AMATH 582 Homework 2: Gabor Transform and Spectrogram for Audio Analysis

Hongda Li

February 7, 2021

Abstract

In this paper, we are interested in applying the principal of Gabor filtering on real world audio data. Our objective is to utilize different filtering techniques using wavelets to extract out sounds for certain instrument in the audio data. In addition, we also discuss the best way of choosing parameters for Gabor filtering to get the best visualizations for the spectrogram. The results we present are musical notes of the guitar solo and bass solo for 2 rock songs.

1 Introduction and Overview

The objective is to reproduce the musical scores for a clip taken from the start of the song "*Sweet Child O' Mine*" by Guns N' Roses and a guitar solo from "Comfortably Numb" by Pink Floyd. We deploy the technique of first using a Shannon Filter on the whole audio to filter out only the frequencies for the instruments that we are interested in, using common knowledge and some basics in music, and then we use Gabor transform to find the best temporal and frequencies resolutions. Finally, we present a way to visualize the spectrogram, and an algorithm to identify the notes with some basics in music theory. In addition, we also explored ways of truncating the audio data so that it runs more efficiently.

2 Theoretical Background

The Gabor transform consider the additional usage of a kernel function in the time frequencies, preferably a function that has bounded L2 Norm over the real complex domain. And this is discussed more in details in [1], we will summarize the important piece here.

$$g_{\tau,w}(\tau) = \exp(i\omega\tau)g(\tau - t) \tag{1}$$

ω is the multiplier for the Fourier Kernel. g is a bounded function for filtering signal near t. A specific case of the Gabor transform is called the Short Time Fourier Transform(STFT), where we consider a real symmetric matrix to be the kernel. Together with the function $f(\tau)$ representing the signal, we have:

$$\widetilde{f}_g(t, w) = \int_{-\infty}^{\infty} f(\tau)g(\tau - t)\exp(-i\omega t)d\tau \tag{2}$$

Notice that, the variance under the time and frequencies domain are bounded by some constant. This means that, shortening the width of the kernel will reduce Frequency resolution while increasing the with will reduce time resolution. The problem can be addressed with some oversampling to produce better visualization to compensate. In addition, we consider the usage of several wavelet function for filtering in the time and frequency domain of the signal.

$$g(\tau; t, w) := \exp\left(-\left(\frac{\tau - t}{\frac{\sqrt{2}w}{2}}\right)^2\right) \tag{wavelet.G}$$

$$g(\tau; t, w) := \begin{cases} 1 & |\tau - t| \le w/2 \\ 0 & |\tau - t| > w/2 \end{cases} \qquad \text{(wavelet.S)}$$

Where each of the wavelet function (Will be referred to as filter in the text) has one time parameter τ as the input, and its location and with is controlled by the parameters: t, w. In the case of the Gaussian Filter, w means a distribution with $2w$ standard deviation. For computational purposes, the filtering processing using wavelets are implemented as element-wise vector multiplications and the STFT is implemented via FFT with a for loop that slide the kernel through the domain.

3 Algorithm Implementation and Development

Two core parts of the algorithm is the parameters for the STFT, the implementations of STFT and the conversion between time domain vector and the frequencies domain vector, other aspects of the algorithm will be discussed as we view the results. The whole procedure of the processing any given audio can be summarized by:

1. Setting up the parameters for the spectrogram, choosing a section of the music to analyze. Implemented in HW2Demo1 and HW2Demo2.
2. FFT on the data, filter out a range of frequencies that the instrument is plying in with the Shannon Filter.
3. Create the spectrogram, truncate it to get the best view for the musical scores, and find out the peaking frequencies.

3.1 Time to Frequencies Domain Conversion

The time domain vector is by a vector with the same length as the number of floats in the audio data. As common knowledge, digital audio data usually are sampled 480000 times second (Uncompressed m4a format).

Algorithm 1: Converting Time Domain to Frequencies Domain

```
1: Input: TimeVec
2: n := length(TimeVec); L := TimeVec(end) - TimeVec(1);
3: if n is even then
4:     hz:=$\frac{2\pi}{L}[0 : n/2 - 1, -n/2 : -1]$
5: else
6:     hz:=$\frac{2\pi}{L}[0, 1 : (n-1)/2, -(n-1)/2 : -1]$
7: end if
8: hz:= $\frac{\text{hz}}{2\pi}$
9: Output: FFTshift(hz)
```

Notice that, depending on the parity of the width of the signal, we need to partition the corresponding frequencies domain differently. In the case of even number of partitions, we want 0 appears at index $n/2 - 1$ after the fftshift, assuming index start with 0, and we want 0 to be at index $\lfloor n/2 \rfloor$ after fftshift assuming index starts with 0. Most importantly, we need to divide it by 2π to actually obtain the frequencies for the audio.

Implemented in: TVecToHz.

3.2 Creation of spectrogram Using STFT

The next part is the creation of the spectrogram. We implement the width of the filter to be relative to the window's size, and the window's size is simply the total length divides by the number of partitions we

Algorithm 2: Creating spectrogram with STFT

```
1: **Input:** TimeVec: Tvec, Audio: A, RelWidth: W, Number of Partitions: N, WaveletFunction: g
2: **Initialize:** SpectroMatrix
3: dt := (Tvec(end) - Tvec(1))/n
4: Hz := Input Tvec into Algorithm 1
5: for II = 0:N - 1 do
6:     t := Tvec(0) + II*dt
7:     F := g(Tvec; Tvec(1) + II*dt + dt/2, W*dt)
8:     Signal := F*A
9:     Signal := fftshift(fft(Signal))
10:    Signal := Filter out excessive frequencies using Hz vector.
11:    SpecMatrix(:, II + 1) = Signal
12:    SpecMatrix(:, II + 1) = Normalize the II + 1 th row of SpecMatrix, and put it into log scale
13: end for
14: Output: Tvec, Hz, SpecMatrix
```

want. This is convenient when we sync up the STFT with the bars of music, which helps us get better results and visualization. This is implemented with SpectroGram.m class and CreateSpectrogram.m function.

This algorithm (Algorithm 2) is generic for all audio and wavelet function: A, and g. The variable "RelWidth:W" represents the with of the wavelet relative to the with of the partition of the signal in time domain, this makes things easier to tweak, in addition, if "$N \times W$": is set to be a constant, then the amount of oversampling is kept unchanged.

In addition, extra processing (line 12 in Algorithm 2) is involved to make better presentation on the spectrogram, trimming of frequencies that is negative and outside the range of the instrument we are interested in, it is then presented on a log scale to scale down the relative sizes of the frequencies coefficients making some of the features of the spectrogram more visible.

3.3 Global Frequencies Filtering and Audio Truncations

By common knowledge, music are written in bars of notes in Chromatic Scale or the Diatonic Scales for pop songs (Diatonic Scale is in the Chromatic Scale). Each note in Chromatic Scale increments by a multiplier of $2^{1/12}$ in term of frequencies (A semi-tone). An instrument such as guitar usually operates inside a range of 3 octaves (12 Semi-tone per octave).

The reasoning of filtering out the frequencies for just the instrument makes it easier to identify particular instruments using the spectrogram. Implementations wise this is achieved via applying a Shannon Filter to the FFT of an audio section we are interested in, and then Inverse Transform it back. The process acts as a band pass filter and it isolate one of the instruments from other instruments. Implemented in OctaveFilter.m

Audio is truncated either by seconds, or by BPM (Beats per minute), both are implemented. The reasoning is that, music repeats around a theme with some variations, and they are located in sections of bars. Truncating the music in terms of bars reduce loads for computations, it also makes better visualization for the spectrogram.

4 Computational Results

4.1 GNR: Notes and Spectrogram

For the clip from GNR (*Sweet Child O' Mine*), a shannon filter located at the interval $[220, 800]$ is applied, it covers the alto an tenor range of the music, the expected range for the guitar. There are 8 bars of music in total, each 2 bars repeats and builds on the same motif, the first 3 sections build on variations, while the 4th one repeats the first 2 bars.

After running the routine highted in the previous section, the result is shown in figure 1. The melody is written on the C# major key. The music score is: C#3,C#4, G#3, F#3,F#4, G#3, F4, G#3, the 2 bars is then repeated with variations on the first, second and 4th keys.

Computationally, The first bars of music is partitioned into 128 sections, and the Gabor kernel is the Gaussian function with a relative width of 4, this allows the audio to be super-sampled, allowing for an smoother image. Other wavelets for STFT is experimented however Gaussian and Shannon filter was proven to produce the best spectrogram and the most accurate results.

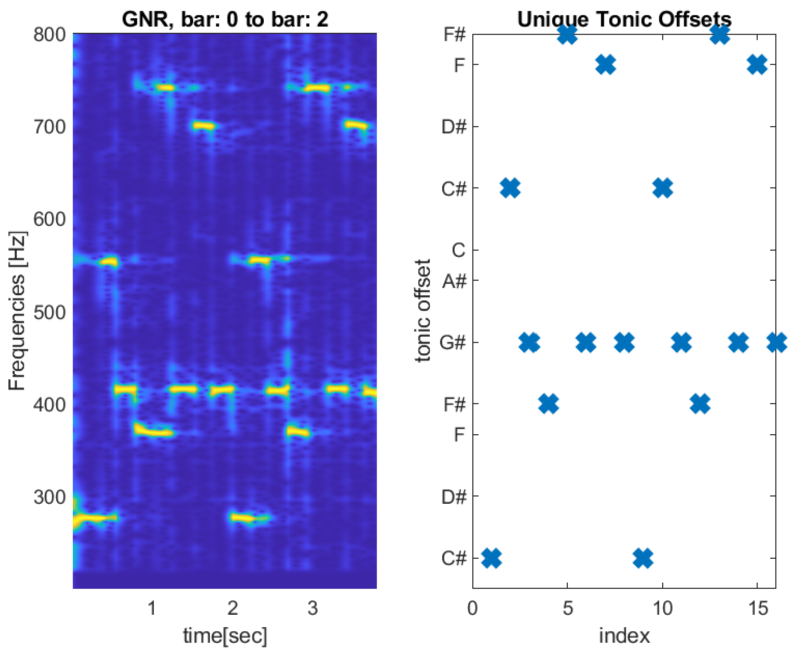

Figure 1: GNR: Spectrogram of the Guitar, Gaussian Wavelet

4.2 Floyd: Bass Notes and Spectrogram

The bass of the song "*Comfortably Numb*" performs a Basso Obstinate on the G minor scale. A Shannon Filter is used globally to filter out frequencies in the $[50, 110]$ hertz Because that is the expected acoustic range off the instrument. The result is presented in Figure 2. It shows one bars of the score that is played byt the bass which is repeated through out the audio clip. The music score is: D1, B1, A1, G1, F1, D1. However, it's unclear whether the key B, and D are presented at the same time to create a harmony.

For this example, both the Gaussian and the Shannon Wavelet function is deployed. Take note that Figure 2 Left shows the spectrogram when the Guassian Wavelet is used and Figure 2 Right shows the spectrogram when Shannon Wavelet is used. Both Clearly identifies the same frequencies content, however it visually Shannon Wavelet produces sharper image.

The audio clip is partitioned into 64 sections. (Observe how much bigger it is compare to audio clip); and a relative with of 4 is chosen. The choice is justified by the fact that lower acoustic signal need longer window to identify.

4.3 Floyd: Guitar Solo and Spectrogram

In this section, we will present the spectrogram for the guitar solo in the audio clip. However, due to the virtuosity of the guitar soloist, the music score cannot be presented. More sophisticated algorithm is need to identify complex harmony and melodic contour, unfortunate it's not as simple as STFT.

The routine is run on the whole music clip for Floyd's "Comfortably Numb" on the acoustic range of the guitar via global filtering, and the results in shown in: Figure 3. Please observe the complexity of the melodic contour, and how simple the previous 2 examples are. In this case, multiple notes seems to be played

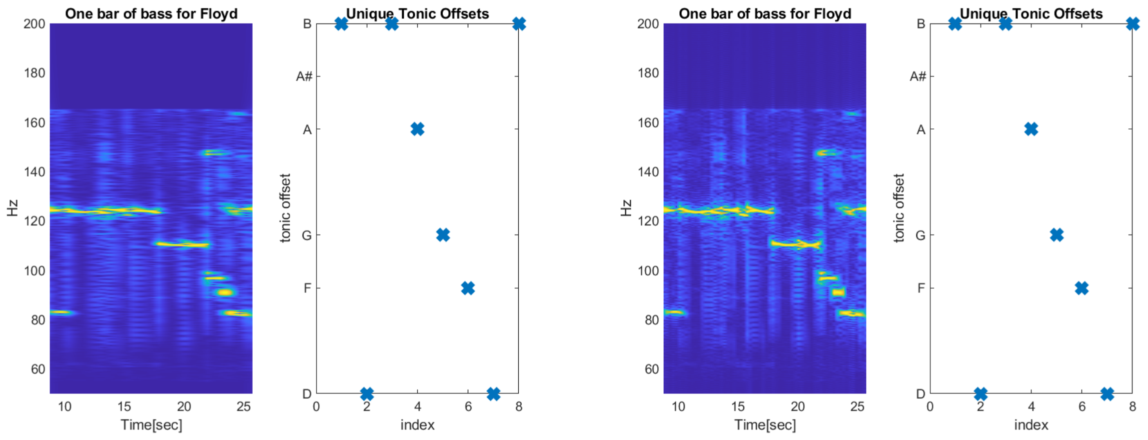

Figure 2: Floyd's Bass, Gaussian and Shannon Wavelet

at the same time. The notes are Legato instead of staccato, creating significant challenge for reading the notes from the spectrogram. Simply trick by pulling out the frequencies with the maximal magnitude is not feasible anymore.

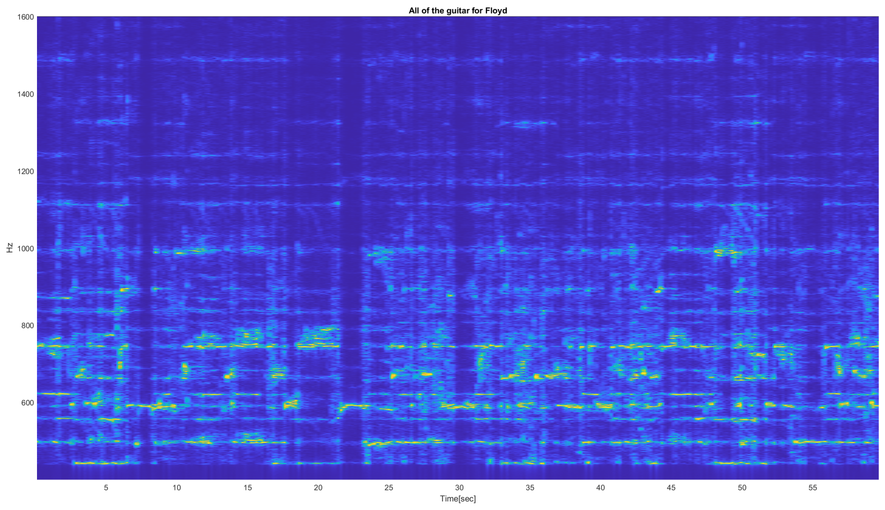

Figure 3: Floyd whole Song Guitar with Shannon Wavelet

5 Summary and Conclusions

The algorithmic frameworks for the STFT can be easily implemented in high level scientific programming language. The visualization process is however, where mostly the works comes in. To get the best out of the digital signal, we need domain specific knowledge for analysis, trial errors to figure out the best window with and discretization of the signal. However, it also shows us that, when dealing with more complex melodic contour and musical articulations, significant challenge is placed on the process of interpreting the spectrogram as music score.

References

[1] Jose Nathan Kutz. *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press, 2013, p. 324.

Appendix A MATLAB Functions

1. HW2Demo1.m:
   This file demonstrate the main routine for producing results for GNR's Song.
2. HW2Demo2.m:
   This file demonstrate the main routine for producing the retuls for Floyd's "Comfortably Numb" .
3. ProblemParam.m
   This file is a class that package all the parameters that are associated with reading the audio files, and truncating the audio file. It also computes the time domain vector and the frequencies domain vector.
4. SpectroGram.m
   This file contains all the parameters that are associated with a given spectrogram. Filter width, and the number of spatial discretization.
5. TVecToHz.m
   This is a function that convert the time domain discretization vector into its shifted frequency domain vector.
6. ShannonFilter.m
   A function for the Shannon Filter.
7. Play.m
   A function that play a given chunk of audio from the clip.
8. PeakingFrequencies.m This file extract out the peaking frequency for each window for the STFT.
9. OctaveFilter.m
   This is a function that filters away the frequencies by octave given the root note for any given signal.
10. Normalize.m
    This function will normalize real and complex vector.
11. HatFilter.m
    Mexican Hat filter.
12. GFilter.m
    The Guassian Filter.
13. CreateSpectrogram.m Given the a signal and the time domain vector together with an instance of the SpectroGram class, this function crate a all the results needed for visualizing the Spectrogram.

And this is the github link that contains all the cource codes and results in the paper: Github Repo.

HW2Demo1.m

```
%% GNR PRODUCE SPECTROGRAM: 2 BARS EACH
% Good Parameters for GNR:
%   bpm: 126/8, Chunk: 1, chuncation: 128, filterwidth: 4, filter: Gauss
%   bpm: 126/8, Chunk: 2, chunkation: 16, filterwidth: 2, filter: Guass
%   bpm: 126/8, Chunk: 3, chunkation: 16, filterwidth: 2, filter: Gauss
%   bpm: 126/8, Chhnk: 4, chunkation: 16, filterwidth: 2, filter: Gauss

CHUNK = 1;            % <- Choose which 2 bars of music you want here
WAVELET = @GFilter;  % <- Change your type of wavelet here.

p = ProblemParam(126/8, "GNR.m4a");
sp = SpectroGram(128, 4, [200 800]);

%
% -------------------------------------------------------------------------
% 1. Construct all parameters needed for the music.
% 2. Filter out the frequencies for the instrument.
% -------------------------------------------------------------------------

[original, t, hz] = p.getChunk(CHUNK);
filtered = OctaveFilter(t, original, 220, 3);
Play(p, CHUNK);

% -------------------------------------------------------------------------
% 1. Plot the filtered sound waves.
% 2. Plot the fft of filtered sound wave.
% 3. Plot the original sound waves.
% 4. Plot the fft of the original sound wave.
% -------------------------------------------------------------------------
figure(1)
subplot(4, 1, 1); plot(t, Normalize(filtered));
filteredHz = fftshift(abs(fft(filtered)));
subplot(4, 1, 4); plot(hz, Normalize(filteredHz));

subplot(4, 1, 3); plot(t, Normalize(original));
originalHz = fftshift(abs(fft(original)));
subplot(4, 1, 2); plot(hz, Normalize(originalHz));
title("Global Frequencies Filtering");

% -------------------------------------------------------------------------
% 1. Visualizing the spectrogram.
% -------------------------------------------------------------------------
[m, hzvec, spectroTvec] = CreateSpectrogram(filtered, t, sp, WAVELET);
figure(2);
subplot(1, 2, 1);
pcolor(spectroTvec, hzvec, m); shading interp;
title(strcat("GNR, bar: ", num2str((CHUNK - 1)*2), ...
    " to bar: ", num2str(CHUNK*2)));
xlabel("time[sec]"); ylabel("Frequencies [Hz]");


% -------------------------------------------------------------------------
% Getting the keys from the spectro analysis.
% 1. Trim off the repreated peaking frequencies.
% ------------------------------------------------------------------------
[tonicOffSet, peakingFreq] = PeakingFrequencies(sp, 261.63);
notes = [];
for II = 1: length(tonicOffSet)

   if length(notes) == 0
       notes(1) = tonicOffSet(II);
   else
       if notes(end) == tonicOffSet(II)
           continue;
       end
       notes(end + 1) = tonicOffSet(II);
   end

end
h = subplot(1, 2, 2);
plot(notes, "x", "markersize", 10, "linewidth", 4);
title("Unique Tonic Offsets"); xlabel("index"); ylabel("tonic offset");
set(h, "Ytick" , [1, 3, 5, 6, 8, 10, 11, 13, 15, 17, 18], ...
    "YtickLabel", ["C#", "D#", "F", "F#", "G#", "A#", "C", "C#", "D#", "F", "F#"]);
saveas(gcf, "gnr-spectro", "png");
```

HW2Demo2.m

```
%% SETTING THINGS UP

WAVELET = @ShannonFilter;
p  = ProblemParam(64/32, "Floyd.m4a");

%% WHOLE SONG GABOR TRANSFORM: FOR BASS
% 1. Set up the spectrogram settings.
% 2. Slide up the music audio
% 3. Play it.
% 4. Crate spectrongram base on the globally filtere audio
% 5. Plot it.
% ------------------------------------------------------------------------
sp = SpectroGram(64 , 2, [50 200]);          %  <-- Change your spectrogram settings here.
[original, t, hz] = p.slice(1, 1);
filtered = OctaveFilter(t, original, 55, 1); %  <-- Change your global frequencies filter setting here.
playblocking(audioplayer(Normalize(filtered), p.SampleRate));
[m, hzvec, spectroTvec] = CreateSpectrogram(filtered, t, sp, WAVELET);
figure; pcolor(spectroTvec, hzvec, m); shading interp;
xlabel("Time[sec]"); ylabel("Hz");
title("The Bass for Floyd");

%% JUST ONE OF THE TIME FRAME TRANSFORMED: FOR BASS
sp = SpectroGram(64, 4, [50 200]);
[original, t, hz] = p.timeSlice(8.63, 25.9);
filtered = OctaveFilter(t, original, 55, 1);
playblocking(audioplayer(Normalize(filtered), p.SampleRate));
[m, hzvec, spectroTvec] = CreateSpectrogram(filtered, t, sp, WAVELET);
subplot(1, 2, 1); pcolor(spectroTvec, hzvec, m); shading interp;
```

```
xlabel("Time[sec]"); ylabel("Hz");
title("One bar of bass for Floyd");

% Figuring out the notes
[tonicOffSet, peakingFreq] = PeakingFrequencies(sp, 110);
notes = [];
for II = 1: length(tonicOffSet)

   if length(notes) == 0
       notes(1) = tonicOffSet(II);
   else
       if notes(end) == tonicOffSet(II)
           continue;
       end
       notes(end + 1) = tonicOffSet(II);
   end

end
h = subplot(1, 2, 2);
plot(notes, "x", "markersize", 10, "linewidth", 4);
title("Unique Tonic Offsets"); xlabel("index"); ylabel("tonic offset");
set(h, "Ytick" , [-5 -3 -2  0  1 2], ...
    "YtickLabel", ["D" "F" "G" "A" "A#" "B"]);

saveas(gcf, "floyd-bass-spectro", "png");
%% WHOLE SONG GABOR TRANSFORM: FOR THE GUIARGER: FOR GUITAR
sp = SpectroGram(300, 2, [400, 1600]);
[original, t, hz] = p.slice(1, 1);
filtered = OctaveFilter(t, original, 440, 3);
% playblocking(audioplayer(Normalize(filtered), p.SampleRate));
[m, hzvec, spectroTvec] = CreateSpectrogram(filtered, t, sp, WAVELET);
figure; pcolor(spectroTvec, hzvec, m); shading interp;
xlabel("Time[sec]"); ylabel("Hz");
title("All of the guitar for Floyd");

%% JUST ONE OF THE TIME FRAME TRANSFORMED FOR THE GUIARGER: FOR GUITAR
sp = SpectroGram(64 , 2, [400 1600]);
[original, t, hz] = p.timeSlice(8.63, 25.9);
filtered = OctaveFilter(t, original, 440, 2);
playblocking(audioplayer(Normalize(filtered), p.SampleRate));
[m, hzvec, spectroTvec] = CreateSpectrogram(filtered, t, sp, WAVELET);
figure; pcolor(spectroTvec, hzvec, m); shading interp;
xlabel("Time[sec]"); ylabel("Hz");
title("A bar of guitar for Floyd")
```

ProblemParam.m

```
classdef ProblemParam < handle
    % All parameters regarding the problem will be packed here.
    % This class will cut the audio data into chunks.
    properties
        SampleRate;  % The sample rate associated with the files, floats/sec
        AudioData;   % Audio data as a series of floats.
        TotalTime;   % How many seconds does the audio sample have in total.
        ChunkSize;    % How many floats per chunk?
        ChunkTime;    % How many second does one chunk of data span
        TotalChunks; % total number of chunks on the signal
        BPM;          % Beats Per minutes OR Bars per minites. (Depending on caller's input)

        TimeVec;      % The Time vector for the whole auidio data.
        Hz;           % The frequencies vector for the whole audio data.
    end

    methods
        function this = ProblemParam(bpm, fileName)
            [this.AudioData, this.SampleRate] = audioread(fileName);
            this.ChunkSize = (60/bpm)*this.SampleRate;
            this.BPM = bpm;
            TotalTime = length(this.AudioData)/this.SampleRate;
            this.TotalTime = TotalTime;
            ChunkTime = length(this.AudioData)/this.ChunkSize;
            this.ChunkTime = ChunkTime;
            this.TotalChunks = floor(length(this.AudioData)/this.ChunkSize);
        end


        function [chunk, timevec, hz] = getChunk(this, i)
            % Get chunks according to BPM, one chunk is one beat.
            % chunk:
            %    float row vector, sliced from audio data.
            % timevec:
            %    spatial time domain vector, start with zero.
            % hz:
            %    The frequencies vector, not shifted.

            i     = i - 1;
            Start = i*this.ChunkSize + 1;
            Start = floor(Start);
            End   = (i + 1)*this.ChunkSize;
            End   = min(length(this.AudioData), ceil(End));
            chunk = this.AudioData(Start: End);
            chunk = chunk.';
            if nargout >= 2
               Timespan = length(chunk)*(1/this.SampleRate);
               timevec  = linspace(0, Timespan, length(chunk) + 1);
               timevec  = timevec(1: end - 1);
            end
            hz = TVecToHz(timevec);
        end

        function [chunk, timevec, hz] = slice(this, i, total)
            % Cut the signal into a total amount of time and then slice out
            % the ith one only.
            chunkSize = round(length(this.AudioData)/total);
            timeSpan  = chunkSize*(1/this.SampleRate);
            timevec   = linspace(0, timeSpan, chunkSize + 1);
            timevec   = timevec(1: end - 1);
            hz        = TVecToHz(timevec);
            chunk     = this.AudioData((i - 1)*chunkSize + 1: ...
                min(i*chunkSize, length(this.AudioData)));
            chunk     = chunk.';
        end

        function [chunk, timevec, hz] = timeSlice(this, startT, endT)
            startIndex = ceil(startT*this.SampleRate);
            endIndex  = round(endT*this.SampleRate);
            chunkSize = endIndex - startIndex + 1;
            timevec   = linspace(startT, endT, chunkSize + 1);
            timevec   = timevec(1: end - 1);
            hz        = TVecToHz(timevec);
            chunk     = this.AudioData(startIndex: min(endIndex, length(this.AudioData)));
            chunk     = chunk';
        end

        function n = totalChunk(this)
            N = length(this.AudioData);
            n = N/this.ChunkSize;
        end

    end
end
```

SpectroGram.m

```
classdef SpectroGram < handle
    % This is a class that models the spectrogram, and parameters for the
    % Gabor filtering.
    %
    % It will stores the:
    %   1. Frequencies Axis.
    %   2. The spectrogram Matrix.
    %   3. The Time vector.
    %   4. Spatial discretization.
    %   5. Gobar filter width.

    properties
        Hz;     % The frequencies vector, for the spectrogram
        Tvec;   % The time domain vector, for the spectrogram
        Spec;   % The Spectrogram matrix.

        N;      % The spetial domian discretization of the signal.
        Width;  % The width of the gabor filter.

        FreqCutoff;
                % The frequencies that we are going to retain for the spec,
                % in the form of [low, high].
        PeakingFreq;
                % An vector representing the frequencies that has the max
                % magnitude.
        TonicOffSet;
                % How many semitone above/befow from the given tonic key.
    end
    methods
        function this = SpectroGram(chunkcation, filterwidth, freqcutoff)
            if length(freqcutoff) ~= 2 || freqcutoff(2) <= freqcutoff(1)
               error("Not valid input. ")
            end
            this.N = chunkcation;
            this.Width = filterwidth;
            this.FreqCutoff = freqcutoff;
        end

    end
end
```

TVecToHz.m

```
function hz = TVecToHz(tvec)
    % Given a time domain, return a shifted frequencies domain.
    n = length(tvec);
    L = tvec(end) - tvec(1);
    if mod(n, 2) == 0
        hz = (2*pi/L)*[0: n/2 - 1, -n/2: -1];
    else
        hz = (2*pi/L)*[0, 1:(n - 1)/2, -(n - 1)/2:-1];
    end
    hz = hz/(2*pi);
    hz = fftshift(hz);
end
```

ShannonFilter.m

```
function F = ShannonFilter(center, w, domainvec)
    F = (abs(domainvec - center) <= w/2);
end
```

Play.m

```
function null = Play(p, i)
    [audio, ~, ~] = p.getChunk(i);
    playblocking(audioplayer(Normalize(audio), p.SampleRate));
    null = 0;
end
```

PeakingFrequencies.m

```
function [tonicOffSet, peakingFreq] = PeakingFrequencies(spectroGram, tonickeyFreq)
    %  This function finds the tonic offset vector and the peaking
    %  frequency vector from trhe given instance of SpectroGram.
    %

    SpectroMatrix = spectroGram.Spec;
    SpecHz = spectroGram.Hz;
    for II = 1:size(SpectroMatrix, 2)
       FreqDomain = SpectroMatrix(:, II);
```

```
        [~, MaxIndex] = max(abs(FreqDomain));
        peakingFreq(II) = SpecHz(MaxIndex);
    end
    tonicOffSet = round(12*(log2(peakingFreq) - log2(tonickeyFreq)));

    spectroGram.PeakingFreq = peakingFreq;
    spectroGram.TonicOffSet = tonicOffSet;
end
```

OctaveFilter.m

```
function filtered = OctaveFilter(tvec, chunk, rootnote, octave)
    % Filter out the frequencies for a given chunk of music from the
    % Audio.
    % p:
    %   an instance ofthe ProblemParam.
    % i:
    %   The index of the chunk of music that we are interested in, OR, the
    % signal we are trying to decompose.
    % rootnote:
    %   The tonic key and it's also the lowest key.
    % octave:
    %   The number of octave you want to go up from the tonic key.
    chunkFFT = fftshift(fft(chunk));
    hz = TVecToHz(tvec);
    kFilter = (abs(hz) < rootnote*(2^octave + 1)).*(abs(hz) > rootnote);
    chunkFFTFiltered = kFilter.*chunkFFT;
    filtered = real(ifft(ifftshift(chunkFFTFiltered)));
end
```

Normalize.m

```
function v = Normalize(vec)
    v = vec/max(abs(vec), [], "all");
    if sum(imag(v)) ~= 0
        v = abs(v);
    end
end
```

HatFilter.m

```
function F = HatFilter(center, w, domainvec)
    %  Filter a signal with a mexican hat filter.
    phi = @(t) (1 - t.^2/w^2).*exp((-t.^2/w^2)/2);
    F = phi(domainvec - center);
end
```

GFilter.m

```
function F = GFilter(center, w, domainvec)
    % Give a center and the with of the Guassian Filter, return a vector
    % that is representing the filter.
    %
    % Center:
```

```
    %       Where the guassian filter is located at.
    % w:
    %       The width of the guassian filter.
    % domainvec:
    %       A domain vector represent where the filter filtering. It can be
    %       both in the frequency, or the signal domain.
    f = @(x) exp(-(x - center).^2/(w*(sqrt(2)/2))^2);
    F = f(domainvec);
end
```

CreateSpectrogram.m

```
function [specMatrix, hzvec, t] = CreateSpectrogram(signal, tvec, sp, FilterFunc)
    % returns a matrix, which is ready to be displayed via pcolor.
    % Matrix columns are stacked with frequencies vector.
    % signal:
    %   A series of floats that we want to view process.
    % p:
    %   An instance of the class: SpectroGram, it contains the parameters
    %   we need for the visualization.

    chunkation = sp.N;
    filterWidth = sp.Width;
    freqthreshold = sp.FreqCutoff;
    hzvec = TVecToHz(tvec);
    dt = (max(tvec) - min(tvec))/chunkation;

    low = freqthreshold(1); high = freqthreshold(2);
    indexStart = find(hzvec > low); indexStart = indexStart(1);
    indexEnd = find(hzvec > high); indexEnd = indexEnd(1);

    tstart = tvec(1);
    for II = 0: chunkation - 1
        F = FilterFunc(tstart + II*dt + dt/2, filterWidth*dt, tvec);
        signalFiltered = F.*signal;
        signalFilteredFFTshifted = fftshift(fft(signalFiltered));
        specMatrix(:, II + 1) = signalFilteredFFTshifted(indexStart: indexEnd);
        t(II + 1) = tstart + II*dt + dt/2;
    end
    specMatrix = abs(specMatrix)./max(abs(specMatrix), [], 1);
    specMatrix = log(specMatrix + 1);  % Put into log space.

    % specMatrix = specMatrix(indexStart: indexEnd, :);
    hzvec = hzvec(indexStart: indexEnd);

    % Assign into the parameter object, state mutated.
    sp.Spec = specMatrix;
    sp.Hz = hzvec;
    sp.Tvec = t;
end
```