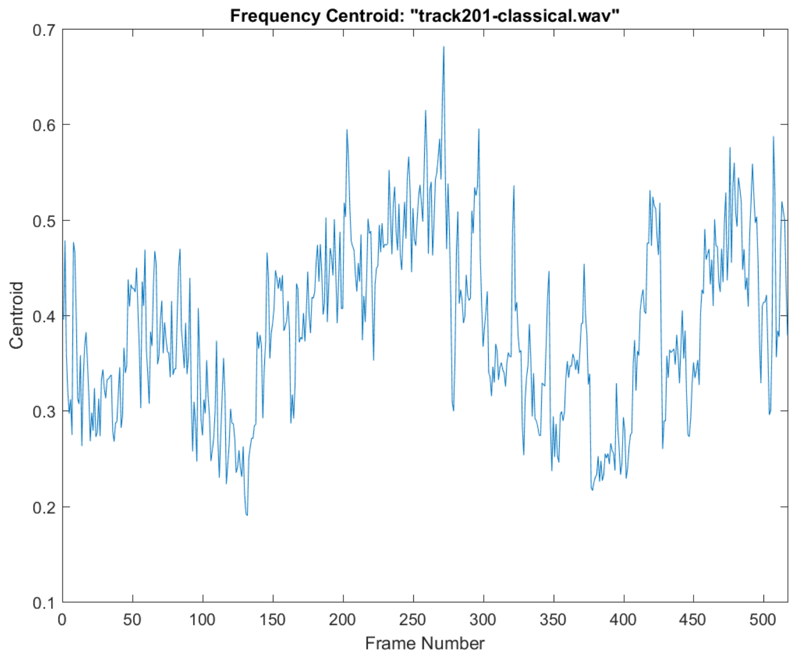


Figure 40: Frequency Centroid

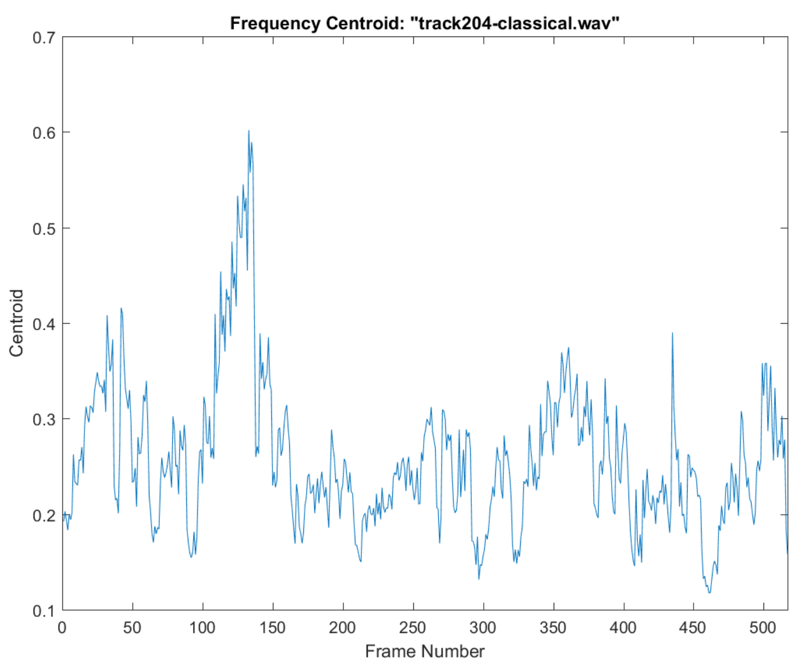


Figure 41: Frequency Centroid

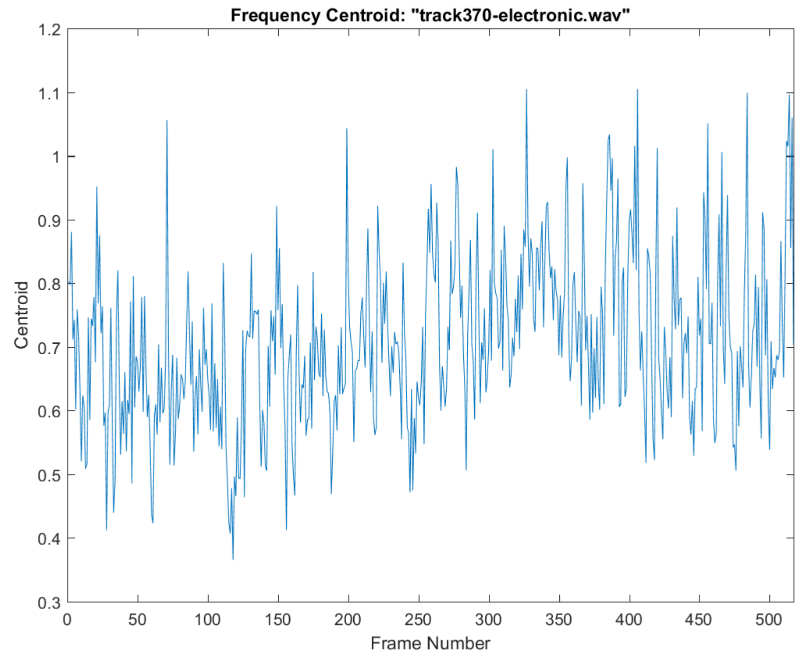


Figure 42: Frequency Centroid

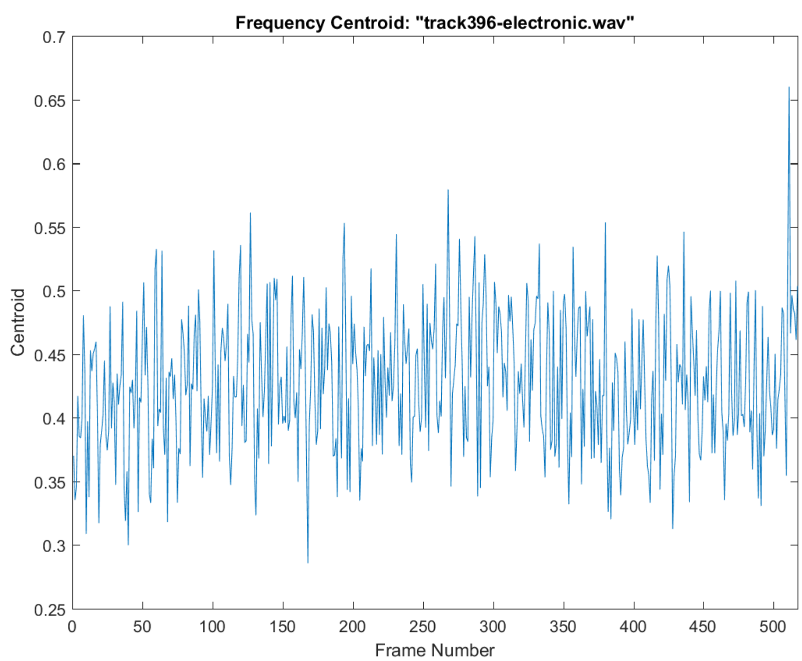


Figure 43: Frequency Centroid

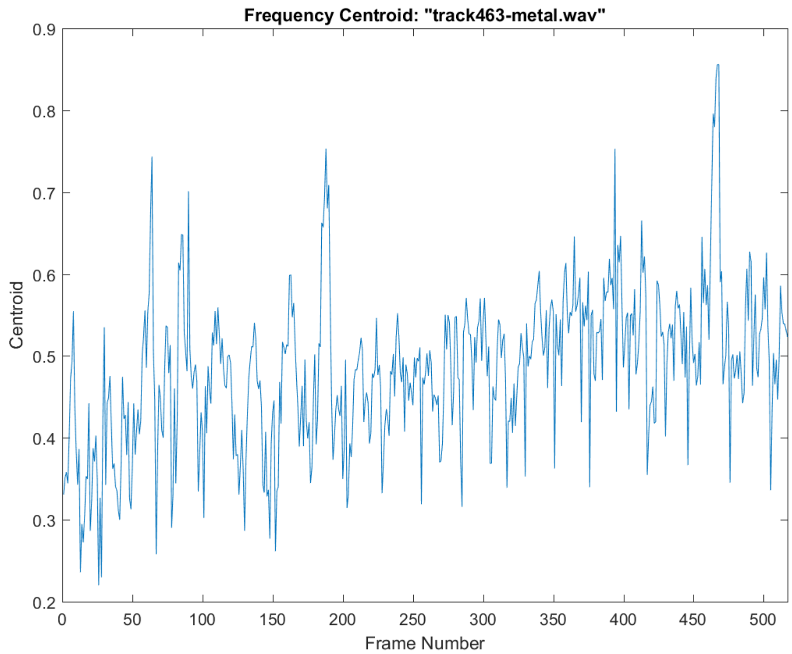


Figure 44: Frequency Centroid

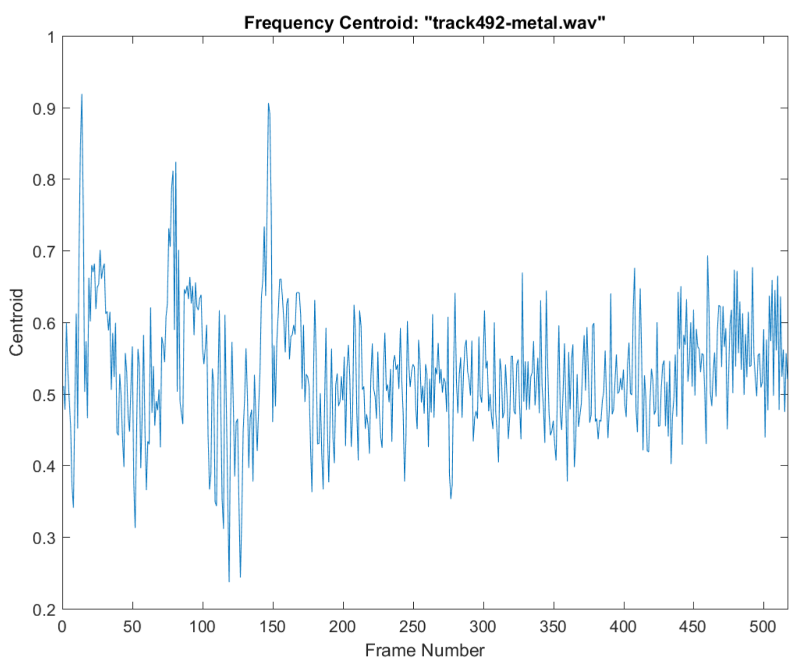


Figure 45: Frequency Centroid

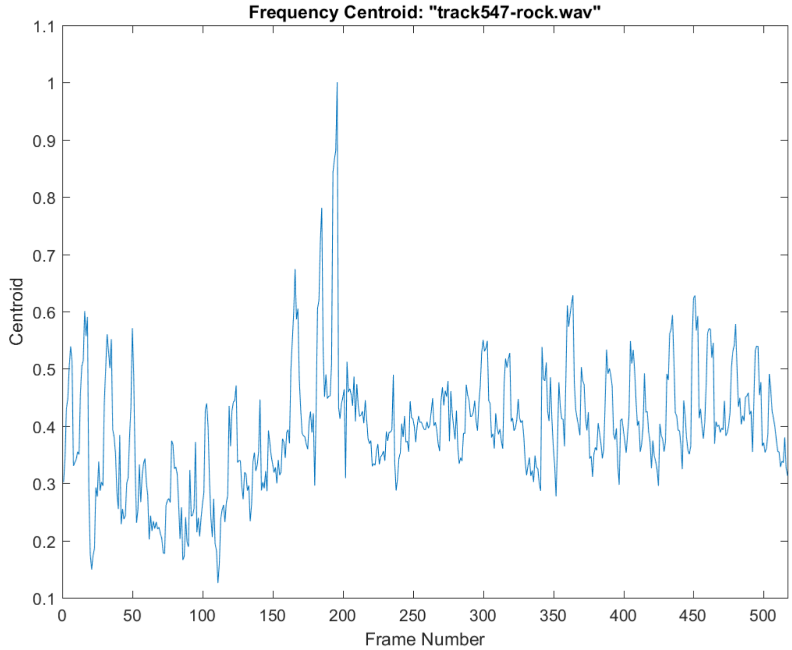


Figure 46: Frequency Centroid

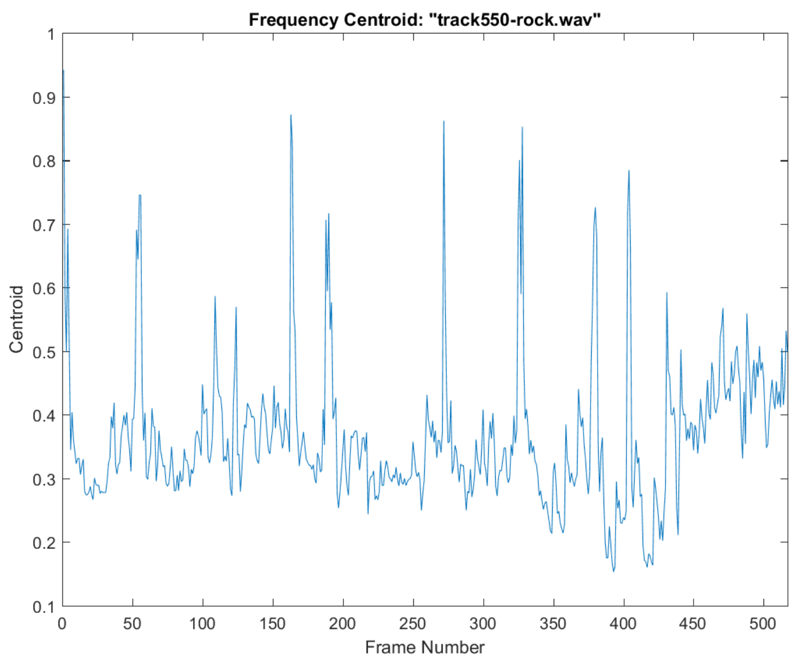


Figure 47: Frequency Centroid

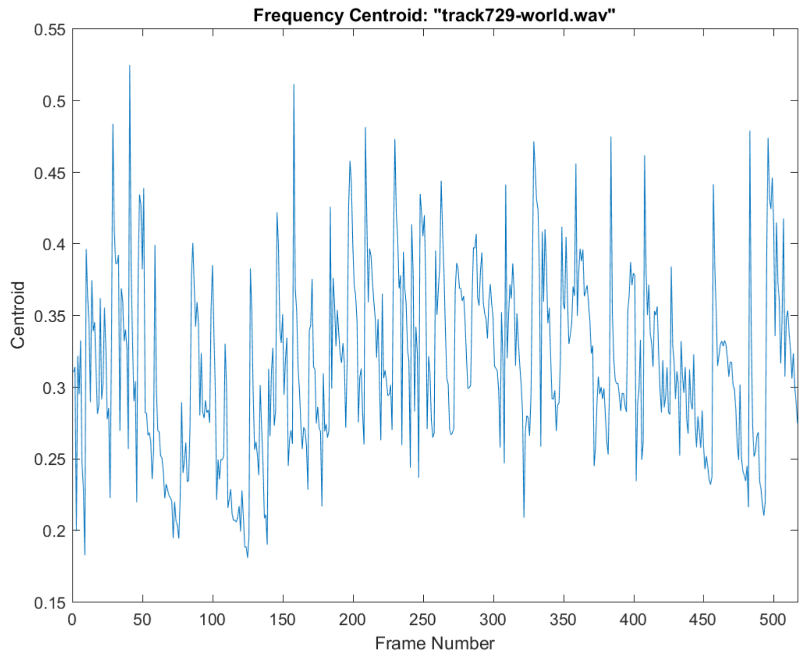


Figure 48: Frequency Centroid

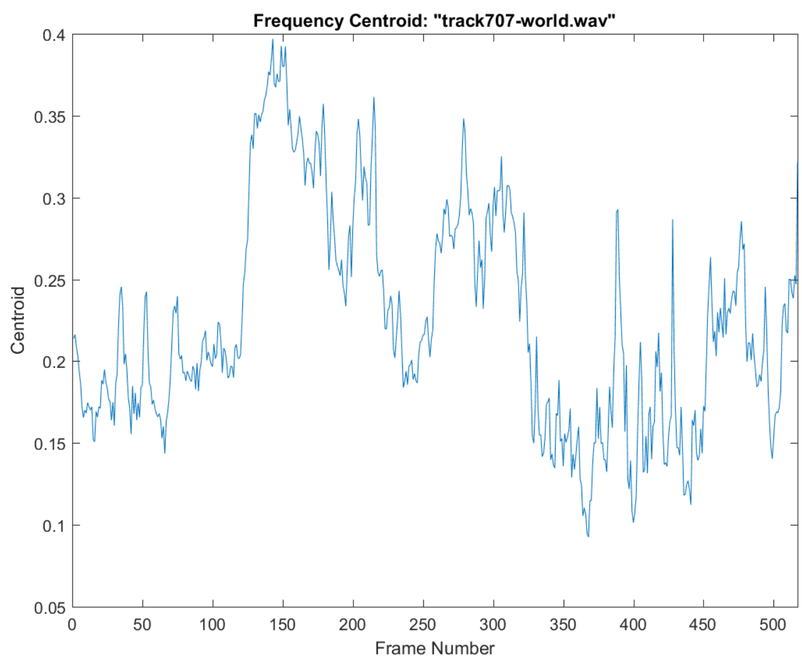


Figure 49: Frequency Centroid

A.5 Spread

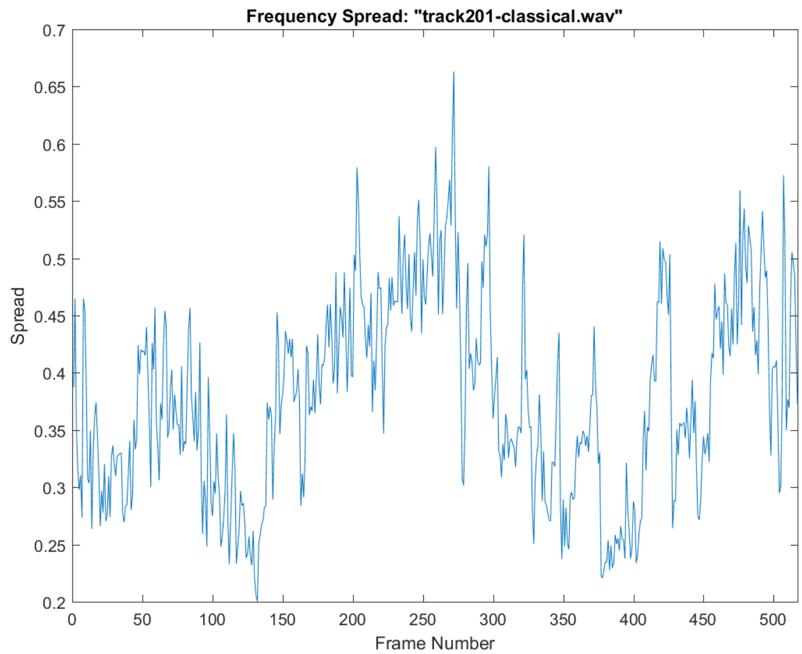


Figure 50: Frequency Spread

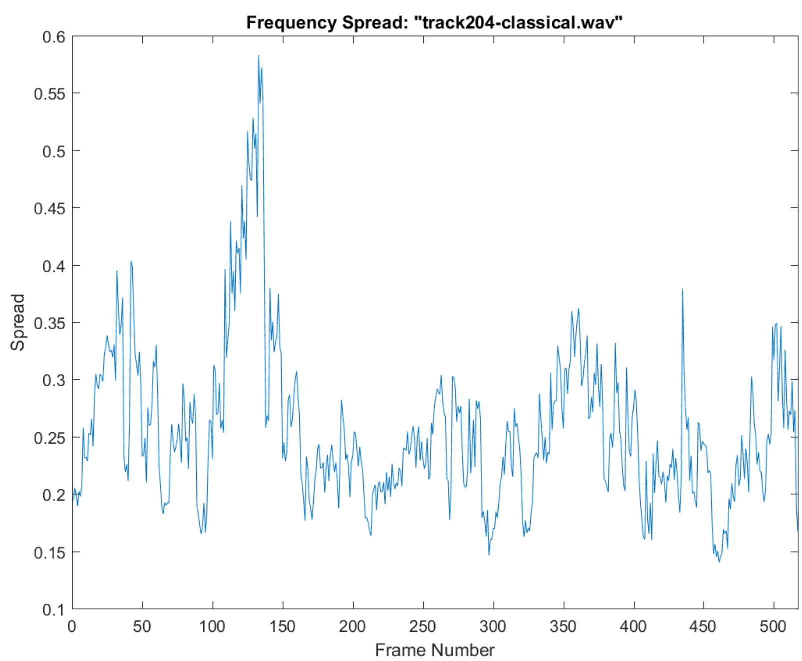


Figure 51: Frequency Spread

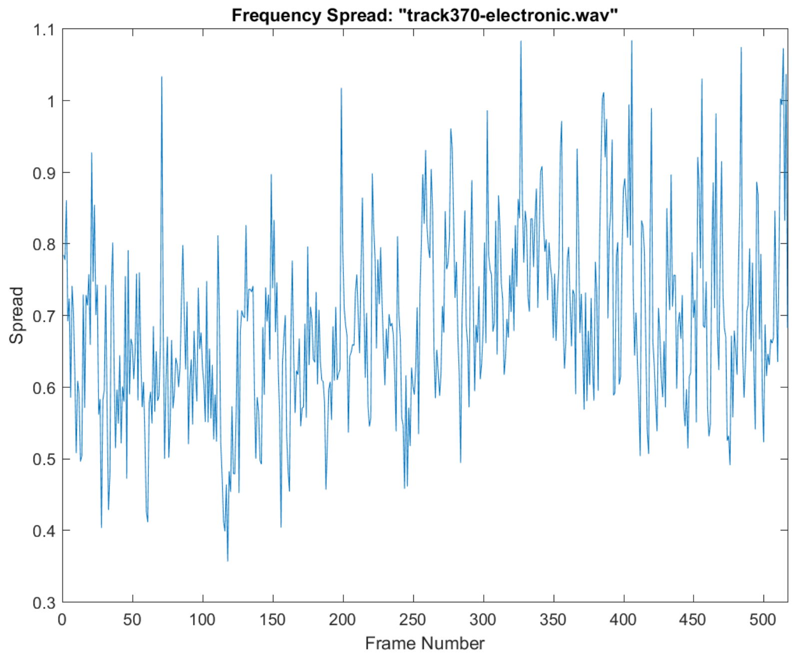


Figure 52: Frequency Spread

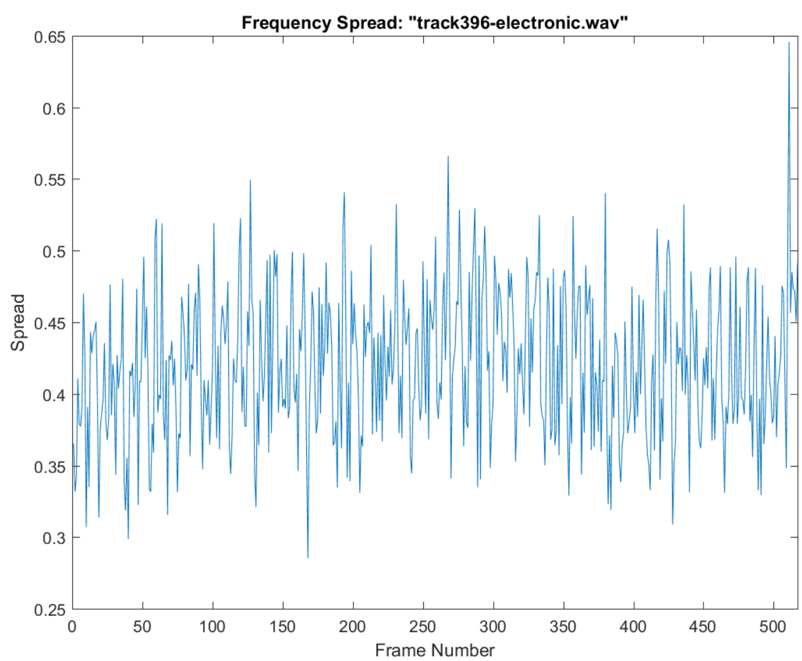


Figure 53: Frequency Spread

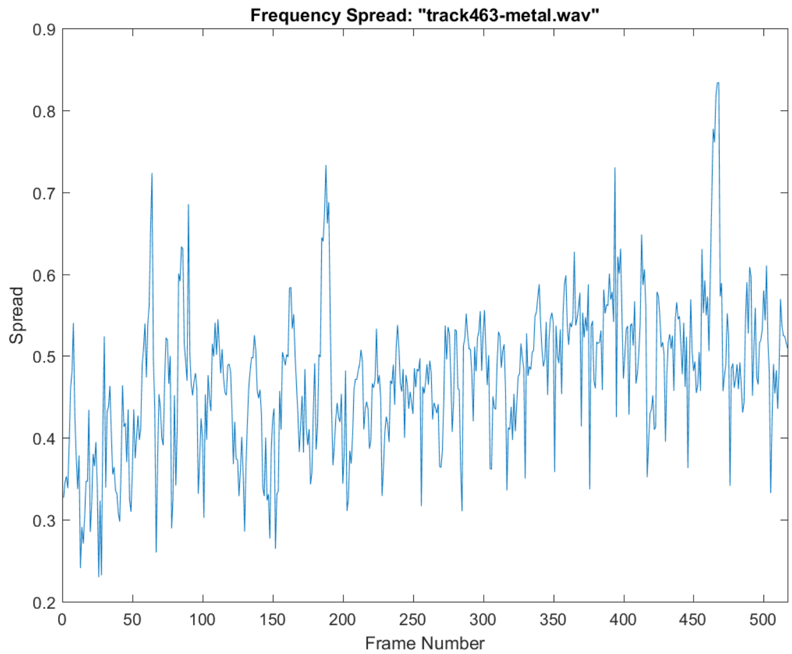


Figure 54: Frequency Spread

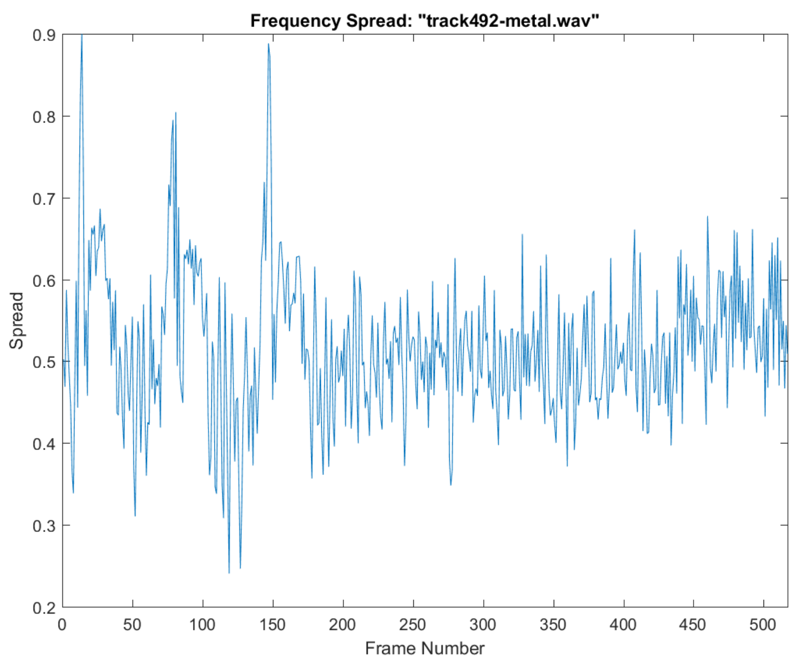


Figure 55: Frequency Spread

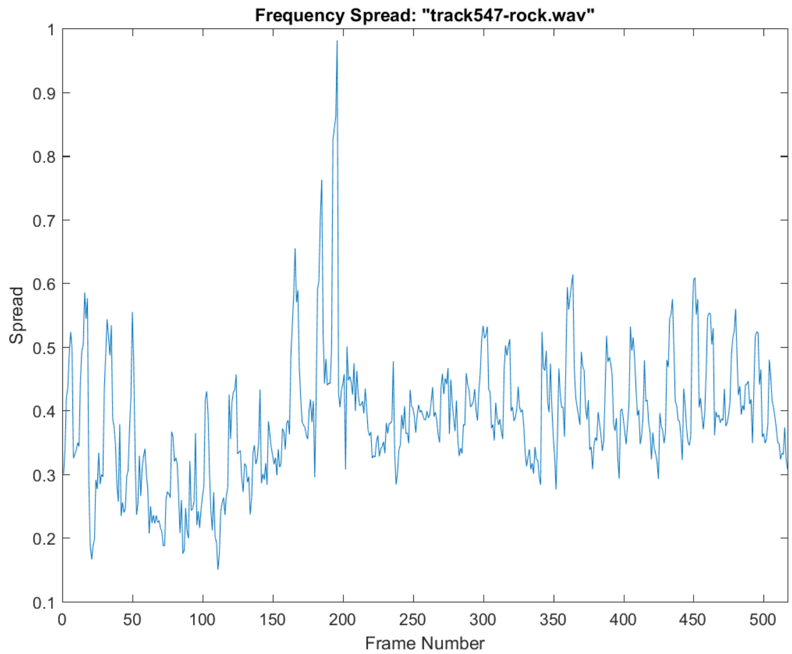


Figure 56: Frequency Spread

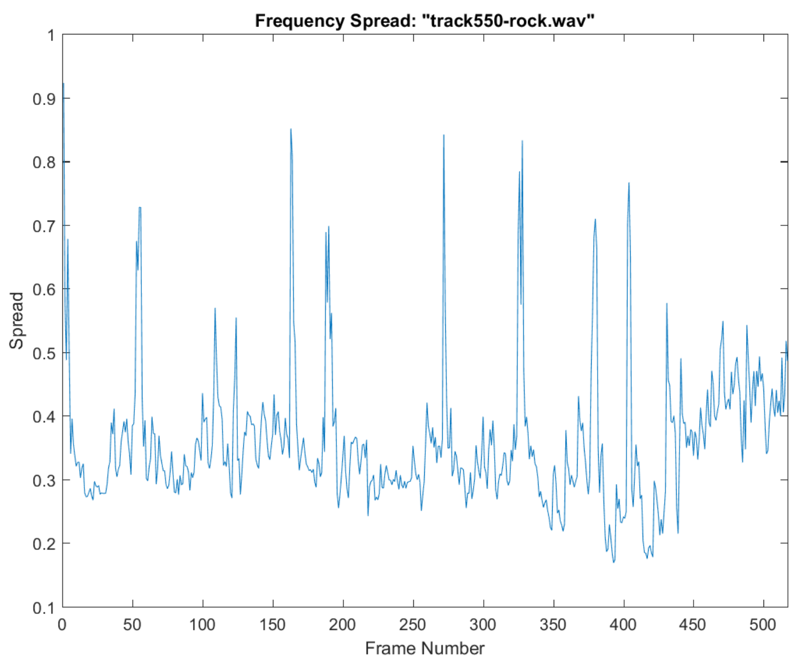


Figure 57: Frequency Spread

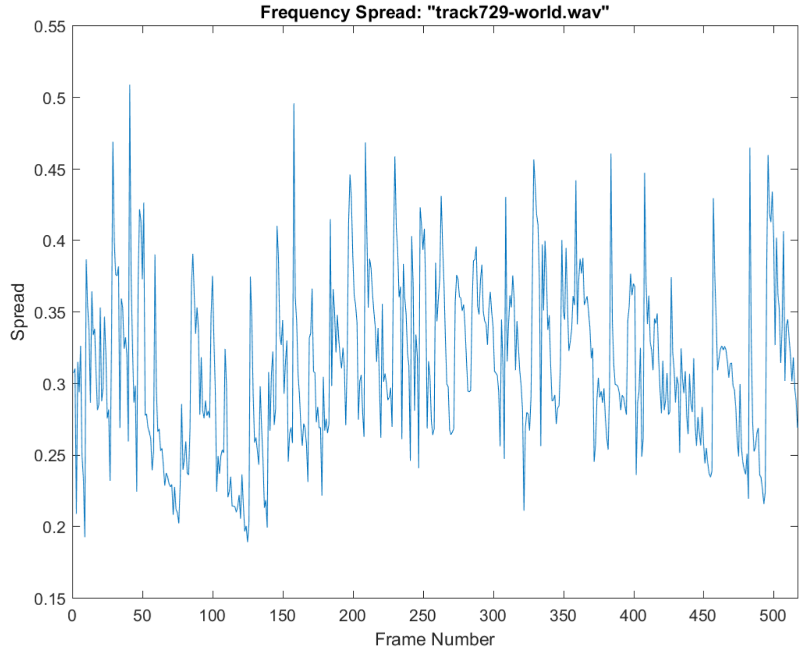


Figure 58: Frequency Spread

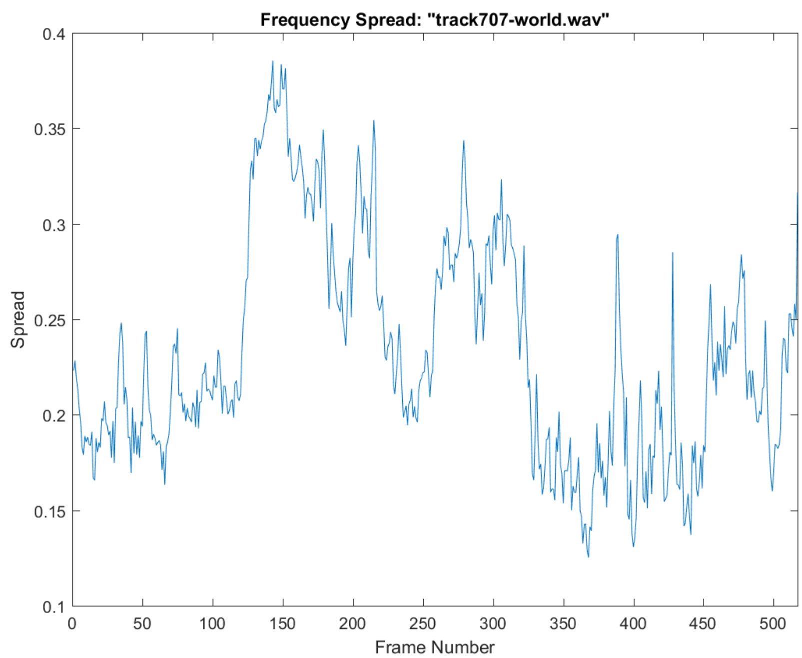


Figure 59: Frequency Spread

A.6 Flatness

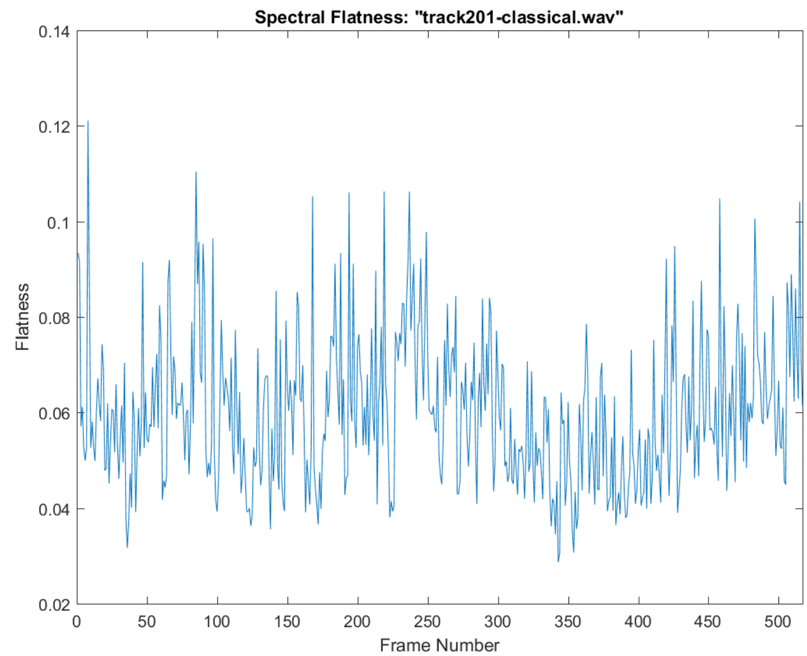


Figure 60: Spectral Flatness

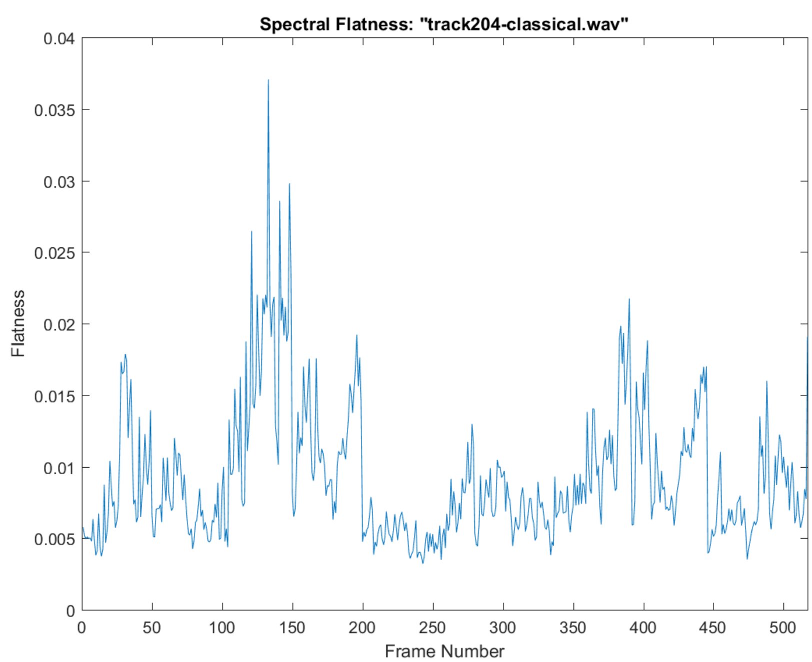


Figure 61: Spectral Flatness

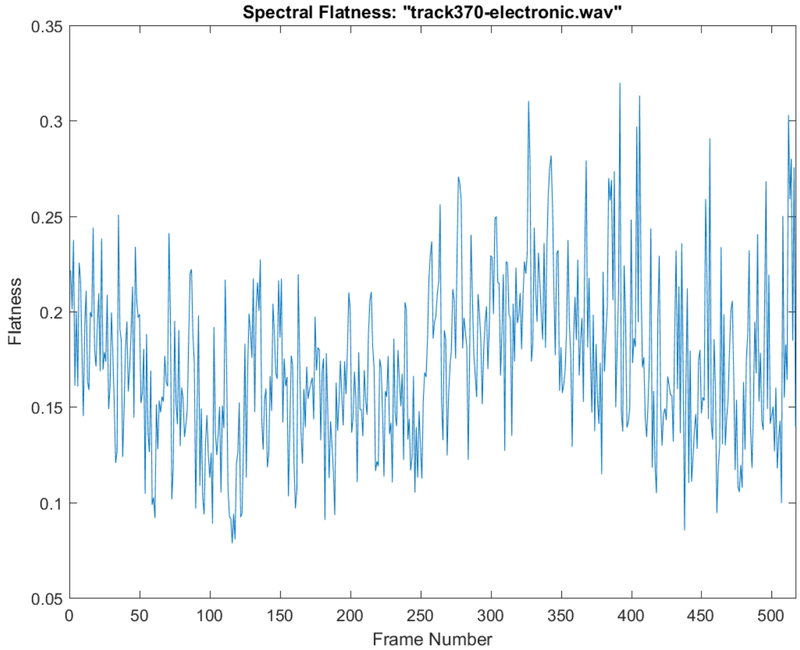


Figure 62: Spectral Flatness

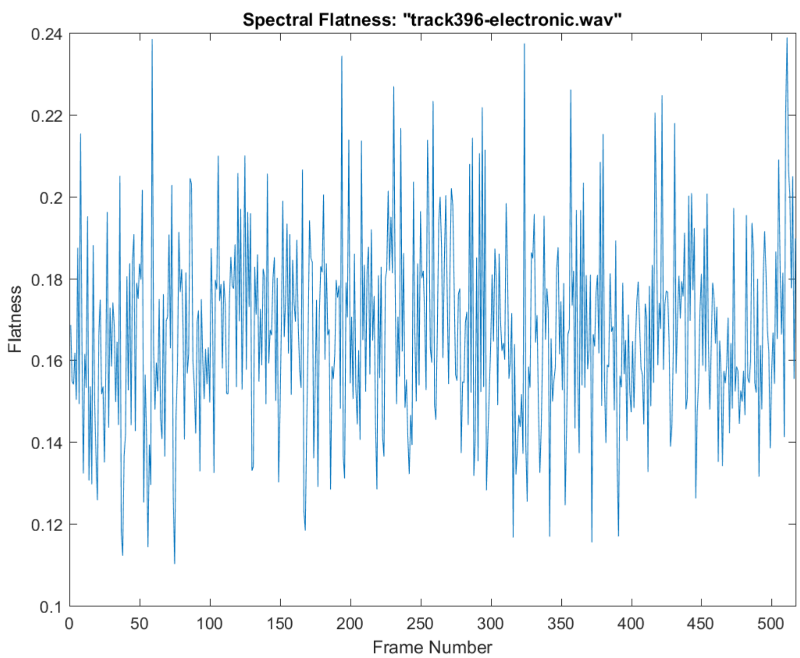


Figure 63: Spectral Flatness

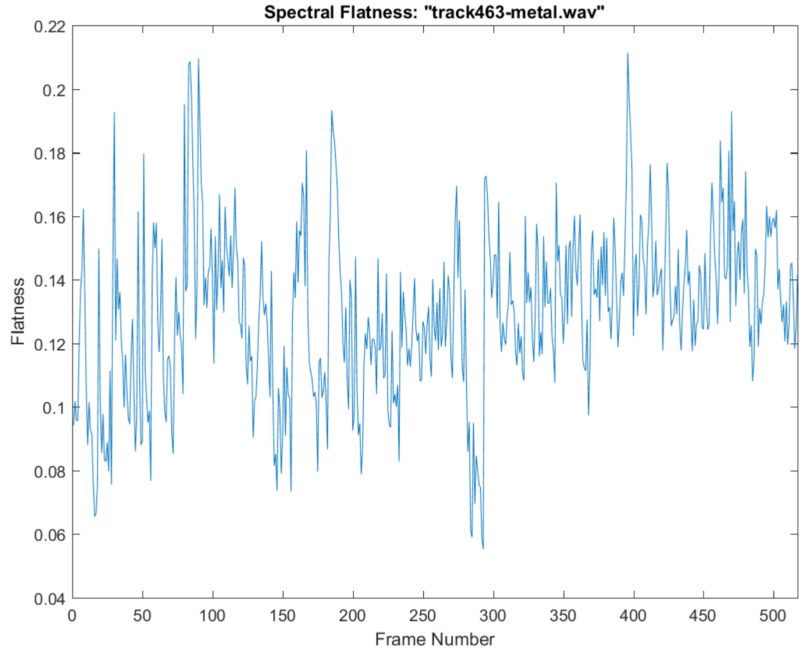


Figure 64: Spectral Flatness

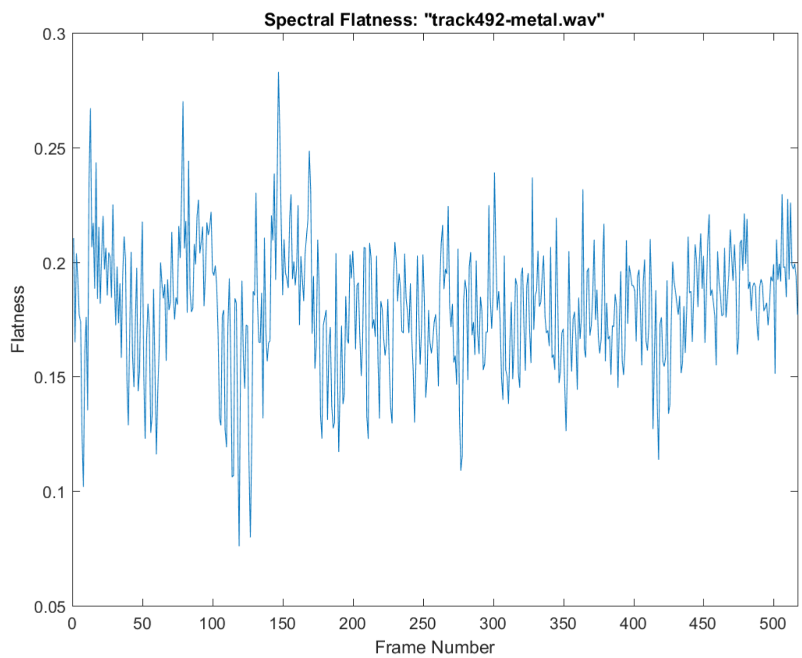


Figure 65: Spectral Flatness

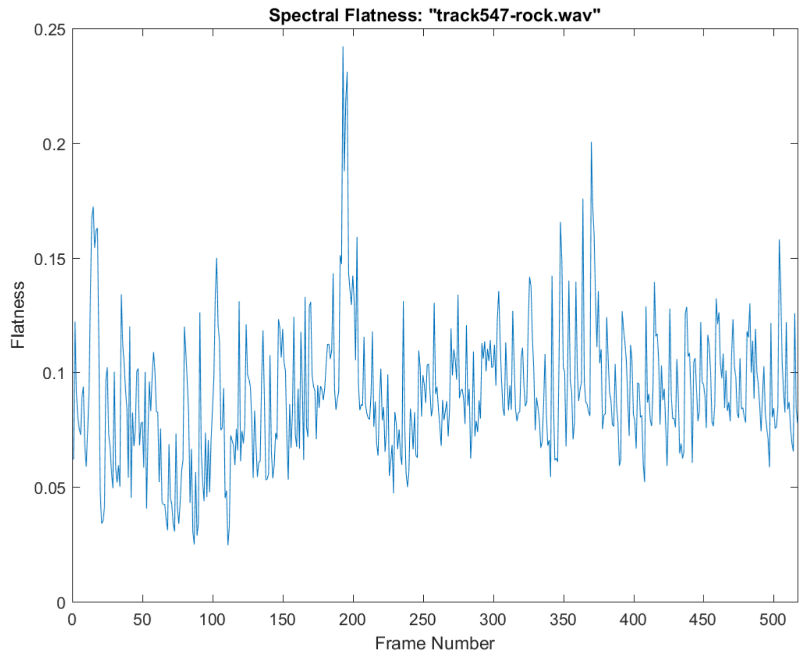


Figure 66: Spectral Flatness

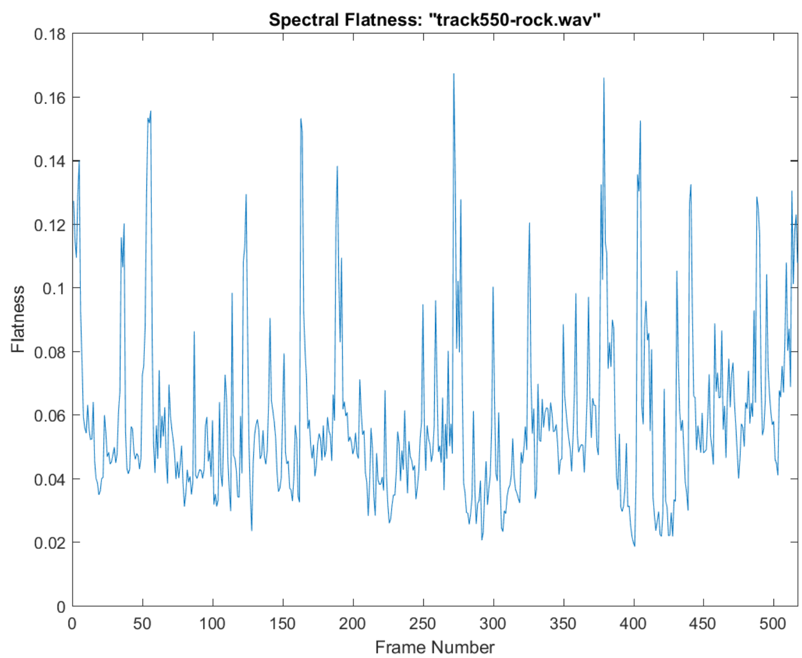


Figure 67: Spectral Flatness

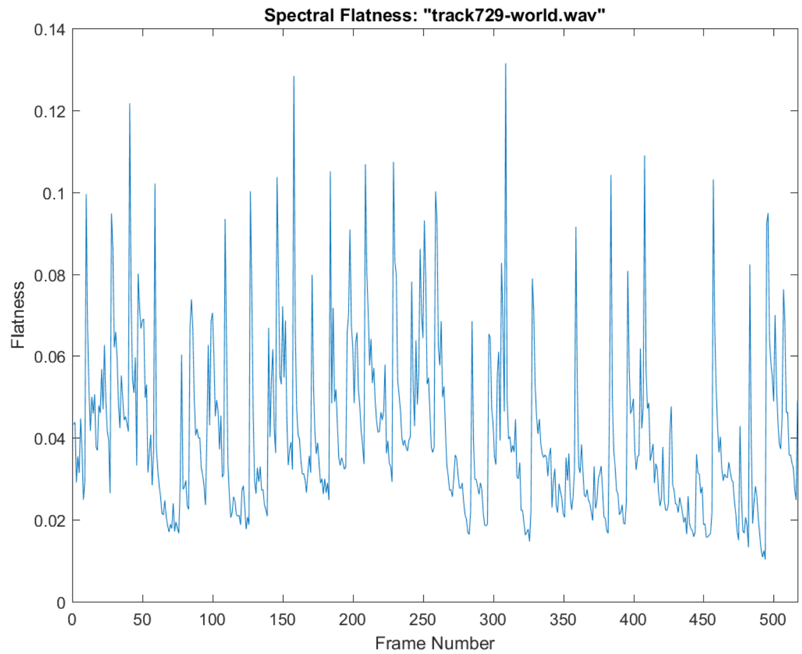


Figure 68: Spectral Flatness

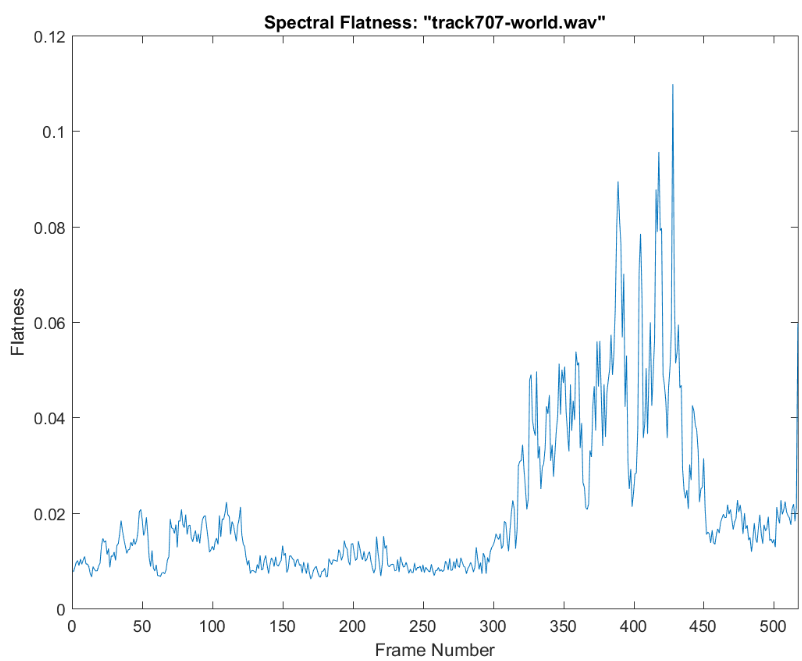


Figure 69: Spectral Flatness

A.7 Flux

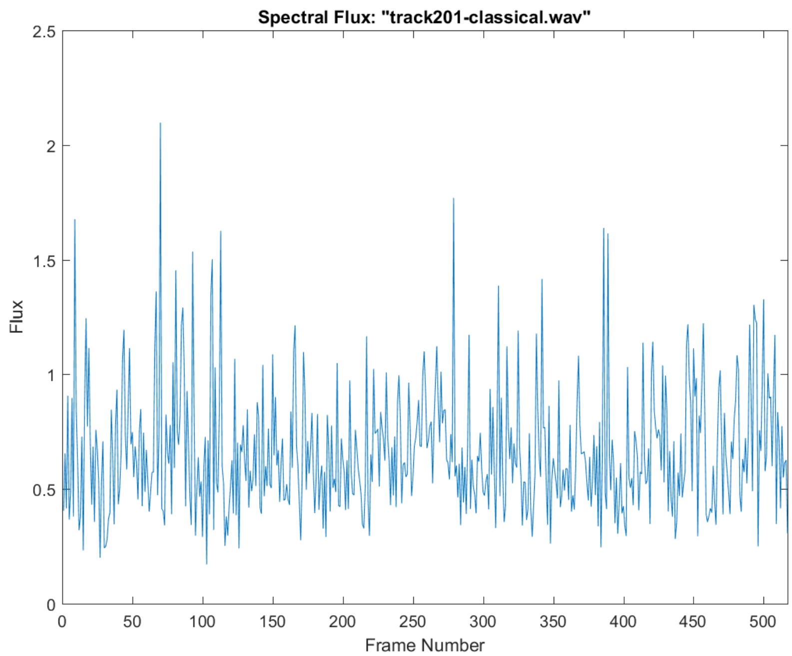


Figure 70: Spectral Flux

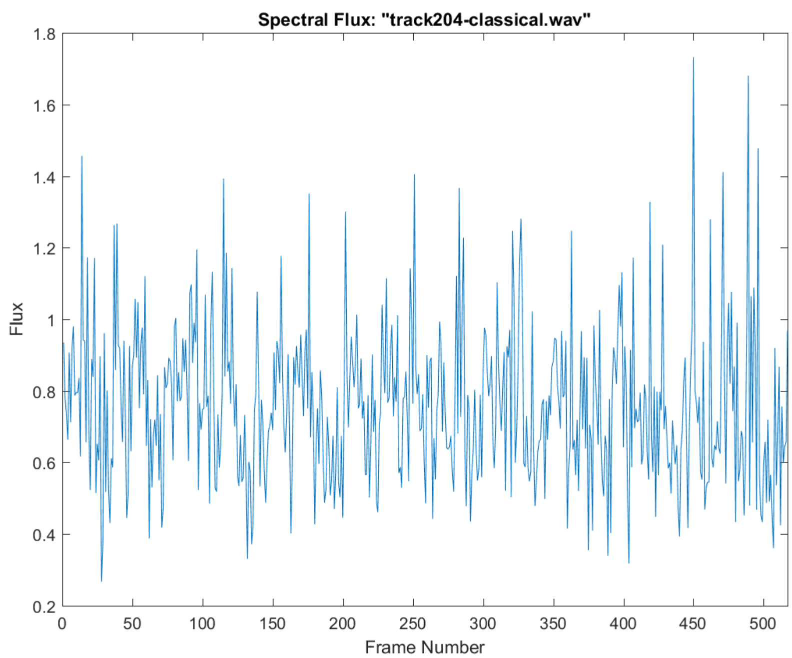


Figure 71: Spectral Flux

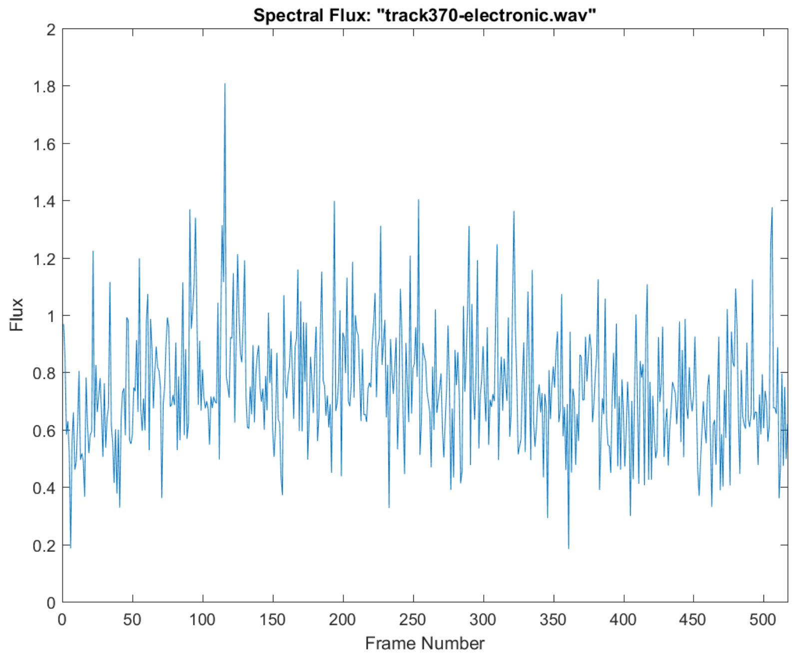


Figure 72: Spectral Flux

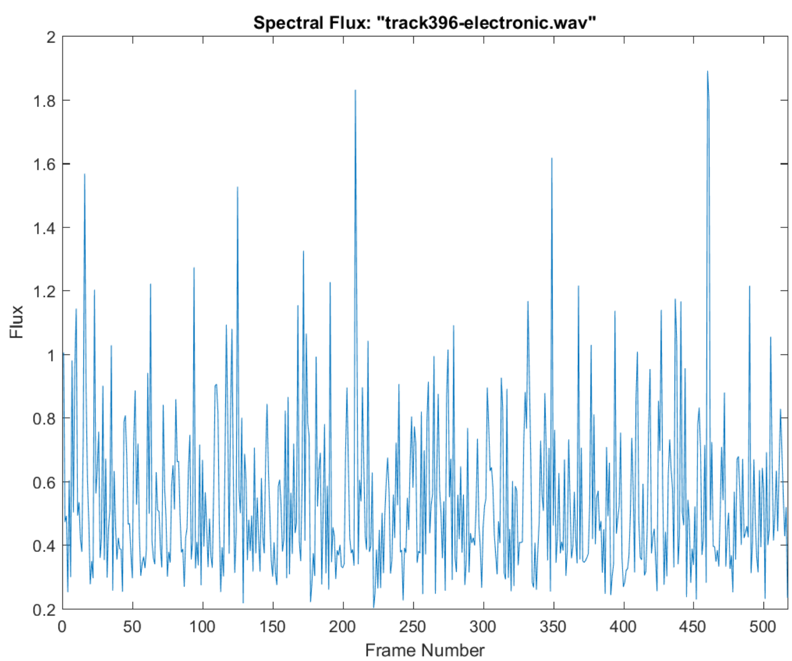


Figure 73: Spectral Flux

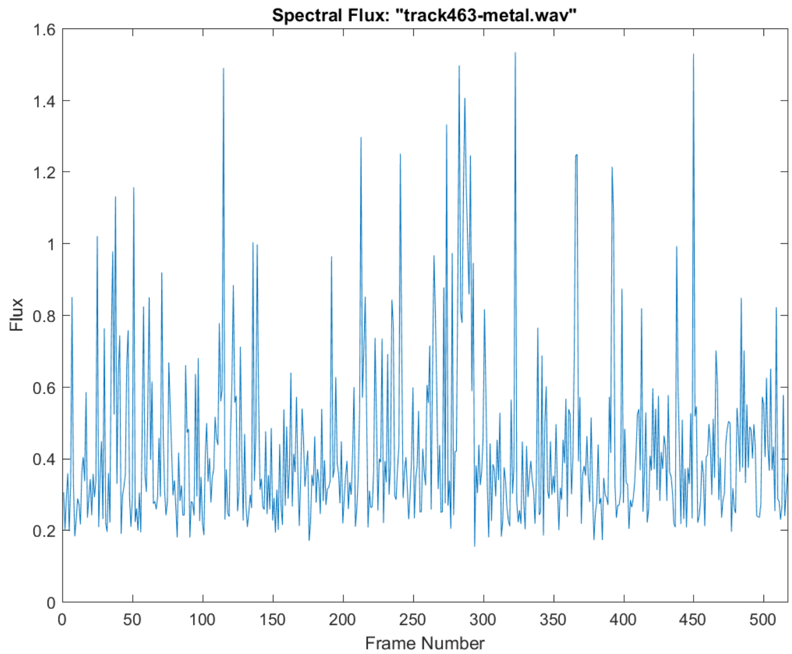


Figure 74: Spectral Flux

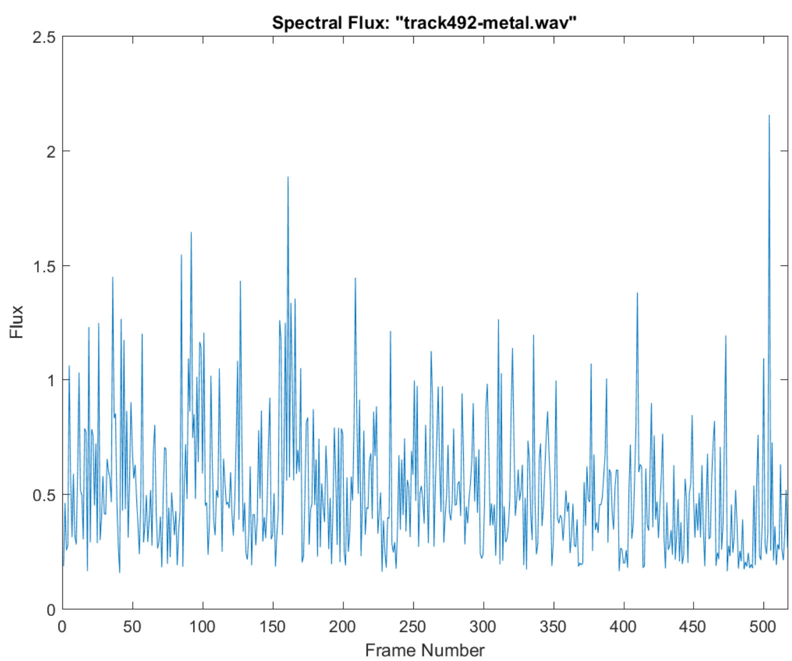


Figure 75: Spectral Flux

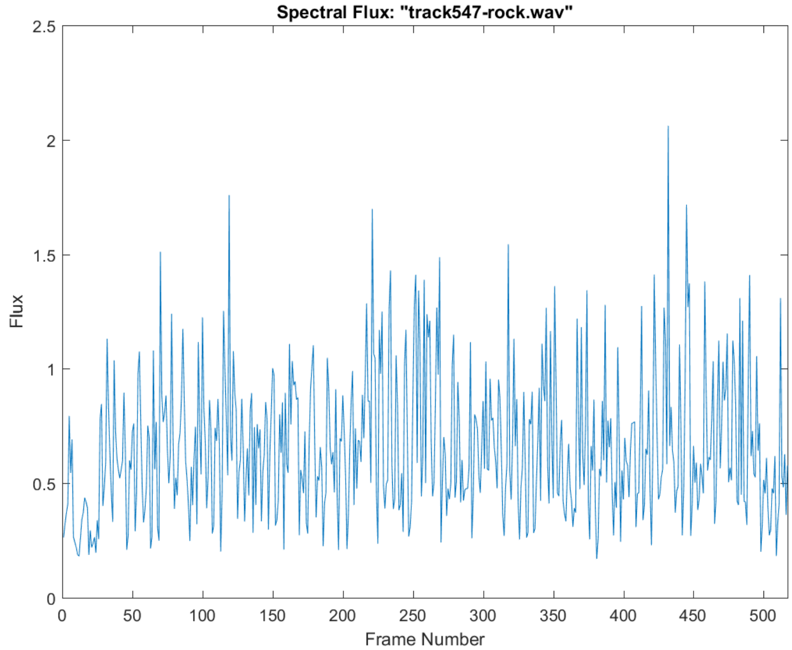


Figure 76: Spectral Flux

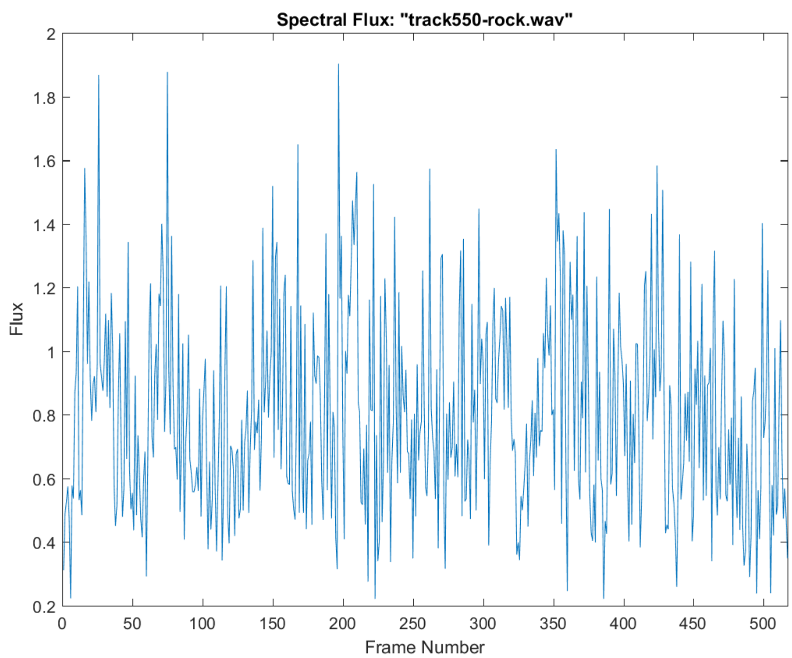


Figure 77: Spectral Flux

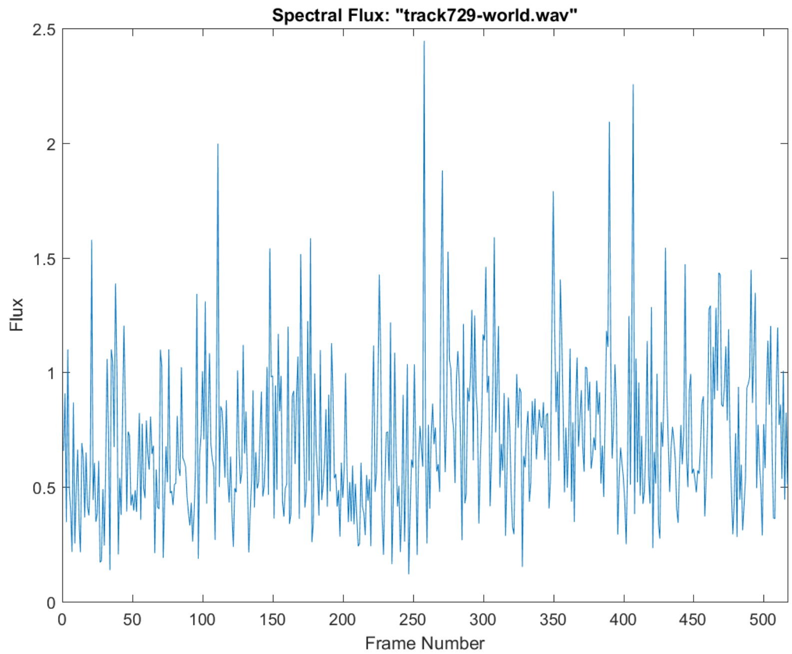


Figure 78: Spectral Flux

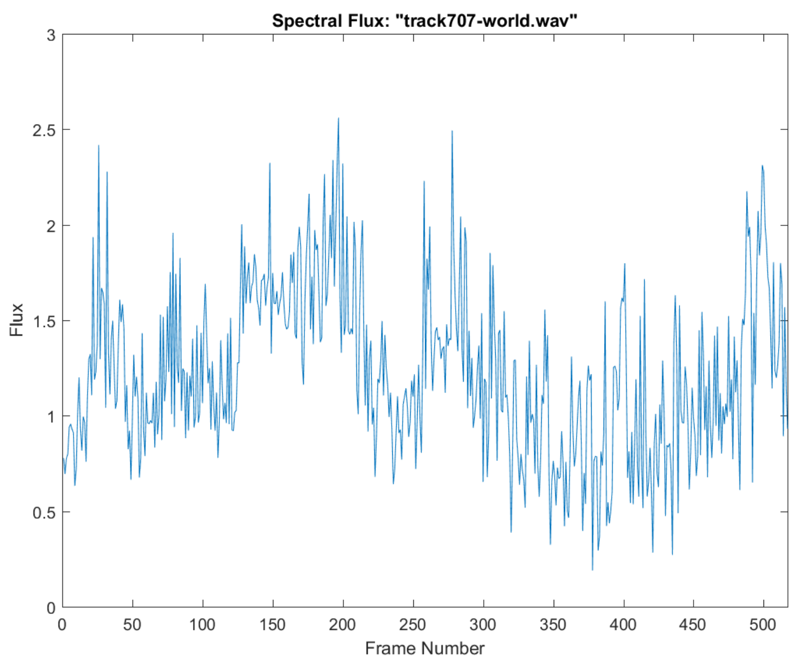


Figure 79: Spectral Flux

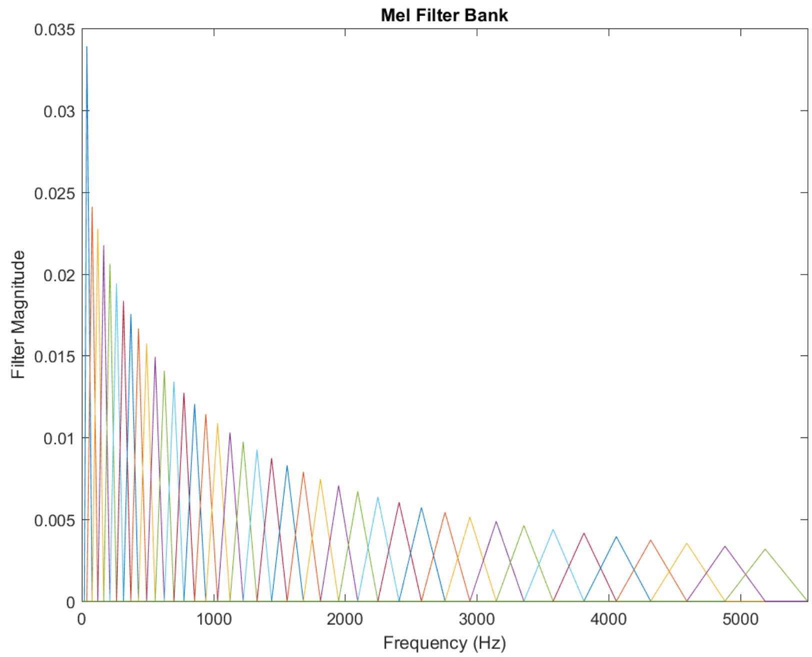


Figure 80: Mel Filter Bank

B Code

Listing 5: extractSound.m

```
function [ soundExtract,p ] = extractSound( filename, time )
%extractSound Extracts time (in seconds) from the middle of the song
%   Write a MATLAB function that extract T seconds of music from a
%   given track. You will use the MATLAB function audioread to
%   read a track and the function play to listen to the track.
info = audioinfo(filename);
[song,~]=audioread(filename);
if time >= info.Duration
    soundExtract=song;
    p=audioplayer(soundExtract,info.SampleRate);
    return;
elseif time<= 1/info.SampleRate
    error('Too small of a time to sample');
end
samples=time*info.SampleRate;
soundExtract=song(floor(info.TotalSamples/2)-floor(samples/2):1: ...
    floor(info.TotalSamples/2)+floor(samples/2));
p=audioplayer(soundExtract,info.SampleRate);
end
```

Listing 6: loudness.m

```
function [ loudness_data ] = loudness( filename,varargin )
%loudness Computes the standard deviation of the original audio divided
%into frames of size 255 loudness(filename,[franeSize,time,plotBool])
%   The standard deviation of the original audio signal x[n] computed over
%   a frame of size N provides a sense of the loudness
%
%   frameSize = size of frame to compute loudness over; defaults to 255
%   time = duration of song to analyze; defaults to 24 seconds
%   plotBool = whether or not to plot results; default is true
switch nargin
    case 1
        frameSize=512;
        time=24;
        plotBool=1;
    case 2
        frameSize=varargin{1};
        time=24;
        plotBool=1;
    case 3
        frameSize=varargin{1};
        time=varargin{2};
        plotBool=1;
    case 4
        frameSize=varargin{1};
        time=varargin{2};
        plotBool=varargin{3};
    otherwise
        error('loudness:argChk', 'Wrong number of input arguments');
end
[y,~]=extractSound( filename, time );
frames_data = buffer(y,frameSize,ceil(frameSize/2));
loudness_data=std(frames_data,0,1);
if plotBool==1
    p=plot(1:length(loudness_data),loudness_data(1,:));
    title(['Loudness per frame: "' filename '"']);
    xlim([0 length(loudness_data)+1]);
    xlabel('Frame Number');
    ylabel('Loudness');
    saveas(p,['Loudness_' filename(1:end-4) '.png']);
```

```
end
end
```

Listing 7: loudnessPlot.m

```
% plot loudness features
close all;
figure(1)
loudness('track201-classical.wav');
figure(2)
loudness('track204-classical.wav');
figure(3)
loudness('track370-electronic.wav');
figure(4)
loudness('track396-electronic.wav');
figure(5)
loudness('track437-jazz.wav');
figure(6)
loudness('track439-jazz.wav');
figure(7)
loudness('track463-metal.wav');
figure(8)
loudness('track492-metal.wav');
figure(9)
loudness('track547-rock.wav');
figure(10)
loudness('track550-rock.wav');
figure(11)
loudness('track707-world.wav');
figure(12)
loudness('track729-world.wav');
close all;
```

Listing 8: zeroCross.m

```
function [ ZCR_data ] = zeroCross( filename )
%zeroCross Computes the zero cross rate of the audio file
%   The Zero Crossing Rate is the average number of times the audio signal
%   crosses the zero amplitude line per time unit. The ZCR is related to...
%   the pitch height, and is also correlated to the noisiness of the...
%   signal.
frameSize=512;
time=24;
% info = audioinfo(filename);
[y,~]=extractSound( filename, time ); % Operate on middle 24 seconds
frames_data = buffer(y,frameSize,ceil(frameSize/2));
ZCR_data=zeros(1,length(frames_data));
ZCR_data(1:end)=sum(abs(diff(frames_data(1:end,:))))/length(frames_data);

p=plot(1:length(ZCR_data),ZCR_data(1,:));
title(['ZCR per frame: "' filename '"']);
xlim([0 length(ZCR_data)+1]);
xlabel('Frame Number');
ylabel('ZCR per frame');
saveas(p,['ZCR_' filename(1:end-4) '.png']);
end
```

Listing 9: zeroPlot.m

```
% plot zeroCross features
close all;
figure(1)
zeroCross('track201-classical.wav');
figure(2)
```

```
zeroCross('track204-classical.wav');
figure(3)
zeroCross('track370-electronic.wav');
figure(4)
zeroCross('track396-electronic.wav');
figure(5)
zeroCross('track437-jazz.wav');
figure(6)
zeroCross('track439-jazz.wav');
figure(7)
zeroCross('track463-metal.wav');
figure(8)
zeroCross('track492-metal.wav');
figure(9)
zeroCross('track547-rock.wav');
figure(10)
zeroCross('track550-rock.wav');
figure(11)
zeroCross('track707-world.wav');
figure(12)
zeroCross('track729-world.wav');
close all;
```

Listing 10: windows.m

```
function [Xn] = windows( filename, window )
%windows Implement the computation of the windowed Fourier transform of y
frameSize=512;
switch window
    case 1
        w=bartlett(frameSize); % Bartlett window
    case 2
        w=hann(frameSize); % Hann (Hanning) window
    case 3
        w=kaiser(frameSize,0.5); % Kaiser window
    otherwise
        w=kaiser(frameSize,0.5); % defaults to Kaiser window
end
[x,~]=extractSound(filename,24);
xn=buffer(x,frameSize);
Y=zeros(size(xn));
for i=1:length(xn)
    Y(:,i)=fft(xn(:,i).*w);
end
K=frameSize/2+1;
Xn=size(Y);
for i=1:length(xn)
    Xn(1:K,i)=Y(1:K,i);
end
end
```

Listing 11: windowsTiming.m

```
% Windows timing
fileArray=cellstr(['440Amp1.wav  '; '440Amp5.wav  ';'11025Amp1.wav';...
    '11025Amp5.wav'; '14080Amp1.wav'; '14080Amp5.wav']);
timings=zeros(1,length(fileArray)*3);
for i=1:length(fileArray)
    for j=1:3
        g=@()audioread(['' fileArray{i} '']);
        h=@()windows(['' fileArray{i} ''],j);
        timings((i-1)*3+j)=timeit(h)-timeit(g);
    end
end

```

```
bartTime=timings(1:3:end);bartTotal=sum(bartTime);
hannTime=timings(2:3:end);hannTotal=sum(hannTime);
kaiserTime=timings(3:3:end);kaiserTotal=sum(kaiserTime);
x=1:length(fileArray);
a=bar([bartTime;hannTime;kaiserTime],'stacked');
ax=gca;
ylim([min([bartTotal,hannTotal,kaiserTotal])*0.995,...
    max([bartTotal,hannTotal,kaiserTotal])*1.005]);
ylabel('Time (s)');
xlabel('Window method');
set(gca,'XTickLabel',['Bartman Window'; 'Hanning Window';...
    'Kaiser Window ' ]);
title('Window Performance');
saveas(gca,'windowsTiming.png');
close all;
```

Listing 12: freqDist.m

```
function [ Xk ] = freqDist( filename )
%freqDist Computes the frequency distribution
Xn=windows(filename, 1);
Xk=zeros(size(Xn));
for n=1:length(Xn)
    for k=1:min(size(Xk))
        Xk(k,n)=abs(Xn(k,n))./sum(abs(Xn(1:k,n)));
    end
end
end
```

Listing 13: centroidSpread.m

```
function [ centroid,sigmaK ] = centroidSpread( filename )
%centroid Summary of this function goes here
%   The spectral centroid can be used to quantify sound sharpness or
%   brightness. The spread quantifies the spread of the spectrum around
%   the centroid, and thus helps differentiate between tone-like and
%   noise-like sounds.
close all;
Xk=freqDist(filename);
[a,b]=size(Xk);
centroid=zeros(1,b);
sigmaK=zeros(1,b);
for n=1:b
    for k=1:a
        centroid(n)=centroid(n)+(k/a*Xk(k,n));
    end
end

for n=1:b
    for k=1:a
        sigmaK(n)=sigmaK(n)+(1/(a-1))*(Xk(k,n)-centroid(n))^2;
    end
end
sigmaK=sqrt(sigmaK);
x=1:b;

figure
plot(x,sigmaK);
title(['Frequency Spread: "' filename '"']);
xlabel('Frame Number');
ylabel('Spread');
xlim([0,b]);
saveas(gca,['freqSpread' filename(1:end-4) '.png']);

figure
```

```
plot(x,centroid);
title(['Frequency Centroid: "' filename '"']);
xlabel('Frame Number');
ylabel('Centroid');
xlim([0,b]);
saveas(gca,['freqCent' filename(1:end-4) '.png']);
close all;
end
```

Listing 14: centroidPlot.m

```
close all;
centroidSpread('track201-classical.wav');
centroidSpread('track204-classical.wav');
centroidSpread('track370-electronic.wav');
centroidSpread('track396-electronic.wav');
centroidSpread('track437-jazz.wav');
centroidSpread('track439-jazz.wav');
centroidSpread('track463-metal.wav');
centroidSpread('track492-metal.wav');
centroidSpread('track547-rock.wav');
centroidSpread('track550-rock.wav');
centroidSpread('track707-world.wav');
centroidSpread('track729-world.wav');
close all;
```

Listing 15: specFlat.m

```
function [ SFn ] = specFlat( filename )
%specFlat Computes the spectral flatness of a signal
%   Spectral flatness is the ratio between the geometric and arithmetic
%   means of the magnitude of the Fourier transform
Xk=freqDist(filename);
SFn=(geomean(Xk)./mean(Xk));

close all;
figure
plot(SFn);
title(['Spectral Flatness: "' filename '"']);
xlabel('Frame Number');
ylabel('Flatness');
xlim([0,length(SFn)]);
saveas(gca,['specFlat_' filename(1:end-4) '.png']);
close all;
end
```

Listing 16: specFlux.m

```
function [ Fn] = specFlux( filename)
%specFlux Measures how quickly power spectrum is changing
%   The spectral flux is a global measure of the spectral changes between
%   two adjacent frames, n-1 and n
Xk=freqDist(filename);
Fn=sum(diff(Xk).^2);

close all;
figure
plot(Fn);
title(['Spectral Flux: "' filename '"']);
xlabel('Frame Number');
ylabel('Flux');
xlim([0,length(Fn)]);
saveas(gca,['specFlux' filename(1:end-4) '.png']);
close all;
```

```
end
```

Listing 17: specPlot.m

```
close all;
specFlux('track201-classical.wav');
specFlux('track204-classical.wav');
specFlux('track370-electronic.wav');
specFlux('track396-electronic.wav');
specFlux('track437-jazz.wav');
specFlux('track439-jazz.wav');
specFlux('track463-metal.wav');
specFlux('track492-metal.wav');
specFlux('track547-rock.wav');
specFlux('track550-rock.wav');
specFlux('track707-world.wav');
specFlux('track729-world.wav');

specFlat('track201-classical.wav');
specFlat('track204-classical.wav');
specFlat('track370-electronic.wav');
specFlat('track396-electronic.wav');
specFlat('track437-jazz.wav');
specFlat('track439-jazz.wav');
specFlat('track463-metal.wav');
specFlat('track492-metal.wav');
specFlat('track547-rock.wav');
specFlat('track550-rock.wav');
specFlat('track707-world.wav');
specFlat('track729-world.wav');
close all;
```

Listing 18: melBank.m

```
function [ fbank ] = melBank(  )
%melBank Creates a set of mel filter banks
%   Implement the computation of the triangular filterbanks
%   Hp, p = 1,...,NB. Your function will return an array fbank of size
%   NB x K such that fbank(p,:) contains the filter bank Hp.
close all;
nbanks = 40; %% Number of Mel frequency bands
% linear frequencies
N=512;
K=N/2+1;
fbank=zeros(nbanks,K);
fs=11025;
linFrq = 20:fs/2;
% mel frequencies
melFrq = log ( 1 + linFrq/700) *1127.01048;
% equispaced mel indices
melIdx = linspace(1,max(melFrq),nbanks+2);
% From mel index to linear frequency
melIdx2Frq = zeros (1,nbanks+2);
% melIdx2Frq (p) = \Omega_p
for i=1:nbanks+2
    [~, indx] = min(abs(melFrq - melIdx(i)));
    melIdx2Frq(i) = linFrq(indx);
end
melTemp=zeros(nbanks,max(melIdx2Frq));
% Implement equation for mel filters
for i = 2:nbanks+1
    currMel = melIdx2Frq(i);
    nextMel = melIdx2Frq(i+1);
    lastMel = melIdx2Frq(i-1);
    A=(2/(nextMel-lastMel));
```

```
    melDex=lastMel:currMel;
    melTemp(i-1,melDex)=A.*(melDex-lastMel)./(currMel-lastMel);
    melDex=currMel:nextMel;
    melTemp(i-1,melDex)=A.*(nextMel-melDex)./(nextMel-currMel);
end

W = round(linspace(1,max(melIdx2Frq),K));
for i = 1:nbanks
    fbank(i,:) = melTemp(i,W);
end
figure
plot(melTemp.');
title('Mel Filter Bank');
xlabel('Frequency (Hz)');
ylabel('Filter Magnitude');
xlim([0,length(melTemp)]);
saveas(gca,'melFilterBank.png');
close all;
end
```

Listing 19: mfcc.m

```
function [ mfccp ] = mfcc( fbank,Xn )
%mfcc Compute the Mel frequency coeffecients
%   The mel-spectrum (MFCC) coefficient of the n-th frame is defined for
%   p = 1,...,NB
[a,b]=size(fbank);
fbank=fbank./max(fbank,2);
mfccp=zeros(a,b);
for p=1:a
    for k=1:b
        mfccp(p)=mfccp(p)+(abs(fbank(p,k)*Xn(p,k)).^2);
    end
end
```