DSP Melody Transcription

1. Sai Manish Sasanapuri (IMT2018520)
2. Shubhayu Das (IMT2018523)
3. Tanmay Joshi (IMT2018527)
4. Veerendra S. Devaraddi (IMT2018529)

November 26, 2020

Contents

1 Description of Task

Roughly defined, a melody is a sequence of musical notes. To a listener, it is the most identifiable element of a musical composition. The task of melody transcription is to obtain the sequence of notes, given a music signal.

We attempt to do this in the following steps:

1. For the given audio clip, detect the boundary between two consecutive notes.
2. Segment the audio clip at these boundaries.
3. Detect the main frequency (frequencies) present in each segment and infer the note that was played by the musical instrument.

Once this is done for a piano recording, we attempt two more tasks.

1. Add in noise to the input audio clip. Utilize filters and other techniques to deal with this noise.
2. Try and generalize to other musical instruments.

While we get correct results for the basic tasks, our transcriptions are not accurate for another instrument that we tried (guitar).

2 Dealing with Noise

Noise is generally any unpleasant sound and, technically any unwanted sound that is unintentionally added to a desired sound. In this section, we try to address some techniques to remove noise.
One of the popular noise distributions to consider is White Gaussian noise(WGN), the properties of WGN would help us to remove it. The properties of WGN are:

1. It spreads uniformly over the frequency spectrum.
2. It is a normal distribution, with 0 mean and variance as square root of the power.

Plot of a white Gaussian noise.

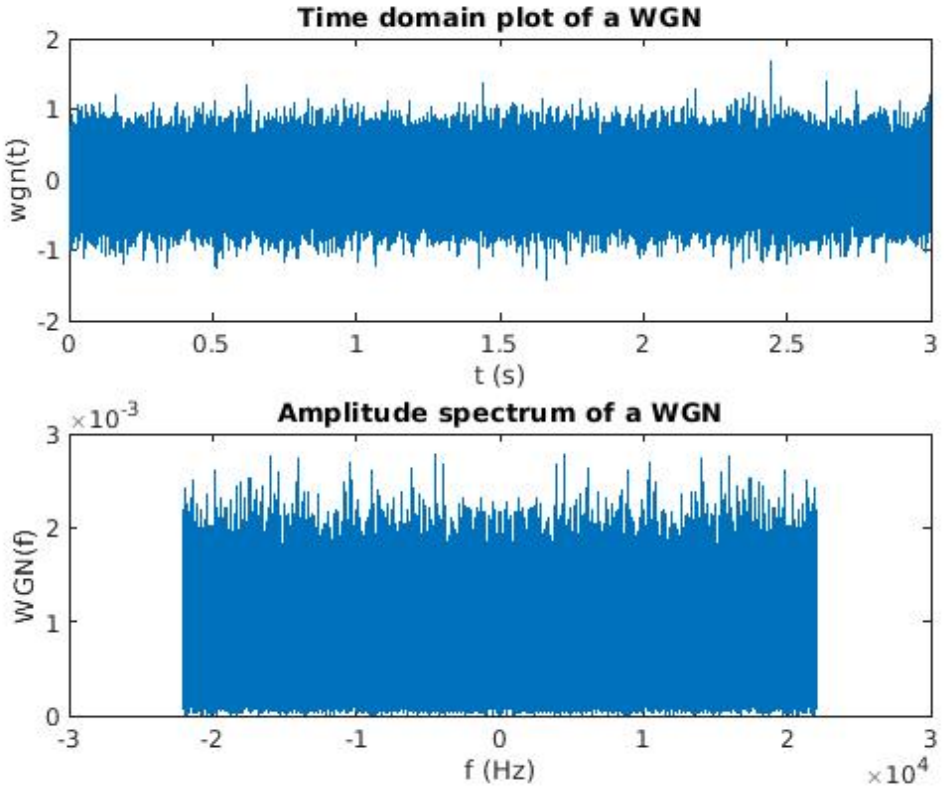

Figure 1: Plot of a WGN, to emphasize its properties.

Methods to filter WGN noise

2.1 Applying a moving average.

The audio signal containing a WGN noise can be averaged over a moving window to reduce the contribution of noise to the total power of the audio. This is effective in doing so because of its normal distribution and mean equal to zero. Larger the averaging window better the performance of this method, in removing WGN. This technique fails in the case where the average over a window of the desired signal is itself close to zero and a large(depends on the signal) window length is chosen.

Plot of a scenario where this method fails.

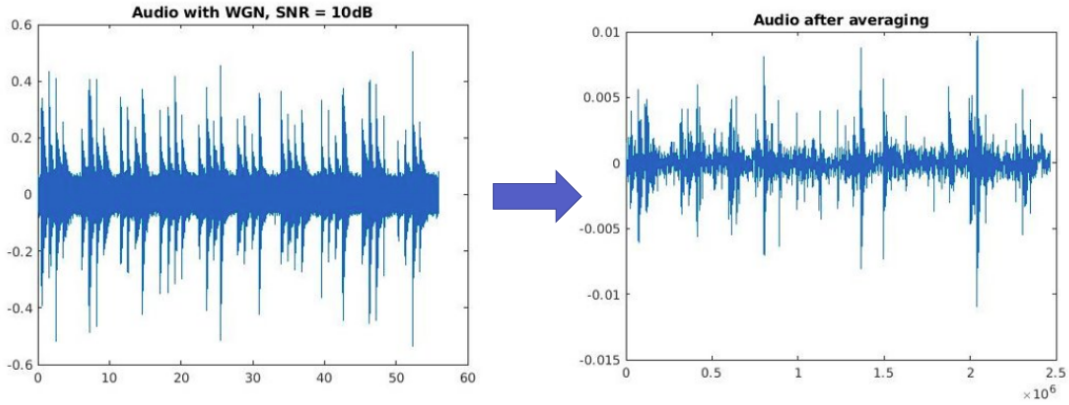

Figure 2: Plot of a scenario where this method fails.

2.2 Filtering using a FIR filter.

White Gaussian Noise is of "white" nature, meaning it spreads uniformly over the available frequency spectrum. As a consequence, WGN can not be completely removed by filtering, from the audio signal. The situation can be improved by filtering out over the relevant range of frequency (frequency range where the desired audio exists) or removing the frequency range where uniform amplitude samples exist. The result is that the signal-to-noise ratio of the audio signal increases.

2.2.1 Designing a Kaiser window

Kaiser window can be used to filter out the relevant frequency range. Kaiser window can be implemented with the help of MATLAB's FilterDesigner app.

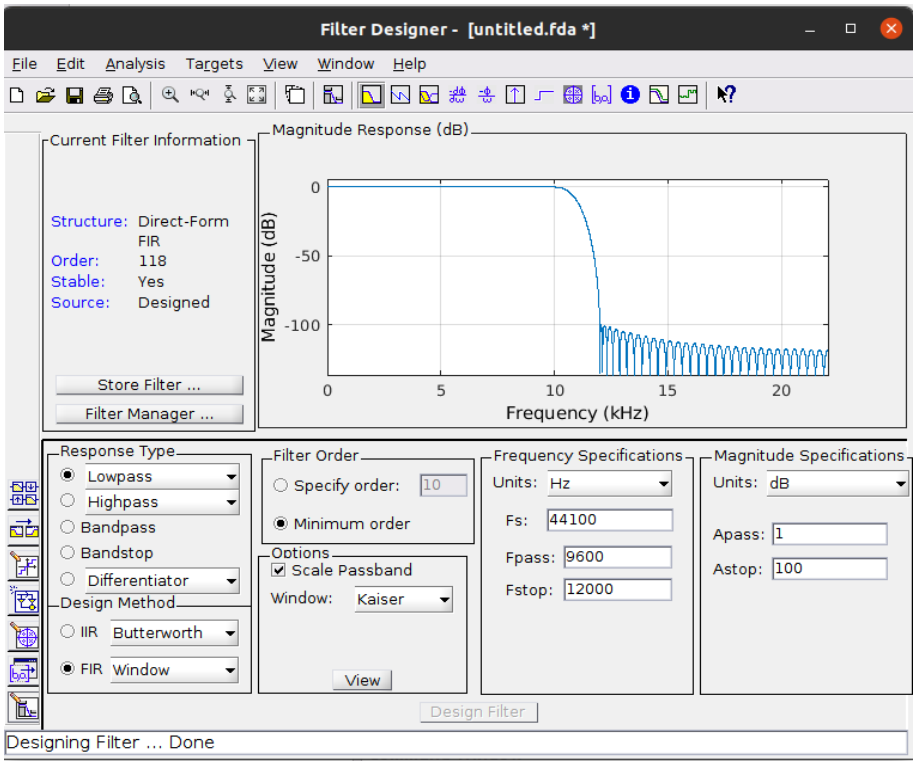

Figure 3: Filter designer dialog box

2.2.2 Determining a energy containment bandwidth

The process can be generalised by judging the relevant frequency band, based on the energy containment specification. For example, a 95% energy containment bandwidth will help us remove most of the energy, that is contributed by WGN.

Code snippet

```
function [hfreq] = EnergyContainmentBandwidth(y, percentage, Fs)
    % Get the PSD of the signal
    [~, psd] = GetPSD(y, Fs);

    % Find the total power from the PSD
    total_power = sum(psd);

    % Account for even symmetry and convert to a percentage
    target_power = (total_power/200) * percentage;
```

```
10
11      % Filter to get 95% containment bandwidth
12      dummy = 0;
13      index=1;
14      while index < length(psd)
15          dummy = dummy + psd(index);
16          if(dummy > target_power)
17              break
18          end
19          index = index + 1;
20      end
21
22      % Return threshold frequency
23      hfreq = index * Fs / length(y);
24  end
```

2.2.3 Analysis of this method on a sample clip

The changes this method has made on the audio is :

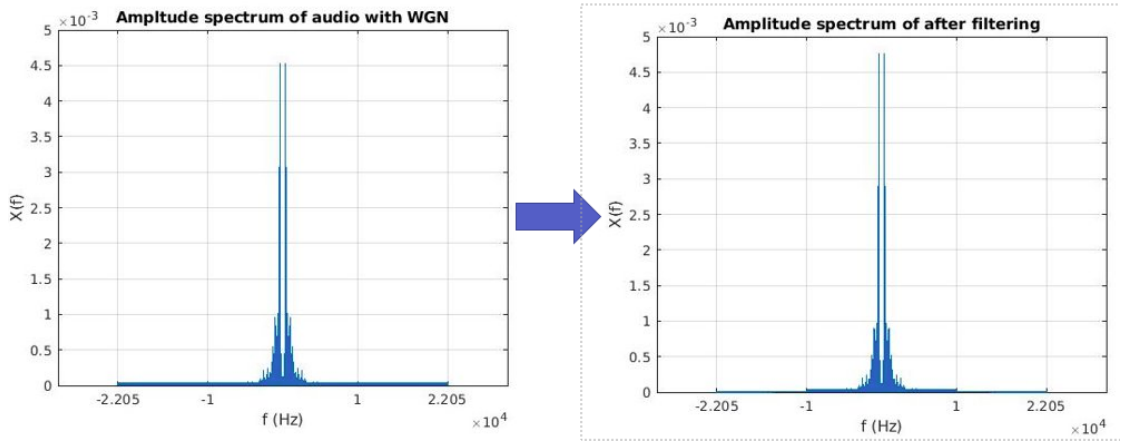

Figure 4: Figure to show the changes due to filtering.

1. Total average power of the audio sample, before adding WGN = 0.0036 W.
2. WGN is added to the audio sample to achieve an SNR of 10dB, now the total power = 0.0040 W = Total power of the audio sample + Power due to the WGN that got added.
3. Therefore, Power due to the WGN that got added = 0.0040 - 0.0036 = 0.0004 W.

4. A 95% energy containment bandwidth is calculated and a low pass filter is applied at that frequency. Now the total power of filtered out audio = 0.0038 W.

5. Therefore, in this case 50% of the WGN is removed.

Effectiveness of this method can be more witnessed when the SNR is lesser (<15dB)(or when fractional power of WGN higher). This method would remove a proportion of WGN noise, so the next method is proposed to remove WGN completely.

2.3 Audio Denoising using wavelet decomposition and thresholding

Comparison between Fourier transform and wavelet transform :

Fourier Transform is powerful tool for signal analysis but it doesn't detect abrupt changes efficiently because

1. The Fourier transform provides frequency information of a signal that represents frequencies and their magnitude.

2. It doesn't tell us when in time the frequencies exist. The transform is therefore ideal for stationary or time-invariant signals.

Short-Time-Fourier Transform(STFT) :

1. The STFT was developed to overcome the lack of temporal information in the Fourier transform. It gives us a time-frequency representation of the signal.

2. With STFT we can assume some portion of the non-stationary signal is stationary. We then take a fourier transform of each stationary segment of the signal and add them up.

Limitation with STFT :

1. The window function is finite,so frequency resolution decreases.

2. Fixed length of the window means time and frequency resolutions are fixed for the entire length of the signal.

3. We cannot know which frequencies exist at which time instance, but we can find which frequency bands exist at given time intervals.

Narrow window in time → good time resolution, poor frequency resolution
Wide window in time → poor time resolution, good frequency resolution

Therefore, to accurately analyze signals and images that have abrupt changes, we need to use a new class of functions that are well localized in time and frequency. This brings us to the topic of Wavelets.

What is a Wavelet?

A wavelet is a rapidly decaying, wave-like oscillation that has zero mean. Unlike sinusoids, which extend to infinity, a wavelet exists for a finite duration.

The wavelet Ψ is our new basis function, and acts as a window function.

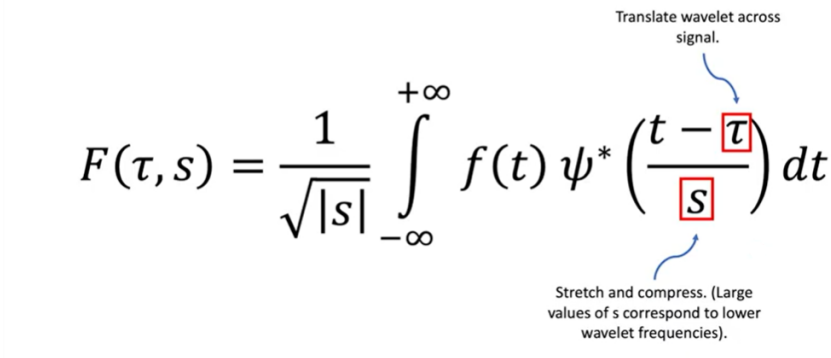

Figure 5: Wavelet Coefficient

When we translate the wavelet across the signal using τ, it is scaled by dividing by **s**. so this stretches and compresses the wavelets. Large values of **s** corresponds to lower wavelet frequencies and small values of **s** corresponds to higher wavelet frequencies. A stretched wavelet helps in capturing slowly varying changes in a signal, and while the compressed wavelet helps in capturing abrupt changes.

So, when the frequency of a wavelet matches part of the signal with very similar frequency we get high outputs $F(\tau, s)$ which are called wavelet coefficients(approximation coefficients). Wavelet coefficients(approximation coefficients) represent the low frequency of our signal and detailed coefficients represent high frequency components of our signal.

Discrete Wavelet Transform:

1. Calculating wavelet coefficients at every possible scale produces a lot of data.
2. If s and τ are based on powers of two(dyadic) then analysis becomes much more efficient and accurate.

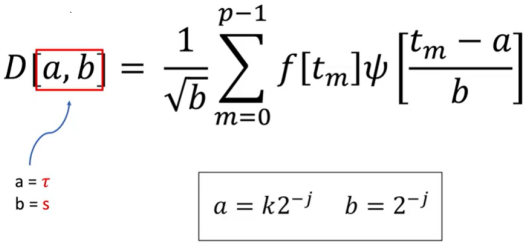

Figure 6: Wavelet Coefficient in Discrete Wavelet Transform

a and b are now dyadic, j is the scale index and k is our wavelet transformed signal index.

Multilevel Decomposition : Using Multilevel wavelet decomposition to get approximate coefficients and detail coefficients at each level of decomposition.

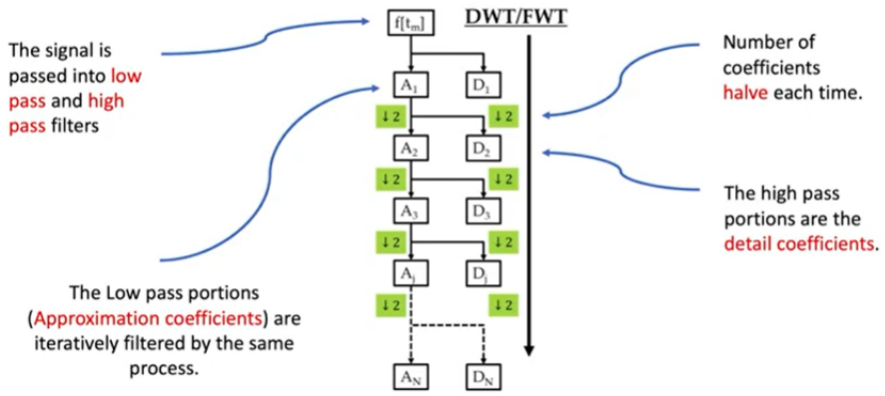

Figure 7: Multilevel Decomposition

Working of the function :

```
%% Function to denoise the input signal
function [de] = Denoise(noisy)
    [c,l] = wavedec(noisy,3,'db4');

    N = length(noisy);
    sigma = median(abs(c))/(0.6745*3);
    lambda = sigma * sqrt(2*9*log(N));

    b = wthresh(c,'s',lambda);
    de = waverec(b,l,'db4');
end
```

Firstly, we apply multilevel wavelet decomposition to decompose the noisy signal using wavelet decomposition into coefficients and levels. Here we are using 3 level decomposition and daubechies 4 wavelet (db4) wavelet for decomposing.

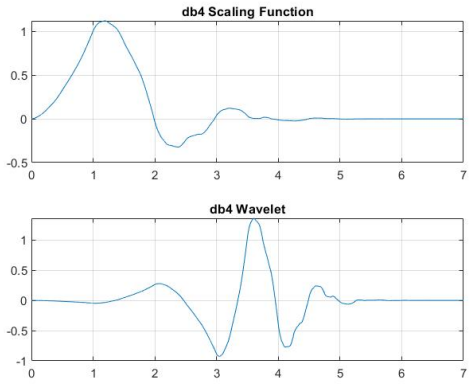

Figure 8: db4 Wavelet

Generally the noise resides at higher frequency compared to actual message signals which reside at lesser frequency. In Wavelet decomposition the noise in the signal is generally contained in the smaller coefficients, the more dominant and useful parts have larger coefficients.

Therefore removing or nullifying the smaller coefficient values using threshold makes denoising of the signal possible.

Calculating the threshold value λ by using universal threshold(square root log) method with little modification is given by,

$$\lambda = \sigma * \sqrt{(2 * 9 * log(N))}$$

Where, N is length of the noisy signal and σ is median absolute deviation(MAD) is given by,

$$\sigma = MAD_j/0.6745 = median(|c|)/(0.6745 * 3)$$

where, c represents wavelet coefficients.

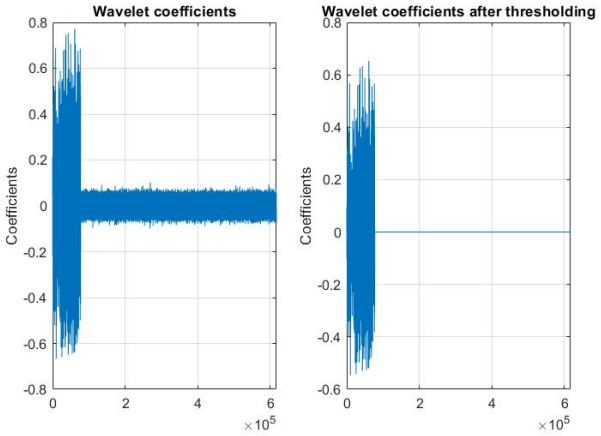

Figure 9: Wavelet Coefficients before and after threshold

In the above plot, left side subplot is the graph of Wavelet coefficients of noisy signal and the right side subplot is the graph of wavelet coefficients after applying threshold.

Now to get a denoised signal, we reconstruct the signal using coefficients after applying a soft threshold to the wavelet coefficients

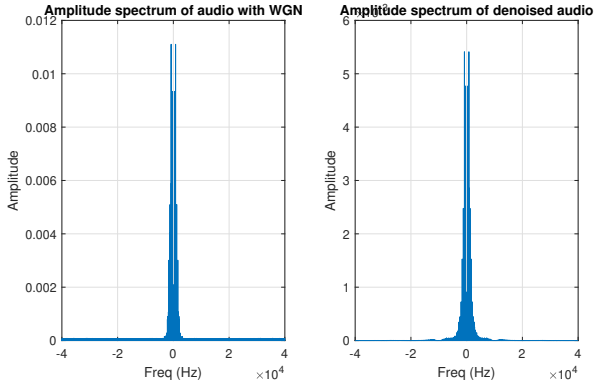

Figure 10: Noisy signal and Denoised signal

The above plot shows the difference in the Amplitude spectrum before and after denoising.

3 Various attempts at segmentation

3.1 Attempt 1

This method of segmentation is based on thresholding the amplitude. We define two threshold values, *threshup* and *threshdown* in terms of the maximum amplitude in the whole audio.

$threshup \leftarrow a \times max(data)$
$threshup \leftarrow b \times max(data)$

where a and b are picked through trial and error, and $a > b$.

Using these threshold values, we get the boundaries of segments on iterating through all the samples.

quiet ← 1
while iterating through the samples **do**
  if *quiet* **then**
    if *data* ≥ *threshup* **then**
      quiet ← 0;
      record the sample index in *boundaries* array;
    end if
  else
    if *data* ≤ *threshdown* **then**
      quiet ← 1;
      record the sample index in *boundaries* array;
    end if
  end if
end while

All indices at which the segments should start or end have now been stored in the *boundaries* array. These values can be used to get separate segments of the audio.

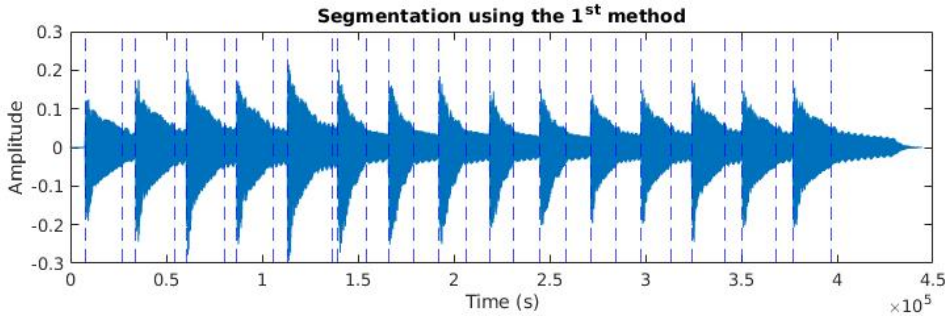

Figure 11: Segments obtained using method 1.

The method works when different notes are played for different durations. So, we can say that it generalises in time.

However, it doesn't work when different notes are played with different intensities. This is because the thresholds are defined in the terms of highest amplitude in the whole audio. For example, if a piano player strikes the keys with varying strengths, the harder struck keys will produce notes of higher amplitude, and the softly struck keys produce notes of lower amplitude. The method doesn't work in such cases. Hence, the method doesn't generalise in amplitude.

This hints towards the need for a moving average based approach, which was explored as the second attempt.

3.2 Attempt 2

To improve performance, we tried an alternative approach to segment the audio. The key steps in this method are:

3.2.1 Non-linear transform

The noise-free audio signal undergoes the following transform: $e^{|y|+1} - e$. This results in amplifying all amplitudes of the signal. Peaks in the original signal become even more prominent. The differences between the high-amplitude peaks and low-amplitude noise becomes more prominent.

A plot showing the difference.

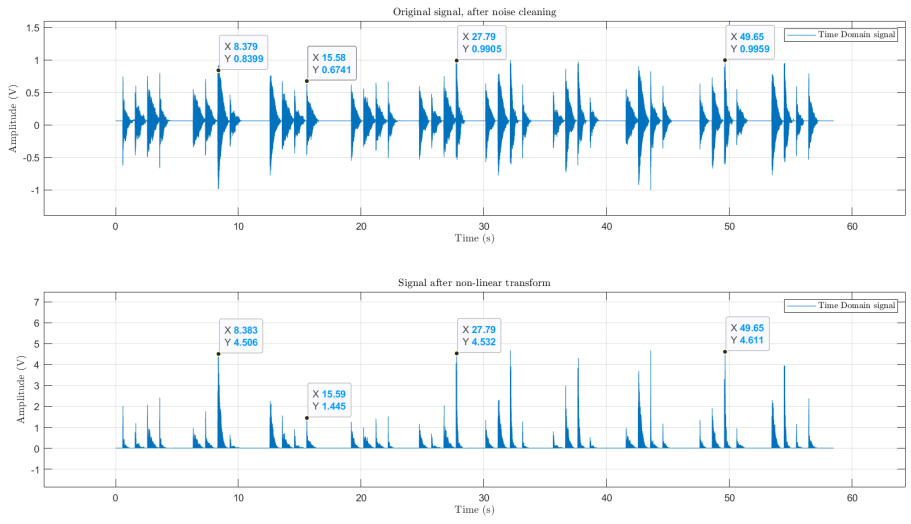

Figure 12: Comparison between the noise-filtered signal and the transformed signal. The labelled peaks show the difference.

Code snippet

```
y_norm = normalize(y_filtered, 'range', [-1, 1]);
y_transformed = exp(abs(y_norm.^2) + 1) - 2.718;
```

3.2.2 Compute moving average of the transformed signal

The next step is to compute a windowed moving average of the transformed signal. This helps in smoothing out sudden changes in the signal. Low amplitude components decrease this average, telling us where the end/start points of the next segment are likely to be.

Problems with different tones being played at different amplitudes are taken care of, by choosing an appropriate window length. I am further scaling this moving mean by a constant factor of two for convenience only. Experimentally, a window length of $\frac{Fs}{10}$ samples was found to generalize well over a variety of sound clips. With this window size, we can detect notes which are played for longer than 100ms approx. Faster notes *might* not be detected properly.

The smoothing effect of the moving average

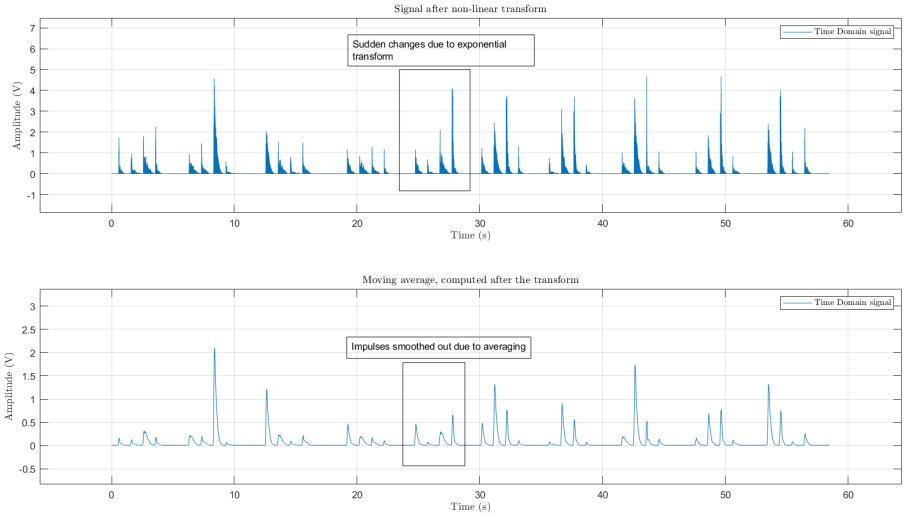

Figure 13: Sudden changes in the signal are removed, with a smoothing effect due to the moving average. The window size prevents over-smoothing, which can lead to loss of some segments/notes.

Code snippet

```
y_mov_avg = 2 * movmean(y_transformed, floor(Fs/10));
```

3.2.3 Detecting minima in the moving average

The next steps involves finding all the local minima in the moving average, which will give us our segment boundaries. For this, we utilize the *islocalmin* function in MATLAB.

Local minima are detected, by calculating their prominence. The prominence gives a measure of how a "valley" stands out with respect to its surrounding peaks. For further details, refer here.

We experimentally found that the mean of the original signal, after noise filtering, is a good threshold of prominence.

Detection of local minima

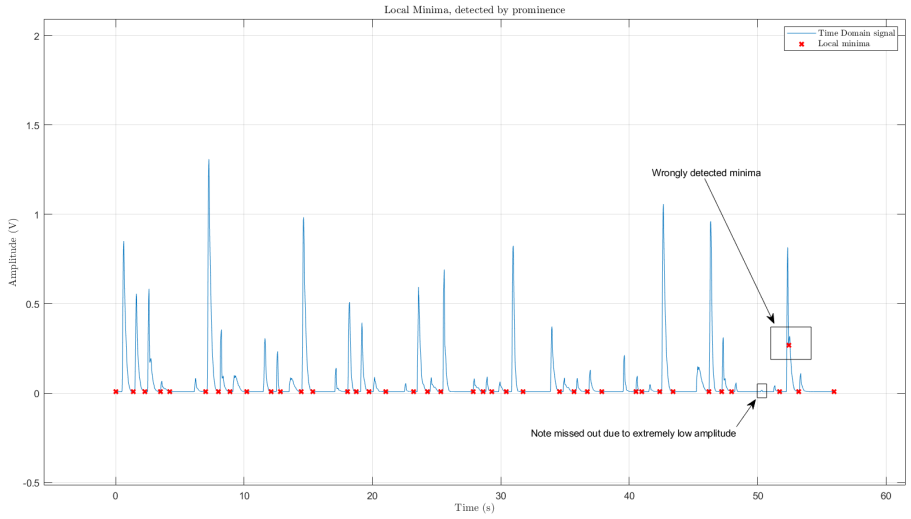

Figure 14: Detecting the local minima, which indicate the beginning/end of an audio segment.

Code snippet

```
1  minima = islocalmin(y_mov_avg, 'MinProminence', mean(abs(y)));
2  minima(1) = 1; % Forcing first sample to be detected as a boundary
3  minima(end) = 1; % Forcing last sample to be detected as a boundary
4  localMinima = find(minima);
```

3.2.4 Detecting and removing inaccurate local minima points

By taking such a small window for the previous moving average, we risked not smoothing out some peaks enough missing out on some sudden peaks. As a result of this, some sharp changes in the moving average are considered to be local minima, with high prominence.

To deal with this, we perform outlier detection on the detected minima points. The outlier detection works by computing a local threshold over a window. Any local minima points greater than this threshold are detected as outliers.

The local threshold is calculated by computing the moving mean over a window of length: $\frac{Fs}{20}$ samples. The size of this window was determined experimentally. However, this length has to be shorter than the window using for calculating the moving average in the previous step, to be effective.

Detection of actual local minima

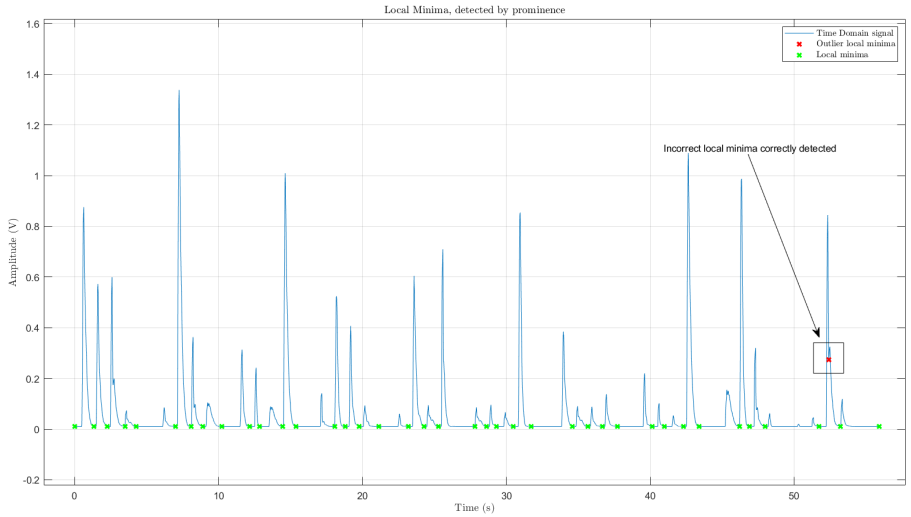

Figure 15: Detecting the outlier local minima. We observe that the spurious local minima was correctly detected.

Code snippet

```
outlierEdges = isoutlier(y_mov_avg(localMinima), 'movmean', floor(Fs
    /20));
outliers = localMinima(outlierEdges == 1);
actualLocalMinima = localMinima(outlierEdges == 0);
```

3.2.5 Final segmentation

Now that all the actual local minima(segment thresholds) accurately, all that remains is to segment the original audio signal(noise free). This is very easily done, with the detected segment thresholds.

The problem is that one note wasn't detected, because of its short duration. There is nothing more that can be done about this, in this approach.

Code snippet

```
for i = 1:length(actualLocalMinima) - 1
    temp = y(actualLocalMinima(i):actualLocalMinima(i+1)-1);  % Segmenting
    temp = vertcat(zeros(maxLength - length(temp), 1), temp); % Zero padding for ease
    segments(:, i) = temp;  % Appending segment to array of segments
end
```

Segmenting the audio, based on the detected thresholds

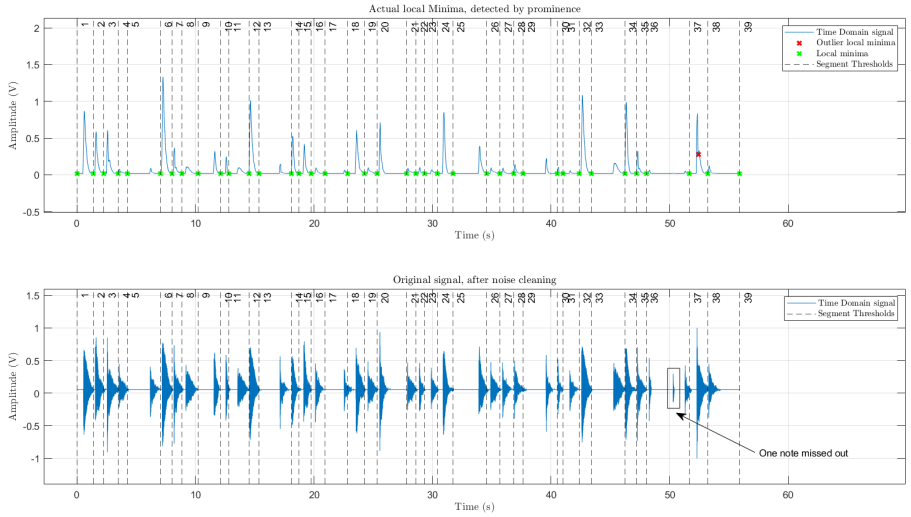

Figure 16: With the exception of one note, all other notes were segmented properly.

4 Frequency and Note Detection

Now that we have individual segments, containing one note/chord, we can detect the frequency(frequencies) present in that segment. Our method of choice is to compute the power spectrum in each segment, then pick up the highest peak. Using this approach on the given audio clip from YouTube:

Segmenting the audio, and detecting the frequency of the first 40 notes in the audio clip

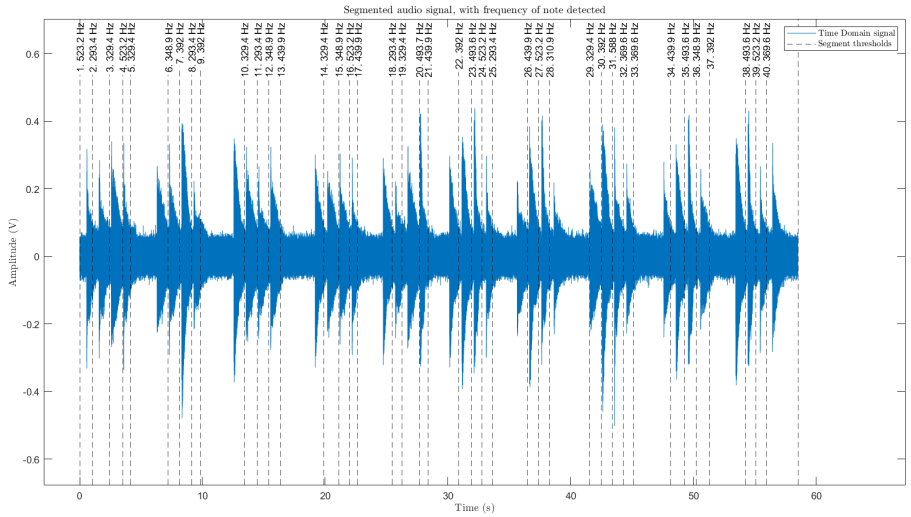

Figure 17: All 40 notes were successfully detected. For many of the segments, the harmonic frequencies were detected instead of the fundamental frequency,

Segmenting the audio, and detecting the frequency of the next 40 notes in the audio clip

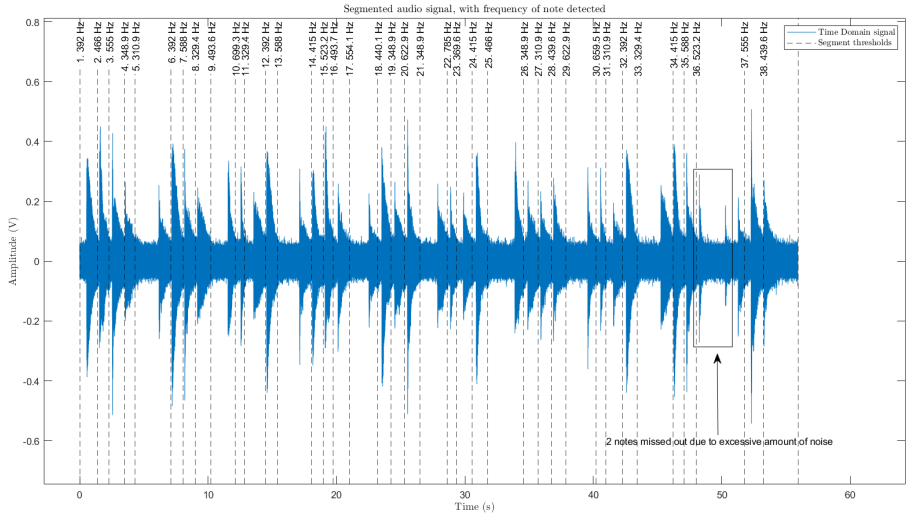

Figure 18: With the exception of two notes, all other notes were segmented properly. Again, harmonics were detected a few times

Code snippet

```
for j=1:length(localMinima) - 1
    [freqs, PSD] = GetPSD(segments(:, j), Fs);
    [~, index] = max(PSD);
    notes(j) = round(abs(freqs(index)), 1);
end
```

Overall, 78 notes out of 80 notes were detected and segmented properly. Our results, compared with the original notes from YouTube, are listed in Appendix A.

5 Testing with other sound clips

5.1 Audio played at different speeds

When an audio is played at n times the original speed, we expect all frequencies to get n times the original. So, for an audio played at $2\times$, we expect all notes to be an octave higher. For $4\times$, all notes should be two octaves higher.

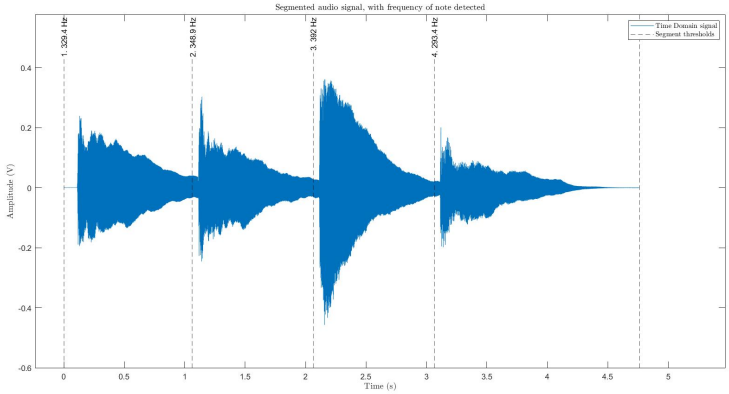

Figure 19: E4, F4, G4, D4 recorded from a piano. Played at the original speed.

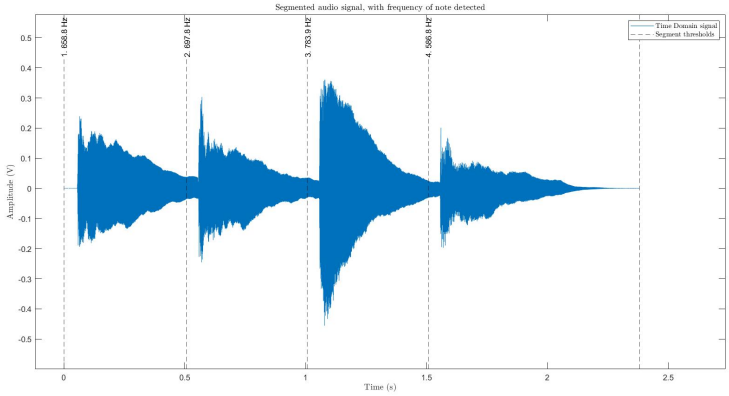

Figure 20: E4, F4, G4, D4 recorded from a piano. Played at $2\times$ the original speed. The observed sequence is E5, F5, G5, D5

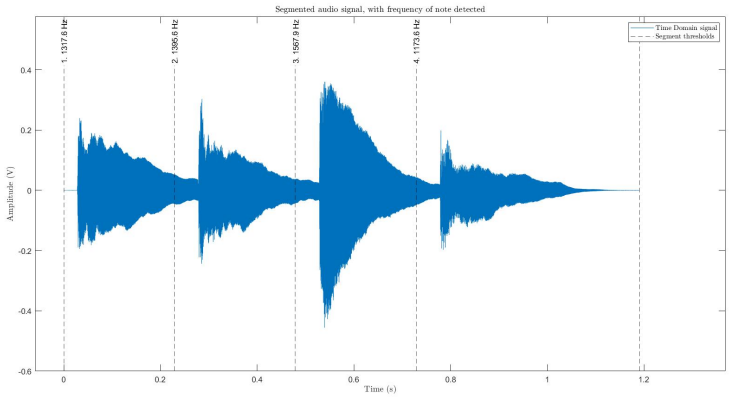

Figure 21: E4, F4, G4, D4 recorded from a piano. Played at the 4× the original speed. The observed sequence is E6, F6, G6, D6.

5.2 Different Musical instrument

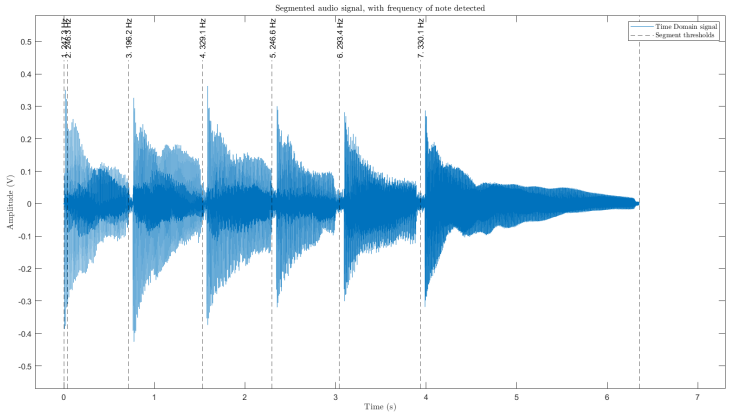

Figure 22: The notes E3, G3, A4, B4, D4 and E4 were played on a guitar

We tried our code on an audio clip from a guitar. The note with label 4 was detected with wrong frequency and rest of the notes are detected correctly. The segmentation makes a mistake towards the beginning of the clip and detects an extra note. This is due to the large amplitude components at the very beginning.

6 Future improvements

1. Energy containment bandwidth can be replaced with a method that can detect the frequency ranges that contains as much as WGN only in all the cases.
2. A better method to calculate threshold for nullifying the coefficients that corresponds to noise.
3. A similar method of segmentation can be done using local maxima and combined with the current segmentation method(using minima) to improve the precision of time bounds of each note segment.
4. A better method for detecting the frequency of each note. Eg: Using n-way crossover filters to detect all present frequencies and then removing harmonics.

7 Appendix

7.1 Part A

We report 2 undetected segments and 9 segments, in which the next harmonic was detected. All other segments were accurately detected

Segment #	Note	Actual note frequency	Detected Note frequency	Conclusion
1	C4	262	523.2	Next harmonic(C5) detected
2	D4	293.665	293.4	Accurately detected
3	E4	329.628	329.4	Accurately detected
4	C4	262	523.2	Next harmonic(C5) detected
5	E4	329.628	329.4	Accurately detected
6	F4	349.228	348.9	Accurately detected
7	G4	391.995	392	Accurately detected
8	D4	293.665	293.4	Accurately detected
9	G4	391.995	392	Accurately detected
10	E4	329.628	329.4	Accurately detected
11	D4	293.665	293.4	Accurately detected
12	F4	349.228	348.9	Accurately detected
13	A4	440	439.9	Accurately detected
14	E4	329.628	329.4	Accurately detected
15	F4	349.228	348.9	Accurately detected
16	C4	262	523.2	Next harmonic(C5) detected
17	A4	440	439.9	Accurately detected
18	D4	293.665	293.4	Accurately detected
19	E4	329.628	329.4	Accurately detected
20	B4	493.88	493.7	Accurately detected
21	A4	440	439.9	Accurately detected
22	G4	391.995	392	Accurately detected
23	B4	493.88	493.6	Accurately detected
24	C4	262	523.2	Next harmonic(C5) detected
25	D4	293.665	293.4	Accurately detected
26	A4	440	439.9	Accurately detected
27	C5	523.25	523.2	Accurately detected
28	Eb	311.27	310.9	Accurately detected
29	E4	329.628	329.4	Accurately detected
30	G4	391.995	392	Accurately detected
31	D5	587.33	588	Accurately detected
32	F#	369.994	369.6	Accurately detected
33	F#	369.994	369.6	Accurately detected
34	A4	440	439.9	Accurately detected
35	B4	493.88	493.6	Accurately detected
36	F4	349.228	348.9	Accurately detected
37	G4	391.995	392	Accurately detected
38	B4	493.88	493.6	Accurately detected
39	C4	262	523.2	Accurately detected
40	F#	369.994	369.6	Accurately detected

41	G4	391.995	392	Accurately detected
42	Bb	466.16	466	Accurately detected
43	Db	277.183	555	Next harmonic detected
44	F4	349.228	348.9	Accurately detected
45	Eb	311.27	310.9	Accurately detected
46	G4	391.995	392	Accurately detected
47	D5	587.33	588	Accurately detected
48	E4	329.628	329.4	Accurately detected
49	B4	493.88	493.6	Accurately detected
50	F5	698.46	699.3	Accurately detected
51	E4	329.63	329.4	Accurately detected
52	G4	392	392	Accurately detected
53	D5	587.33	588	Accurately detected
54	Ab	415.30	415	Accurately detected
55	C5	523.25	523.2	Accurately detected
56	B3	246.94	493.9	Next harmonic(B4) detected
57	Db	277.183	554	Next harmonic detected
58	A4	440	440.1	Accurately detected
59	F4	349.228	348.9	Accurately detected
60	Eb	311.27	623.1	Next harmonic detected
61	F4	349.228	348.9	Accurately detected
62	G5	783.99	785	Accurately detected
63	F#	369.994	369.6	Accurately detected
64	Ab	415.30	415	Accurately detected
65	Bb	466.16	466	Accurately detected
66	F4	349.228	348.9	Accurately detected
67	Eb	311.27	310.9	Accurately detected
68	A3	220	439.6	Next harmonic(A4) detected
69	Eb5	622.9	622.9	Accurately detected
70	E5	659.25	659.5	Accurately detected
71	Eb4	311.27	310.9	Accurately detected
72	G4	392	392	Accurately detected
73	E4	329.628	329.4	Accurately detected
74	Ab4	415.30	415	Accurately detected
75	D5	587.33	588	Accurately detected
76	C5	523.25	523.2	Accurately detected
77	Ab5	830.61	Not detected	too low amplitude
78	C4	261.63	Not detected	too low amplitude
79	Db5	554.37	555	Accurately detected
80	A4	440	439.6	Accurately detected

Table 1: All the segments detected by our program. The detected frequencies are compared to the original frequency.

8 Code Appendix

```
function [] = EntireCode()
    %% Input section, we receive a signal, which has noise added to
        it
    % Read in the waveform
    [y, Fs] = audioread('Test_Tune1.wav');

    % Selecting one channel from the input audio
    y = y(:, 1);

    % Synthetically add in WGN
    y = awgn(y, 10, 'measured');

    %% Filtering out the noise
    % Finding the 95% energy containment bandwidth
    bw = EnergyContainmentBandwidth(y, 95, Fs);

    % Using a FIR filter to extract this fraction of the audio
    Hd = LowPassFilter(bw, bw + 50, Fs);
    y_filt = filter(Hd,y);

    % Applying denoising to further improve the audio quality
    y_mod = Denoise(y_filt);

    %% Transforming the signal for performance improvement
    % Scale the audio for preprocessing calculations
    y_mod = normalize(y_mod, 'range', [-1, 1]);
    y_mod = exp(abs(y_mod.^2) + 1) - 2.718;

    %% Thresholding and separate note detection
    % Compute the moving average
    movingMean = 2 * movmean(y_mod, floor(Fs/10));

    % Get all the local minima, which indicate the thresholds
    minima = islocalmin(movingMean, 'MinProminence', mean(abs(y)));

    % Force detection of the beginning and end
    minima(1) = 1;
    minima(end) = 1;

    % Extract all the local minima points, throw away all zeros
    localMinima = find(minima);

    % Detect outliers and remove them
    outlierEdges = isoutlier(movingMean(localMinima), 'movmean',
        floor(Fs/20));
    actualLocalMinima = localMinima(outlierEdges == 0);

```

```
46      % Display number of notes for convenience
47      disp("Detected " + (length(actualLocalMinima) - 1) + " notes.");
48
49      % Zero padding length, for storing and FFT
50      maxLength = 2^nextpow2(max(actualLocalMinima.' - [0
            actualLocalMinima(1:end-1).']));
51      segments = [];
52
53      % Extract individual segments, by taking consecutive minima
54      for i = 1:length(actualLocalMinima) - 1
55          temp = y(actualLocalMinima(i):actualLocalMinima(i+1)-1);
56          temp = vertcat(zeros(maxLength - length(temp), 1), temp);
57          segments(:, i) = temp;
58      end
59
60      % Get all the frequencies in each note
61      % The maximum amplitude in the PSD is used to compute the
            dominant
62      % frequency
63      for j=1:length(actualLocalMinima) - 1
64          [freqs, PSD] = GetPSD(segments(:, j), Fs);
65          [~, index] = max(PSD);
66          notes(j) = round(abs(freqs(index)), 1);
67      end
68
69      %% Plotting
70      % Create the time axis
71      t = linspace(0, length(y) / Fs, length(y));
72
73      % Plot the segments in the audio, with the labelled frequency in
            each
74      % segment
75      figure('Name', 'Final output', 'NumberTitle', 'off');
76      plot(t, y, 'DisplayName', 'Time Domain signal');
77      xlim([min(t) - 0.05*max(t) 1.15*max(t)]);
78      ylim([min(y) - 0.4*max(y) 1.6*max(y)]);
79      xlabel('Time (s)', 'Interpreter', 'latex');
80      ylabel('Amplitude (V)', 'Interpreter', 'latex');
81      title('Segmented audio signal, with frequency of note detected',
            'Interpreter', 'latex');
82      h1 = legend('show');
83      set(h1, 'Interpreter', 'latex');
84
85      % Plotting the delimiting lines
86      for i = 1:length(actualLocalMinima) - 1
87          xline(t(actualLocalMinima(i)), '--k', string(i) + ". " +
                notes(i) + ...
88              " Hz", 'HandleVisibility', 'off', '
                    LabelHorizontalAlignment', 'center');
```

```
89      end
90      xline(t(end), '--k', 'DisplayName', 'Segment thresholds');
91  end
92
93  %% Function to denoise the input signal
94  function [de] = Denoise(noisy)
95      % applying wavelet decomposition to input signal, c is array of
            wavelet coefficients and l is level
96      [c,l] = wavedec(noisy,3,'db4');
97
98      N = length(noisy);                      % length of signal
99      sigma = median(abs(c))/(0.6745*3);  % MAD
100     lambda = sigma * sqrt(2*9*log(N));  % threshold value
101
102     % b is array of coefficients after thresholding
103     b = wthresh(c,'s',lambda);
104
105     % reconstructing the signal using coefficients and level
106     de = waverec(b,l,'db4');
107 end
108
109 %% Function to get the PSD of a signal
110 function [f_axis, PSD] = GetPSD(signal, Fs)
111     % Compute the frequency axis
112     f_axis = -Fs/2: Fs/length(signal) : Fs/2 - Fs/length(signal);
113
114     % Compute the FFT and then the PSD
115     signal_fft = abs(fftshift(fft(signal)/length(signal)));
116     PSD = signal_fft.^2;
117 end
118
119 %% Filtering
120 % Function to find the x% energy containment bandwidth of a given
        signal
121 function [hfreq] = EnergyContainmentBandwidth(y, percentage, Fs)
122     % Get the PSD of the signal
123     [~, psd] = GetPSD(y, Fs);
124
125     % Find the total power from the PSd
126     total_power = sum(psd);
127
128     % Account for even symmetry and convert to a percentage
129     target_power = (total_power/200) * percentage;
130
131     % Filter to get 95% containment bandwidth
132     dummy = 0;
133     index=1;
134     while index < length(psd)
135         dummy = dummy + psd(index);
```

```
136             if(dummy > target_power)
137                 break
138             end
139             index = index + 1;
140         end
141
142         % Return threshold frequency
143         hfreq = index * Fs / length(y);
144     end
145
146     % Function to create a parametric low pass FIR filter
147     % Created using FilterDesigner
148     function Hd = LowPassFilter(Fpass,Fstop,Fs)
149         Dpass = 0.057501127785;  % Passband Ripple
150         Dstop = 0.0001;          % Stopband Attenuation
151         dens  = 16;              % Density Factor
152
153         % Calculate the order from the parameters using FIRPMORD.
154         [N, Fo, Ao, W] = firpmord([Fpass, Fstop]/(Fs/2), [1 0], [Dpass,
                Dstop]);
155
156         % Calculate the coefficients using the FIRPM function.
157         b  = firpm(N, Fo, Ao, W, {dens});
158         Hd = dfilt.dffir(b);
159     end
```

9 References

1. Meaning of Prominence - MATLAB documentation
2. islocalmin function - MATLAB documentation
3. isoutlier function - MATLAB documentation
4. wavedec function - MATLAB documentation
5. waverec function - MATLAB documentation
6. wthresh function - MATLAB documentation
7. Wavelet Transformation
8. Audio Denoising reference
9. Wavelet transform images