Table of Contents

1. Analysis of corrupted audio signals

1.1. Signal in time domain

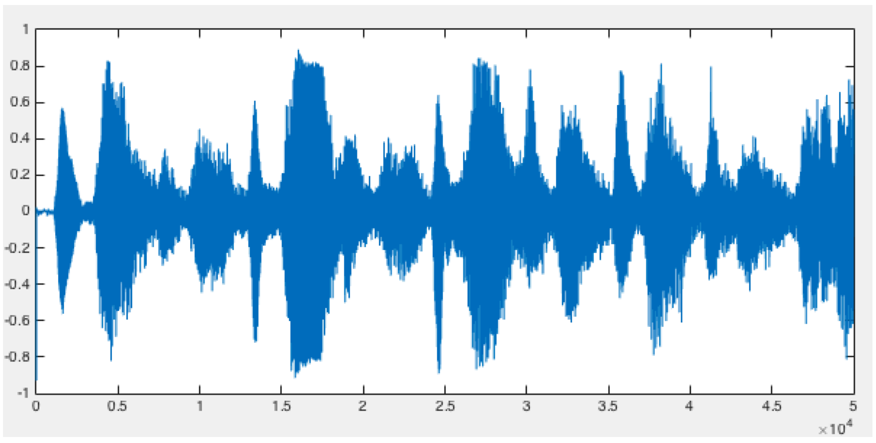


Figure 1:clean signal

Signal contains several distinct envelopes

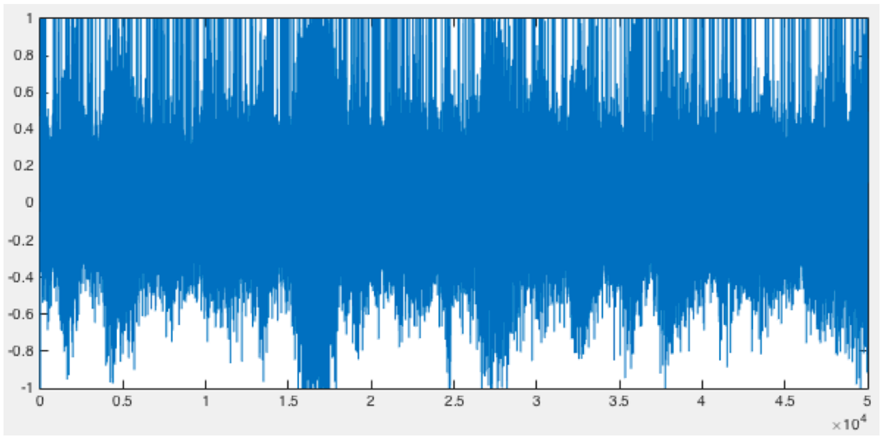


Figure 2: CH1 of corrupted Signal

The envelopes are no longer distinct. We also can observer noises that saturate the signal to value of 1.

1.2. Signal in frequency domain

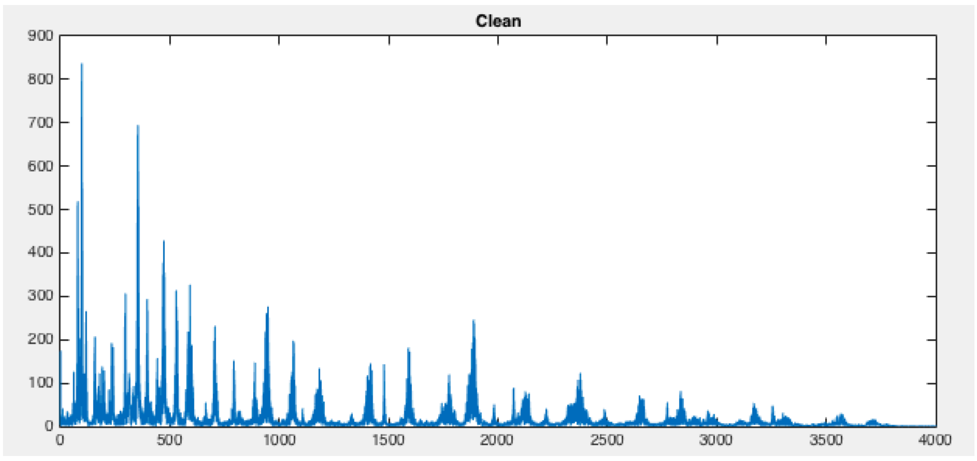


Figure 3: Clean signal in Frequency domain

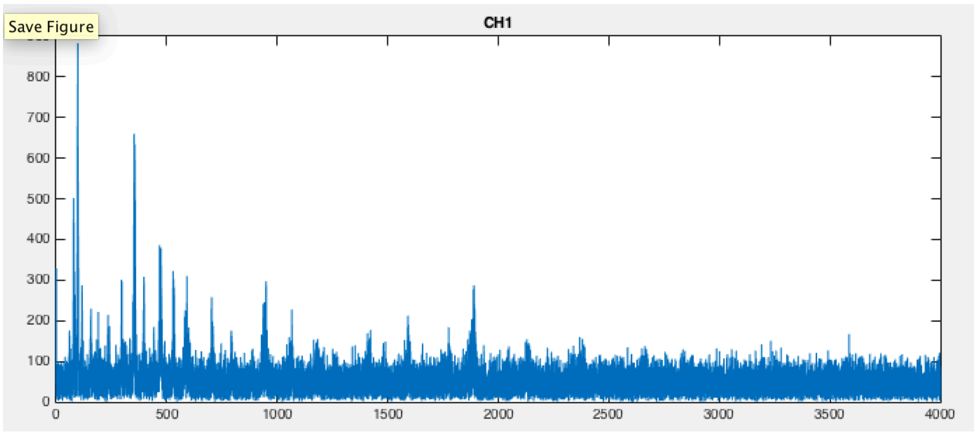
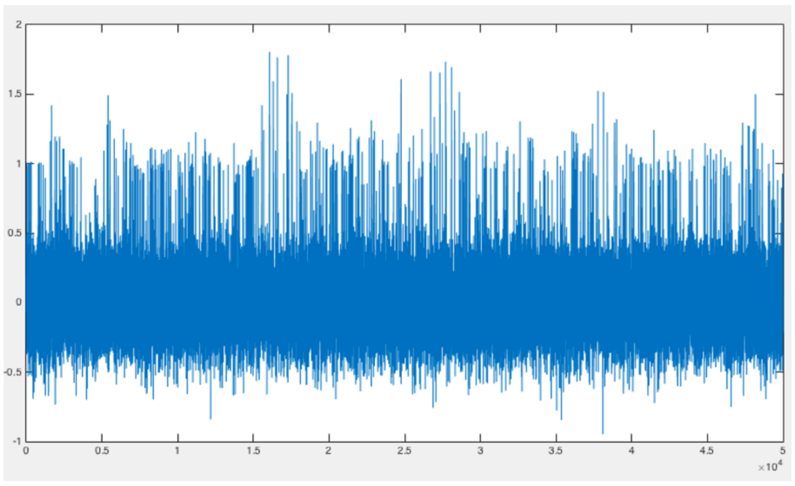
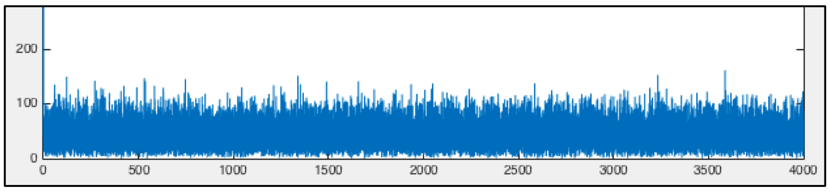


Figure 4: Corrupted signal (CH1) in Frequency domain

From FFT plot, most of the signal is completely buried in the noise spectrum. Magnitude of the high frequency components are significantly smaller than the noise in neighboring region.

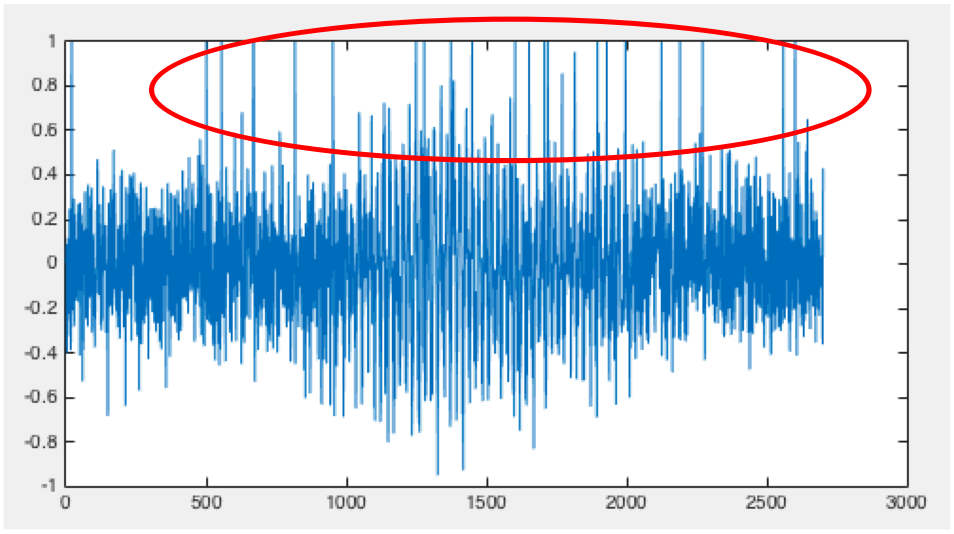
1.3. Identifying type of noises





From the FFT plot, we can see that the noise spread through out the frequency range. We hypothesize that there are 2 type of noise in the corrupted audio files: random noise and salt and pepper noise.

Salt and pepper noise can be seen as sharp spikes in the time domain waveform. This can be typically eliminated by using median filter., which will be discussed later in this report



Another, random noise, seen as a thick dense cluster in the time domain. It is rather interesting to know the statistical distribution and standard deviation of the noise. by plotting histogram, we can roughly estimate that the noise is normally distributed. (code: proj_151117_RandomNoise.m)

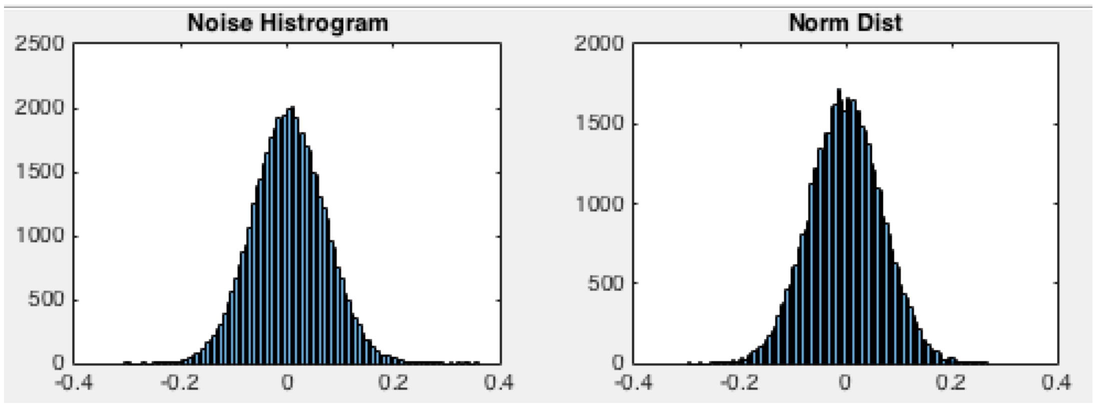
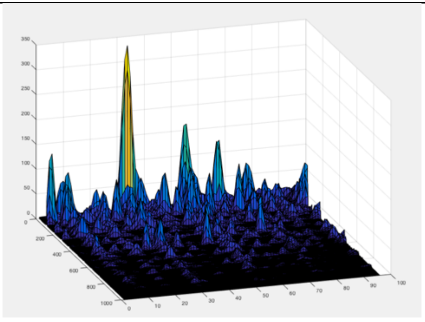
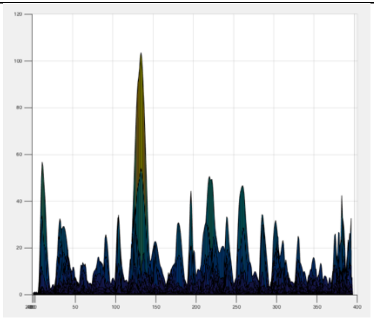
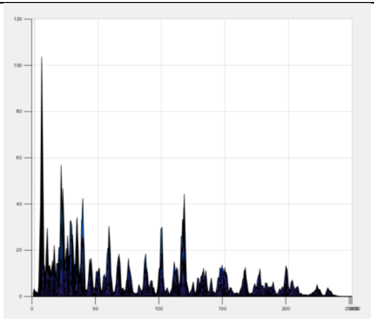


Figure 5: Comparison between noise histogram and normal distribution with the same standard deviation

1.4. 3D - Spectrogram

Spectrogram is another tool that that we implement to visualize the signal and plan how we are going to process it. This allow us to rotate and turn the 3D plot to see the amplitude in both frequency and time domain in the same plot.

We divide code into small overlap sections. Then, perform FFT on each section with windowing. We line up FFT results into large array and use surface 3D plot. (proj_151027_fn_3Dspectrogram.m)

 3D overview	 Time domain side	 Frequency domain side

```
function  fn_151111_Overlap3Dspectrogram(inAudio,fs,sectionLength)
audioLength = length(inAudio);
%%
sections = floor(audioLength/sectionLength) * 4 -4;
sectionOffset = sectionLength/4;
fftArray = zeros(sections,sectionLength/2);
sampleIndex = 1:sectionLength;
sampleSize = sectionLength;
frequencyIndex= linspace(0,fs,sampleSize);
fftIndex = 1:(size(sampleIndex,2)/2); %/2
for i = 1:sections
    sectionIndexOffset = (i-1) *sectionOffset;
    startAudioIndex = 1  + sectionIndexOffset;
    endAudioIndex = sectionLength  + sectionIndexOffset;
    audioIndex = startAudioIndex:endAudioIndex;
    fftSampleLength = length(audioIndex);
    fftAudioSample = inAudio(audioIndex);
    fftAudioSample = fftAudioSample .* hamming(fftSampleLength);
    fftAbsResult = abs(fft(fftAudioSample));
    minVal = min(fftAbsResult);
    normfftAbsResult =  fftAbsResult-minVal;
    fftArray(i,:) = permute(normfftAbsResult(fftIndex),[2,1]);
end
figure();
surf(fftArray);
```

2. Benchmarking:

2.1. mean square error (MSE)

Being able to identify the performance of filter or algorithm is crucial. Beside hearing, we need an indicator how close are we to the original signal. One method that we chose is calculating Mean Square Error (MSE). MSE value allow us to quantify the difference between the 2 signals. The higher the number, the larger the difference.

```
function [mse] = fn_151029_MSE(sample,ref)
    edge = 0;
    diff = sample - ref;
    sq = diff.^2;
    sumVal = sum(sq(edge+1:end-edge));
    mse = sumVal/(size(sample,1)-2*edge);
end
```

MSE value between the corrupted audio channels and the clean Audio is ranging from 0.0490 to 0.509. we will use this value as a guideline in each filtering process.

2.2. Loss of Signals/Information

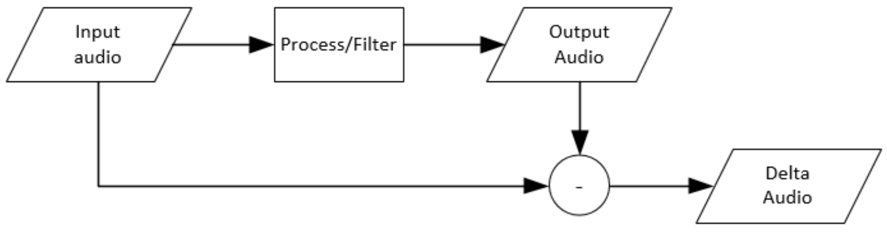
We might loose some signal component in the filtering process. To check that we did not loss any tune or tone, we get difference between the input and output of filter, by subtracting. The difference will when be what we extracted out, we can look into time domain and frequency domain that it does not contain any signal.

2.3. Hearing

Higher MSE value does not always guarantee the better audio quality. Some audio distortions are audible but does not contribute to higher MSE. We experience this some of the filters we implemented in later part.

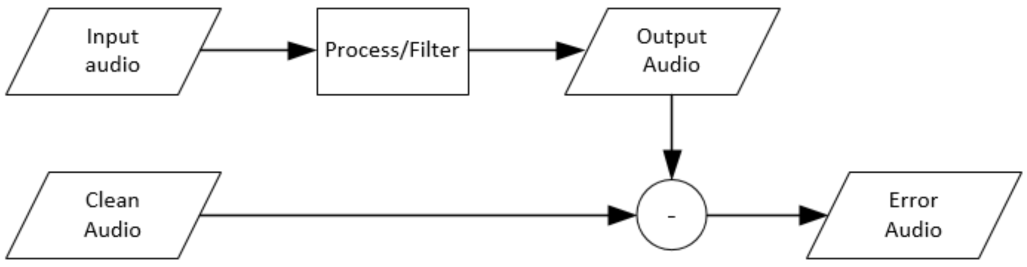
2.4. Delta Audio

"Delta audio" refers to the difference between the input and the output of the filtering process. It is important to analyze the delta audio because it contains components that we take out. Ideally, delta audio should only contain noise and should not contain any tune or rhythm from the music.



2.5. Error audio

"Error audio" let us hear the components of the clean signal that is differ from the output signal..



3. Global Filtering

3.1. Median filtering

Median filter is good in removing spike or sharp sudden value. This type of noise is often called salt and pepper noise. Median filter sort values neighboring region and take the middle one. The extreme value will be excluded from the process.

```
med3Audio = medfilt1(corruptedAudioArray,3,[],2);
%med5Audio = medfilt1(corruptedAudioArray,5,[],2);
```

Higher order of the median filter results in loss of high frequency component. With the order of 3, we achieve the best MSE value with better sound quality.

With median filter, the MSE value, compared with the clean signal, reduce from 0.05 to 0.016

3.2. Average filtering

Noise are random and uncorrelated to each other, while underlying repeated signals are correlate. In our case, we have 10 repeated signals from 10 channels with random noise in each of them. Simply, taking average sample by sample across the 10 channels will retain the underlying original signal. On the other hand, noise will be averaged out towards 0, given that it is normally distributed around 0.

```
avgAudio = sum(corruptedAudioArray,2)/audioChannelCount;
```

With averaging, the MSE value, compared with the clean signal, reduces from 0.05 to 0.0051

3.3. Combining both filers

```
med3AvgAudio= sum(med3Audio,2)/audioChannelCount;
```

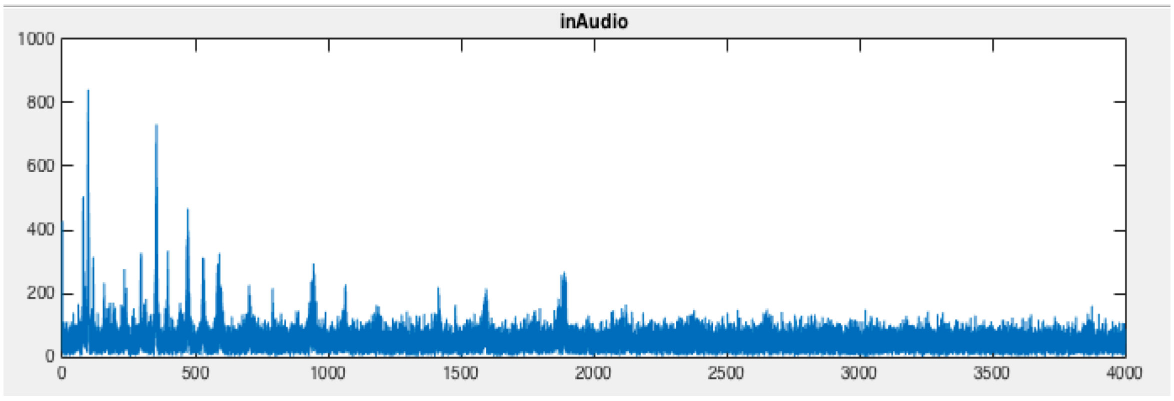
With Both filters, the MSE value (compared with the clean signal) reduces from 0.05 to 0.0046. An improvement than implementing each filter alone. (code: demo_151120_MSE.m)

4. Signal Filtering methods

4.1. FFT-based Frequency selection.

4.1.1. Idea

Based on the assumption that the signal component is significantly larger than the noise component, we can distinguish signal from noise based on FFT plot and design our filter according to this information. In the section of interest, we can identify significant frequency the components if the amplitude is more than certain threshold value and band pass these frequencies.



Often, the frequencies group together in narrow bands. Musical instrument produces a relatively low Q factor. Moreover, with string instruments like violin/cello using in this assignment, playing vibrato (twisting finger on the string) will cause fast frequency shift in narrow range. We create a function to detect the frequencies range and this will be used by filtering process later on. (code: fn_151126_fftBaseFreqWindows_Hm.m)

```
function [windowArray] = fn_151126_fftBasedFreqWindows_Hm(inAudio,fs,...
    sectionBoundary,fftTreshold)
```

4.1.2.Methodology

First, we perform FFT in the audio section of interest to obtain frequency components in that section.

Next, if any section of the FFT plot has magnitude higher than specified fftThreshold, we mark that particular frequency as signal. Noise signal with low amplitude will not be marked. This is the most essential part in selecting out the noise signal. The process loop through all frequency range.

```
%% find Signal Region in FFT
Y = fftAbsResult(fftIndex);
tempMaxArray = Y;
mask = zeros(length(Y));
for i = 1:length(Y)
 if tempMaxArray(i) > fftTreshold ;
     mask(i) = 1;
     end
end
```

Often, the signal component will cluster together in the narrow range of frequency. Once, we have marked all the signal frequencies, we create a range of selected frequencies. This frequency range will be use by the filter in later part.

4.1.3.Effect of changing fftThreshold value

FftThreshold value is significant factor in this filtering method. Too high threshold value will discard audio signal as noise. too low, the noise will not be removed and still audible in the output audio.

4.1.4.FFT Windowing

FFT algorithm assumes that the sample to be repetitive. Any discontinuity of one end of the signal with another will cause distorting in frequency domain output. Windowing provides an attenuation at both end such then they are at same level.

(code: demo_151130_filters.m -> fn_151126_SP_DS_FX_4.m -> fn_151126_fftBasedFreqWindows_Hm.m Line:40)

```
fftAudioSample = inAudio(fftSampleIndex);

    %windowing.
    %hamming( triang( blackman(  blackman( flattopwin( chebwin(
    %audioSample = audioSample .* hamming(sampleLength); %best sound
    %audioSample = audioSample .* chebwin(sampleLength); %best MSE
    fftAudioSample = fftAudioSample .* hamming(fftSampleLength);
    fftAbsResult = abs(fft(fftAudioSample));
```

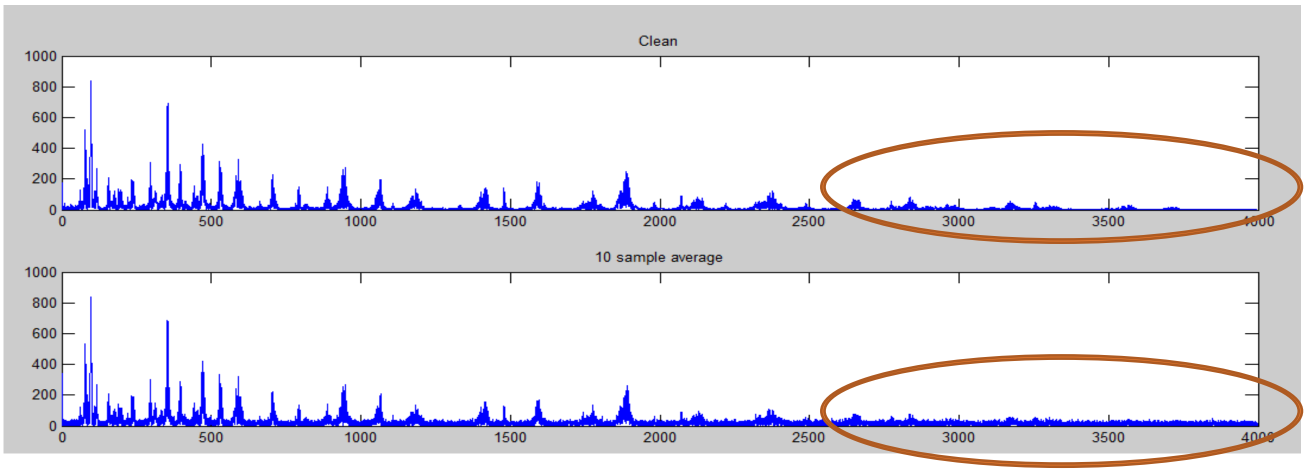
Results

Window	Rectangular	Triangle	Hamming	Blackman	chebwin
MSE	0.0036	0.0024	0.0026	0.0023	0.0022

Testing with different windowing method shows that chebwin give the lowest MSE. However, the window that give the best sound quality is hamming window and this is the window of our choice for performing FFT.

4.2. Including the harmonics in the frequency selection

When sound is produced by the music instruments, often, the harmonics of the fundamental frequency is also generated. These harmonics are lower in amplitude and usually buried in the noise. The amplitude is so low that we cannot detect using FFT graph and exclusion of these harmonics can cause distortion in the filtered audio track.



As an extension in the previous part, frequency selection, we now include frequencies at multiples of the frequency that we detect in FFT if the amplitude is significant.

```
          %include harmonics of the frequency
          if tempMaxArray(i) > fftTreshold*6 ;
              harmonics = 2;
              while i*harmonics < length(Y) && harmonics <6
                 mask(i*harmonics-(harmonics-2):...
                     i*harmonics+(harmonics-2)) = 1;
                 harmonics = harmonics +1;
              end
%
          end
```

The more harmonics we include; we can recover more signal component. On the other hand, it also includes more noise in output track. Choosing the number the right parameter by trial is needed.

As a result, with appropriate number of harmonics included, we can achieve a better audio quality and better MSE value with this addition of high frequency components.

4.3. Filtering from the frequency selection

```
function [outAudioSection,filterDestinationIndex]...
    = fn_151113_FIR_filter_window_2(inAudio,fs,sectionBoundary...
    ,windowArray,order)
```

Algorithm will loop through all the frequency range available and perform band-pass filtering for each range. In our end result, we use band pass FIR filter with order of 2000.

```
mixedOut = zeros(filterInputLength + outbound,1);
    for n = 1:size(windowArray,1)

        <filter code omitted>

        filteredSection = filter(b,a, paddedFilterInputAudioSection);
        mixedOut = mixedOut + filteredSection;

    end
outAudioSection = mixedOut(filterOutputIndex);
```

Many type of band-pass filter can be implemented and will be discussed in the later part of the report.

4.4. FIR Filtering

FIR filter design with Matlab offer linear phase response.

```
filterOrder = order;
filterDelay = filterOrder/2;
```

4.4.1.Implementation

We observed that the output of FIR filters has the group delay of N/2 samples, where N is the order of the filter. Delay need to be compensate at the output of the filter.

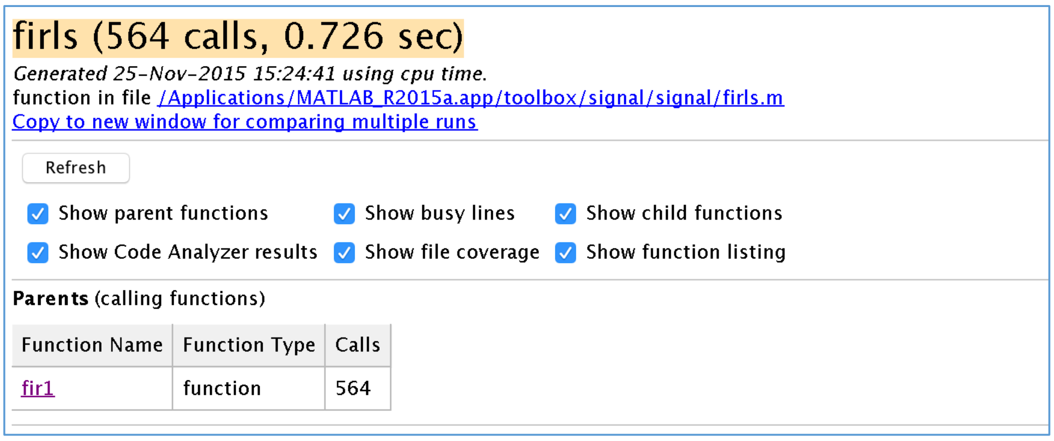
With ‘filter’ function of Matlab, we need to post-pad our input with data, as the delayed section will be truncated. If there is audio data available after the section, we choose to pad the data with that. If there is no audio data available, we decide to pad with 0 instead.

```
    paddingLength = 2000;
    %re-adjust if outbound (prepad)
    if (audioIndexStart <= paddingLength)
         paddingLength = audioIndexStart -1;
    end
    postPaddingLength = 2000;
    filterInputIndexEnd = audioIndexEnd+ postPaddingLength;
    outbound =  filterInputIndexEnd - 50000;

    %re-adjust if outbound
    if outbound > 0
        %pad with zero, if outbound
    else
        outbound =0;
    end
    filterInputIndexStart = audioIndexStart - paddingLength;
    filterInputIndexEnd = filterInputIndexEnd - outbound;
    filterInputIndex = filterInputIndexStart:filterInputIndexEnd;

    filterInputAudioSection = inAudio(filterInputIndex);
    paddedFilterInputAudioSection                                      =
[filterInputAudioSection;zeros(outbound,1)];
```

With modern computer we can afford the order of the filter to be as high as several hundreds to achieve steep cut-off frequency. Another feature to the code is that, if lowCutoff frequency is near to 0, the algorithm will switch from band-pass filter to low-pass filter.

```
if (lowCutoff < 0.1) %if less than 0.1 Hz, change to low pass
        b = fir1(filterOrder,highCutoff/(fs/2),'low');
        a = 1;
else
        b = fir1(filterOrder,[lowCutoff highCutoff]/(fs/2));
        a = 1;
end
filteredSection = filter(b,a, paddedFilterInputAudioSection);
```

Interestingly, we found that 'fir1' function in Matlab internally calls 'firls' (least square) routine to find the optimal coefficient for the band pass filter.



firls (564 calls, 0.726 sec)

Generated 25-Nov-2015 15:24:41 using cpu time.
function in file /Applications/MATLAB_R2015a.app/toolbox/signal/signal/firls.m
Copy to new window for comparing multiple runs

Parents (calling functions)

Function Name	Function Type	Calls
fir1	function	564

4.4.2. Effect of changing Order of the Filter

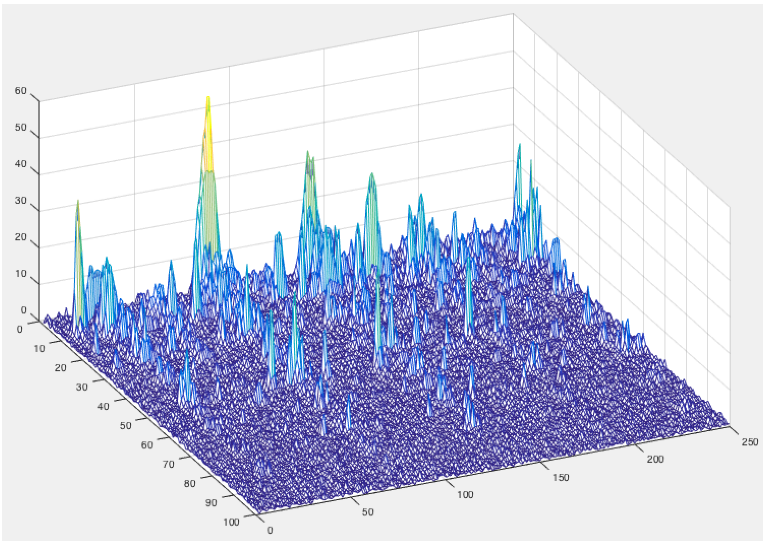
Order	100	250	500	1000	2000	3000
MSE	0.1612	0.0168	0.0041	0.0024	0.0021	0.0021

Despite better MSE value at higher MSE, we can hear some high frequency noise in the output signal. We decide to use filter of order 2000 for our output.

5. Sectional Filtering

5.1. Defining Sections

Using FFT spectrogram, we can see that the audio tracks does contain gaps in both frequency and time domain. The strategy is that we we can define the sections or area that there is no signal, we can clear out the noise in the section.



5.2. Sectioning in frequency domain

In the the audio tract we can hear 2 main parts of the audio, the base and the melody. The base section could come from the instrument like cello and the melody part from violin.

Clearly, both sections have different time that they play. It would be better if we process can separate the input audio into 2 output audio tracks. To verify that we did not introduce artifacts, we checked that both output audio tracts can be combined by summing and getting back the original input track. The problem remains that we need to pick the right frequency to split the audio.

Based on FFT graph, by trial-and-error, we separate the input at different frequencies with minimal amplitude and hear the output audio tracks.

5.2.1.Implementation

We created a function that use high order FIR filter (2000), to separate both channel.

```
function [filteredLow,filteredHigh] =
fn_151029_Split_2CH(inAudio,fs,tresholdFreq)
```

5.2.2. result

We found out that the best frequency to choose is 325 Hz, where we can hear base part and melody part play separately.

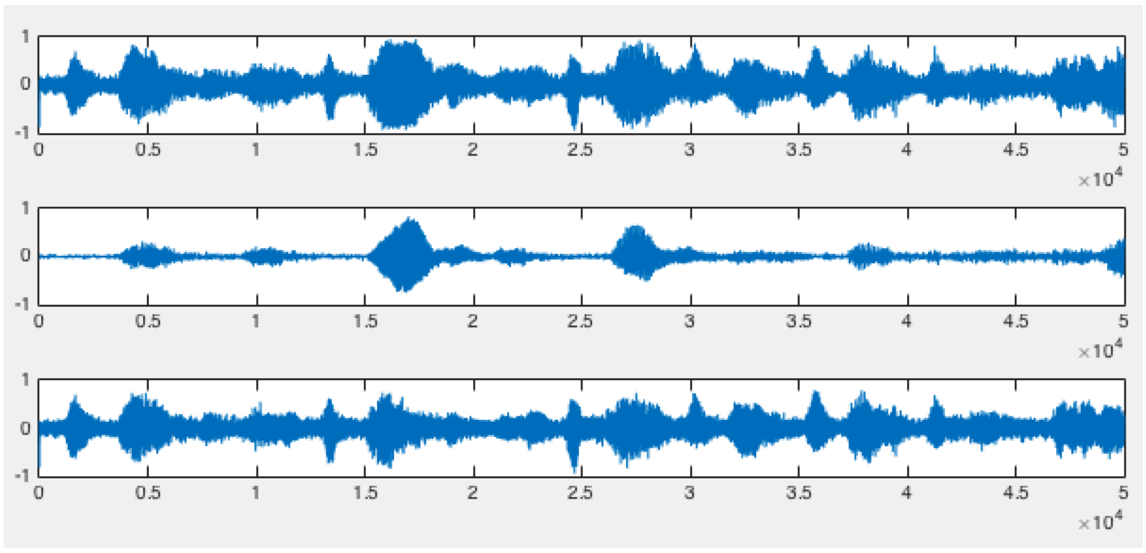


Figure 6: (top)input track (middle) low frequency track (below) high frequency track

5.3. Sectioning in Time domain

In different part of the audio, different notes are played and each of them has different frequency. It is a good idea that we process each section separately, such that we only band-pass frequency signals that is played in that section and filter out the noise in other sections at the frequency.

However, sectioning time domain introduce signal artifact at the border of the sections. To tackle with problem, we process the section in overlapping manner and stich them back together with weighing method.

```
function [outAudio] = fn_151113_OverlapProcessing(inAudio,...
    refAudio,fs,hWB)
```

we create small overlap sections and process them individually

```
%% Sections
sectionLength =1000;

sections = floor(audioLength/sectionLength) * 4 -3;
sectionOffset = sectionLength/4;
```

For each section, we band-pass only signal components. We replace the original tract with the filter output part by part. To improve continuity of the signal, we use windowing method. Middle part of the section is 100% replaced and the 2 edges has reduced weightage accordingly.

```
for i = 1:sections

    <omitted Filtering section>
    <outAudioSection is filter output>

     %Stiching filter output to the main track with windowing
     %triang hamming
    outputWindow = hamming(length(outAudioSection));
    outputWindowCompliment = 1 - outputWindow;

    outAudio(filterDestinationIndex)  =  outAudioSection  .*outputWindow  +
outAudio(filterDestinationIndex) .* outputWindowCompliment ;
end
```

Results

We can see improvement in MSE and hear improvement in audio quality after the processing. However, with high fftThreshold in the frequency selection process, we start to hear distortion in the audio.

As we need to lower fftTreshold and we using overlapping sections, the noise level in the output audio is still considerably high. we decide it as a pre-main filter.

The noise artifact is worse when we section when instruments is being played. This give us idea for the next processing method, which we will only dynamically section when the instrument is not playing. We will discuss about this method in the next part.

5.4. Amplitude-based Sectioning in Time domain

If we cut the track when a note is being played, we might introduce discontinuity in the audio in the section or lose some portion of the note. Therefore, we try to section the audio when no or minimal notes are being played.

We are trying to find points where the amplitude of the audio is low. First, we square the audio trace, since it swings into the negative range, this make easier for use to find local minimum. Another issue in finding local minimum is that the graph is highly fluctuating and often reach 0 value. To solve this, we applied the mean square in small sections.

```
function [sectionIndexes,sectionsCount] = fn 151029 DynSection(inAudio,fs)
```

```
sectionLength = 50;
sections = 1000;
MS = zeros(1,sections);
for i = 1:sections
    offset = (i-1) *sectionLength;
    audioIndexStart = 1 + offset;
    audioIndexEnd = sectionLength + offset;
    audioIndex = audioIndexStart:audioIndexEnd;
    sectionAudio = inAudio(audioIndex);
    MS(i) = sum(sectionAudio.^2,1)/sectionLength;
end
```

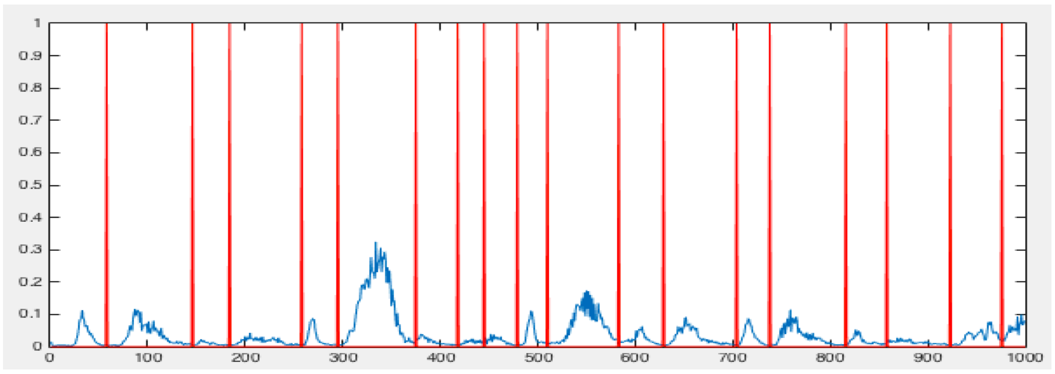
5.4.1.Finding Minimum

We use 'min' function to search for the local minimum. The search range parameter defines general size of each section. The larger the search range, the larger the section. By trial and error, we need to see from the graph for the optimal value that the audio track will be cut into appropriate sections.

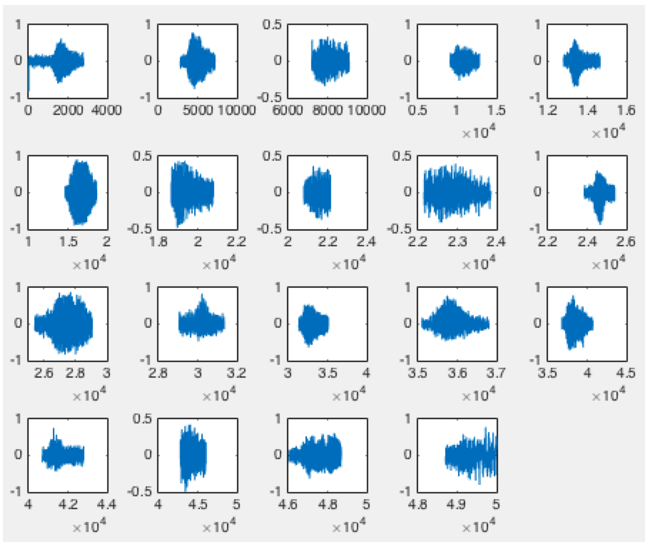
```
mask = zeros(size(MS));
searchRangePlus = 23;
searchRangeMinus = 20;
for i = 1+searchRangeMinus:size(MS,2)-searchRangePlus
    MSSection = MS(i-searchRangeMinus:i+searchRangePlus);
    [minVal,Index] = min(MSSection);
    if MS(i) == minVal
        mask(i) = 1;
    end
end
```

Once we find local minimum, we mark the index and generate the indexes of all the sections.

5.4.2.Results



The red line show how the algorithm divides the audio tracks into section according to the amplitude



Time domain plot of each section of the audio. Each section will then be filtered according to their frequency components separately to remove noise. Finally, each section will be put back together.

6. Combining Different Techniques (I)

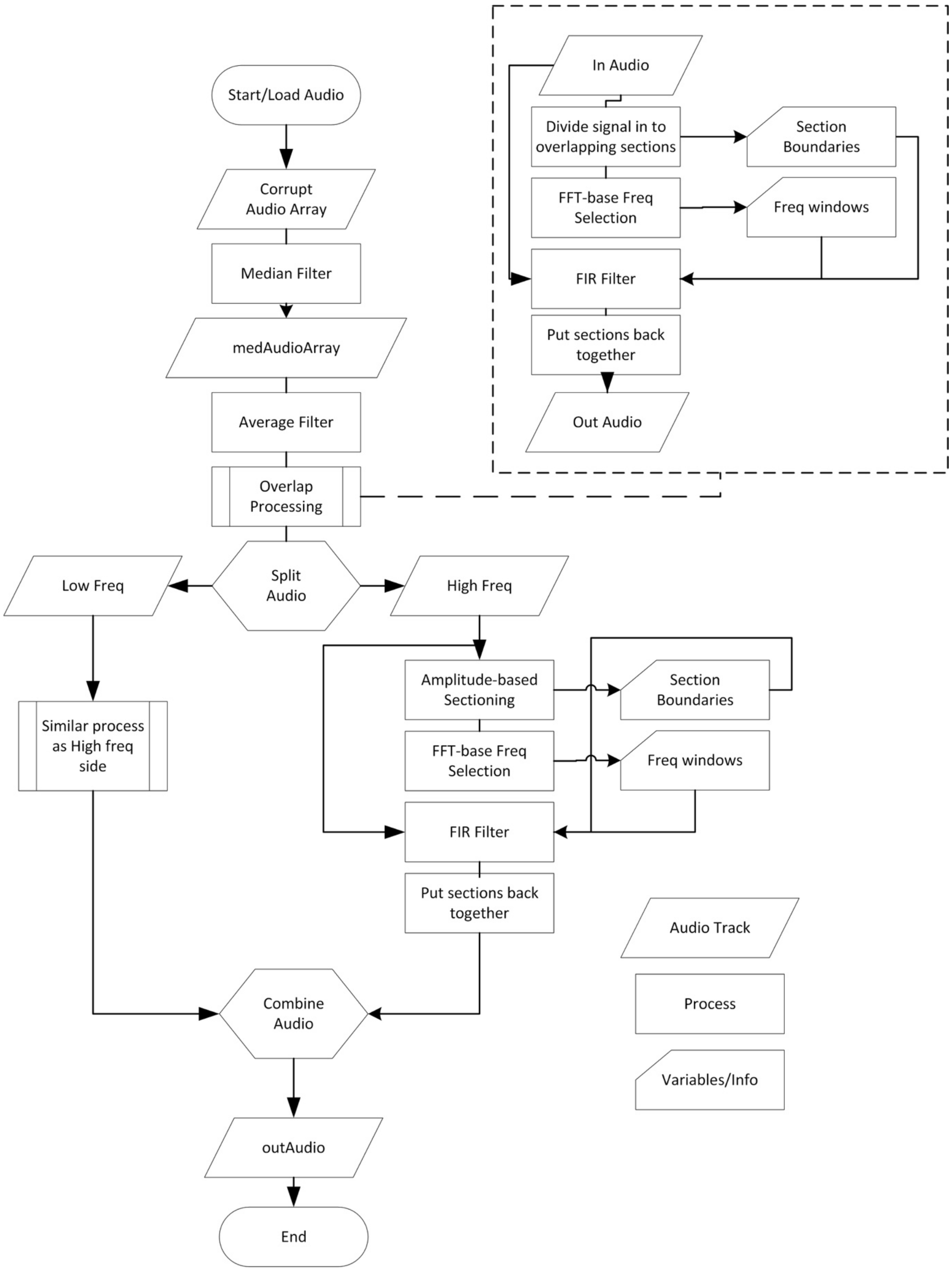
To achieve the best possible result, we decided to cascade multiple filters together. In doing so, it is not important to get most of the noise out in each step, but to maintain the signal information integrity. We can gradually remove noise from the signal in each filter and check if the signal that we removed contain any information.

First We apply median and average filtering to entire input signal. Noise is sufficiently removed for processing in the next part. Overlap Sections Filtering is then applied, where signal is cut into small sections and only band-pass frequency components with significant amplitudes. In this process, we choose fftThreshold value to be small and number of harmonics included to be large (10) such that we do not remove any small signal. Only noise will small amplitude will be removed. Some noise is still remaining audible at the output of this process.

Signal is then split in 2 sections, low frequency and high frequency parts. Each part will then be cut into sections according to their time-domain amplitude. Each section is then band-pass filtered for only signal with significant frequency-domain amplitude. Low frequency and high frequency part is summed to create output Audio.

(code: proj_151126_BestAudio.m)

6.1. Processing Flowchart



6.2. Results

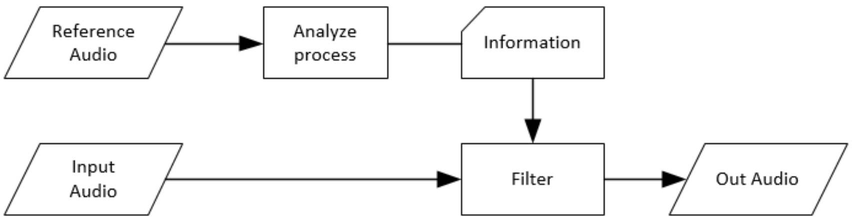
We are able to achieved very low MSE of 0.0018, which is better than implementing each individual method alone. We cannot hear Gaussian noise in the output. However, we can hear high pitch artifact that is introduced during processing. This may due to the method of cutting audio in time domain and put them back together and the edges are not continuous. Our attempt to remove this artifact will be done in the next section.

7. Combining Different Techniques (II)

With sectional processing in the previous part, we still getting small small noise/artifacts from putting the sections together in the final stage.

The noise/artifact is perceptible by hearing and very short in time domain. Overall, this audio track is very close to the original audio. we have an idea of using the output of the previous method as reference signal.

Reference signal is the signal that we use to analyze the frequency component and decide filter cutoff frequencies. Then we apply the filter in the target signal



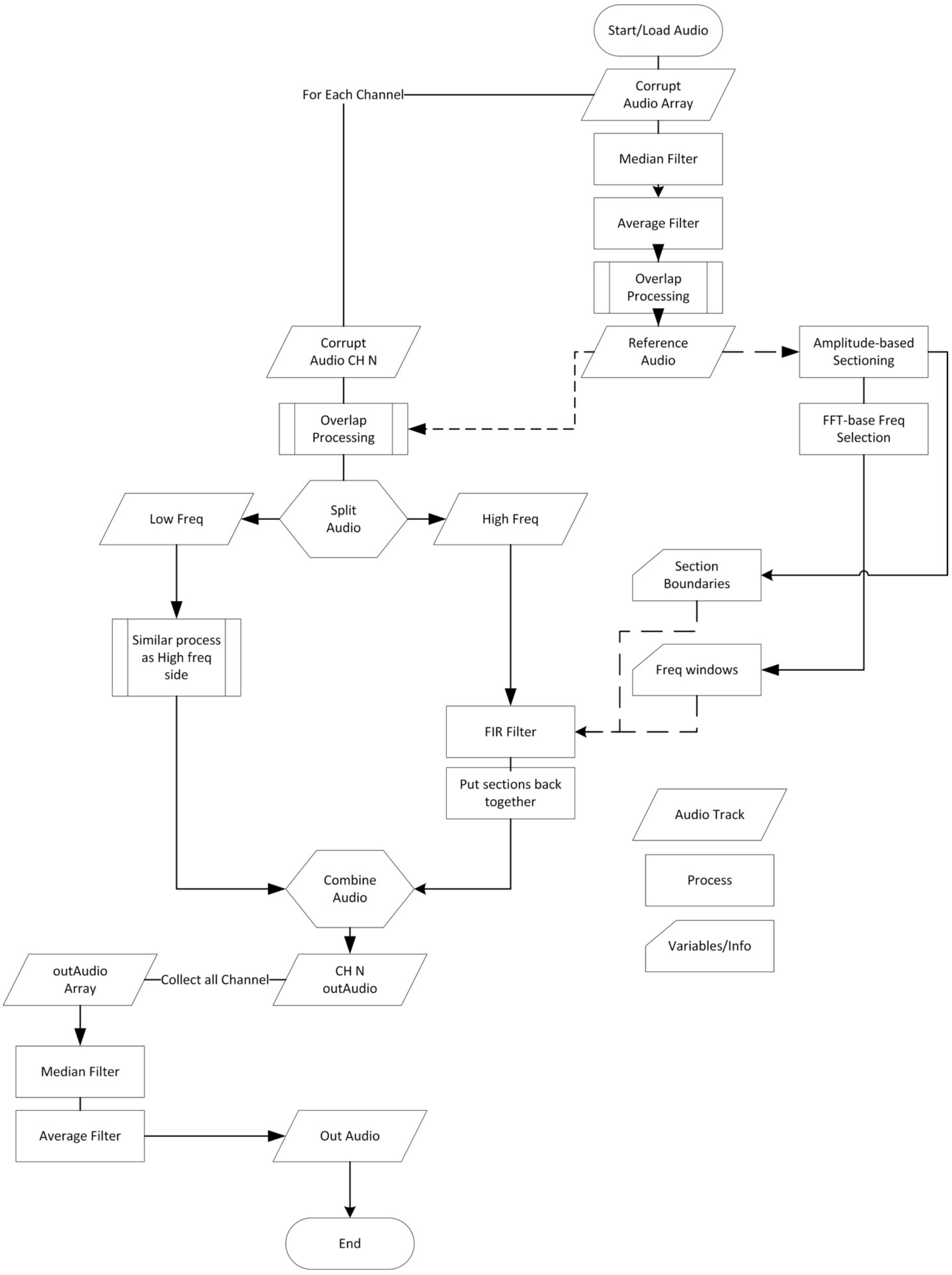
The final approach we applying here is to apply the filter on individual channel of the corrupted audio array based on the reference signal. Finally, we apply median filter and average filter on all the outputs to get our final output.

As a result, the noise/artifact is no longer audible. This maybe because difference input channel produce artifacts different pattern. Averaging and taking median would remove the difference. Even though, there is no significant change in MSE of 0.0019.

(Code: proj_151126_perChannelBestAudio.m)

(dependencies: fn_151029_loadSample.m, fn_151126_fftBasedFreqWindows_Hm, fn_151126_SP_DS_FX_4.m, fn_151029_DynSection.m, fn_151126_FIR_filter_window.m, fn_151126_OverlapProcessing.m, fn_151029_MSE.m, proj_151027_fn_3Dspectrogram.m, fn_151029_fftPlot.m)

7.1. Processing Flowchart



7.2. Results and analysis

We have achieved MSE value of 0.0019. Although, the value is higher than the previous part by 0.0001, the audio quality by hearing is much cleaner. We can no longer hear high pitch artifact at the junction of the sections

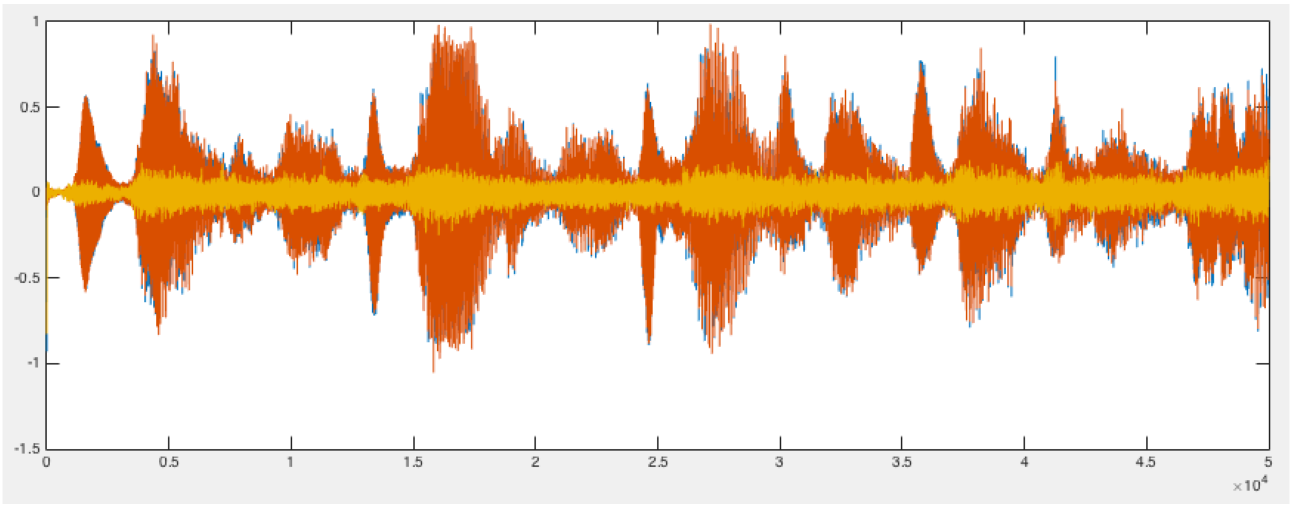
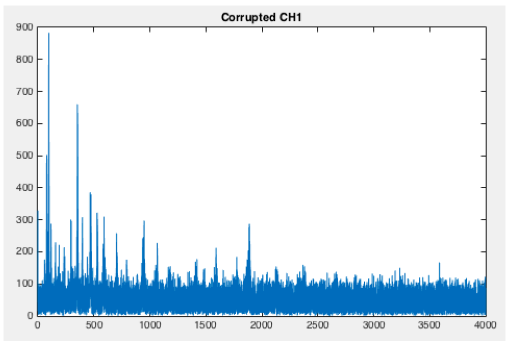


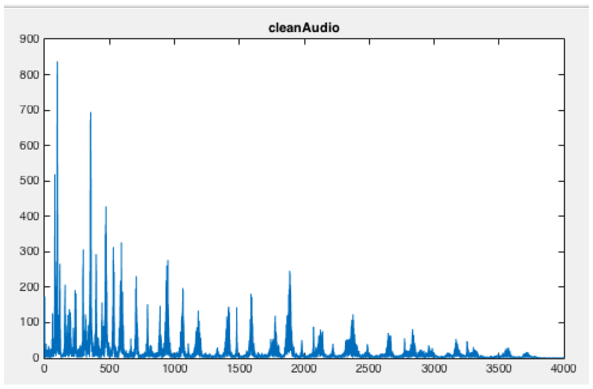
Figure 7 Blue: Clean Audio, Orange: output Audio, Yellow: difference

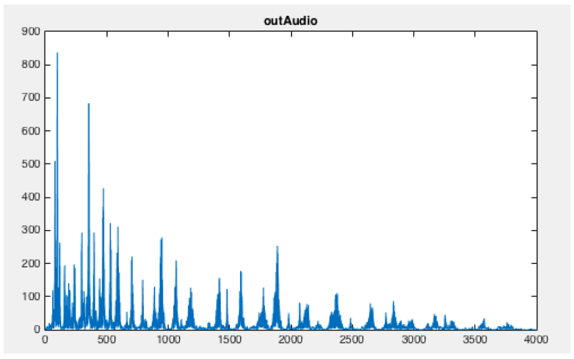
In my opinion, I hardly find any significant difference between the clean audio and the output audio.

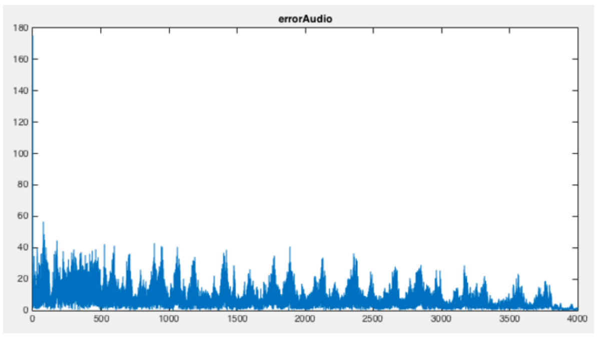
With errorAudio, we can hear some faint rhythm. This maybe due to loss of amplitude of the signal during filtering process. Amplitude response at the edge of bandpass does not have unity gain. From FFT plot, we can see that there are some frequency components in the difference. Their low amplitudes probably make them indistinguishable to the filtering algorithm.

FFT of clean Audio, output Audio and difference







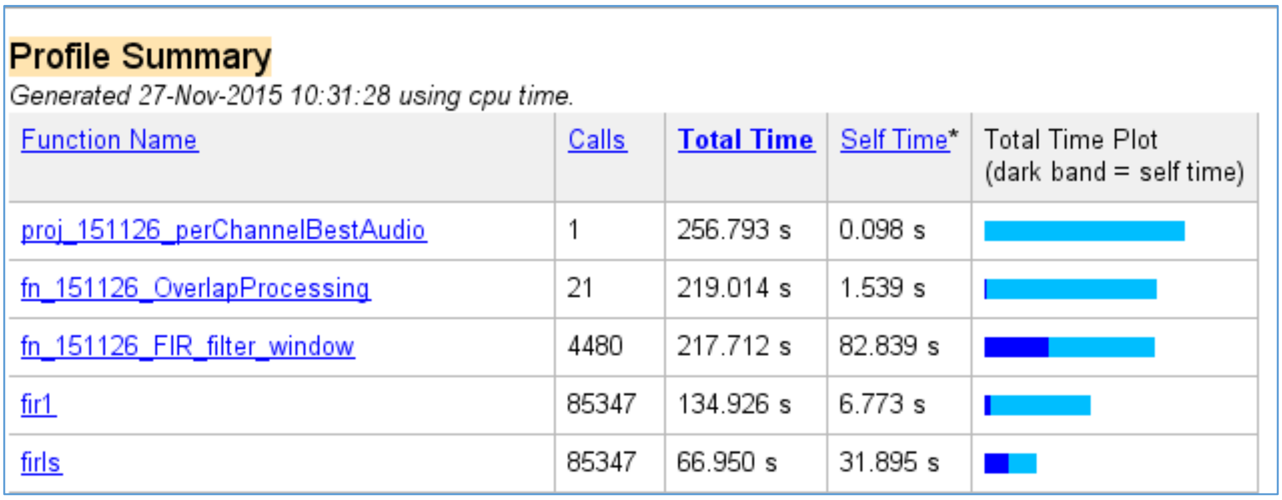


Judging from the FFT plot, the audio signal that we managed to recover from the corrupted audio array is remarkable. From the graph of the corrupted audio, any trace of audio signal beyond 2500 Hz is barely recognizable. Yet, we can extract high signal components in 3500 Hz range.

7.3. Limitations

This filtering method based on many trail-and-error of many parameters (fftThreashold, filterOrder, sectioningRange, windowing type, windowing parameters, number of harmonics included, etc.) and will work best with this audio track only. Additionally, finding the optimal values will take large amount of time to get output signal that closest to the original signal.

Filter design and filtering functions take most of the computing time and, with modern PC, this solution still not suitable for real-time application (maybe faster with C and speed optimization). There are more then 80000 filtering processes for 50000 points sample, or more than 4 minutes (Core i5 processor, Matlab 2015b) for 6.25 seconds of audio.



Profile Summary

Generated 27-Nov-2015 10:31:28 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
proj_151126_perChannelBestAudio	1	256.793 s	0.098 s	
fn_151126_OverlapProcessing	21	219.014 s	1.539 s	
fn_151126_FIR_filter_window	4480	217.712 s	82.839 s	
fir1	85347	134.926 s	6.773 s	
firls	85347	66.950 s	31.895 s	

8. Conclusion

In this project, we learnt to analyze and visualize the signal components (3D spectrogram), identify type of noises and plan for filtering strategy. Additionally, we have applied different techniques in choosing cut-off frequency for our filter. We also explored the effect of using different windowing methods and how it affects the output. We did introduce some high frequency artifact when tried to process the signal section by section and finally managed to remove the artifact by processing individual channel and apply filters afterwards. We have found that output closest to the clean signal achieved (lowest MSE), may not sound the best for the listener. Lastly we noted the limitations of our filtering methods.