AMATH 482 Homework 2

Vineet Palepu

February 10, 2021

1. Abstract

In this paper, we are trying to find the music notes played by certain instruments in audio files that we are given. We use the Gabor transform to construct spectrograms to view the frequencies present in each file over time. We then tried to filter out the bass guitar and its overtones from one of the songs in order to isolate the lead guitar solo. By applying a filter to each of the time points, we were somewhat successful at recovering some of the notes of the stronger and sustained notes in the solo.

2. Introduction and Overview

In this paper, we are given two samples of music, and tasked with finding the notes played by various instruments in each sample. The first sample is a selection from *Sweet Child O' Mine* by Guns N' Roses. It is around 14 seconds long and consists only of a guitar melody. The second selection is a clip from *Comfortably Numb* by Pink Floyd, and is nearly 60 seconds long, and contains a bass guitar, as well as a lead guitar solo.

In order to isolate the instruments when necessary, and to determine what notes are being played, we will be making extensive use of the Gabor Transform, a modification to the Fourier Transform, which is also being used. Through the Gabor Transform, we will generate a spectrogram of the entire audio clip, from which strong frequencies corresponding to notes played by certain instruments can be extracted.
In addition to this, in order to isolate the instruments, we will make use of the band pass filter and an inverted Gaussian filter in order to remove overtones.

3. Theoretical Background

3.1. Fourier Transform

Before we get to the Gabor transform, we must first cover what the Fourier transform does, as the Gabor Transform makes extensive use of it. If we are given a function $f(x)$, its Fourier transform $\hat{f}(k)$ can be calculated as shown in Equation (1). This transform allows us to transform a function from its original domain, in our case, the time domain, into the frequency domain. There also exists an inverse Fourier transform which allows us to return to original domain.

$$\hat{f}(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-ikx} dx \tag{1}$$

Of note is the fact that the Fourier transform requires continuous data on an infinite domain. As this is rarely the case, we turn to the discrete Fourier transform (DFT). The DFT allows us to supply a sequence of values at linearly spaced points and from that can calculate the relative occurrence of frequencies. It can be calculated as shown in Equation (2).

$$\hat{x}_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{\frac{2\pi ikn}{N}} \tag{2}$$

Finally, in practice, the fast Fourier transform (FFT) is often used as it has the same requirements of the DFT, but offers better performance, with a computational complexity of $\mathcal{O}(n \cdot \log n)$ compared to the DFT's $\mathcal{O}(n^2)$. When working with large datasets and using the FFT repeatedly, as is the case with the Gabor transform the increase in performance is significant.

3.2. Gabor Transform

One of the major limitations with the Fourier transform is that while it can provide precise information about the frequencies present in a time series, it provides no information about when those frequencies occur. Thus, for our purposes of reconstructing the musical score, the Fourier transform alone is not enough.
Thus, we turn to the Gabor transform. The Gabor transform applies the Fourier transform to specific subsets of the data, sliding from start to end, thus giving frequency information localized to specific points in time, hence its other name the short-time Fourier transform.

$$\tilde{f}_g(\tau,k) = \int_{-\infty}^{\infty} f(t)g(t-\tau)e^{-ikt}dt \tag{3}$$

Given a filter function $g(t)$, which in our case we take to be the Gaussian filter, and a time point τ around which we center the filter, we can calculate the Gabor transform as shown in Equation (3). Doing so will give us information about the frequencies present in the function $f(t)$ around the region of τ. Again however, this cannot directly be applied to data, so we use the discrete Gabor transform.

$$\tilde{f}_g(m,n) = \int_{-\infty}^{\infty} f(t)g(t-nt_0)e^{2\pi i m\omega_0 t}dt \tag{4}$$

$$m,n \in \mathbb{Z}, k = m\omega_0, \tau = nt_0$$

The discrete Gabor transform is given in Equation (4), along with several constraints. Important to note is that parameters m,n must now be integers, and our parameters k and τ must now be multiples of ω_0 and t_0, the frequency and time domain resolutions. Furthermore, in order to be able to reproduce the original signal, the window given by the filter function must be large enough to have overlap, and the resolutions must be relatively small.

3.3. Gaussian Filter

Although we have many choices for filter functions, the Gabor transform uses a Gaussian filter. A filter is a function that can be applied to a signal to change it somehow. Filters can be applied in both the time and frequency domain, but here we apply it in the time domain. A Gaussian filter is a function of the form given in Equation (5).

$$g(t) = e^{-a(t-\tau)^2} \tag{5}$$

In the Gaussian filter, the width is specified by a, with larger values producing a wider filter and smaller values narrowing it. The center of the filter is specified by τ, which is the point around which we want to analyze the frequencies. Examples of the Gaussian filter can be seen in Figure 1.

The further away from τ, the more the signal gets attenuated. This is how the Gabor transform is able to localize frequencies. When the Gaussian filter is applied to the data in the time domain, only data points nearby to the center of the filter remain, with points far outside the window attenuated to near zero. Thus, when we apply the Fourier transform, it effectively ignores data outside the window.

3.4. Spectrograms

Using the Gabor transform, we can effectively view the frequencies present at a specific point in time. However, if we wanted to view the frequencies present as time passes, we would have to animate them, which is less than ideal. Instead of displaying the frequencies on the x-axis and the amplitude on the y-axis, we instead display the frequencies on the y-axis and place τ on the x-axis. To view the amplitude, we map the values to certain colors, allowing us to display three dimensions of data on a 2-dimensional medium.

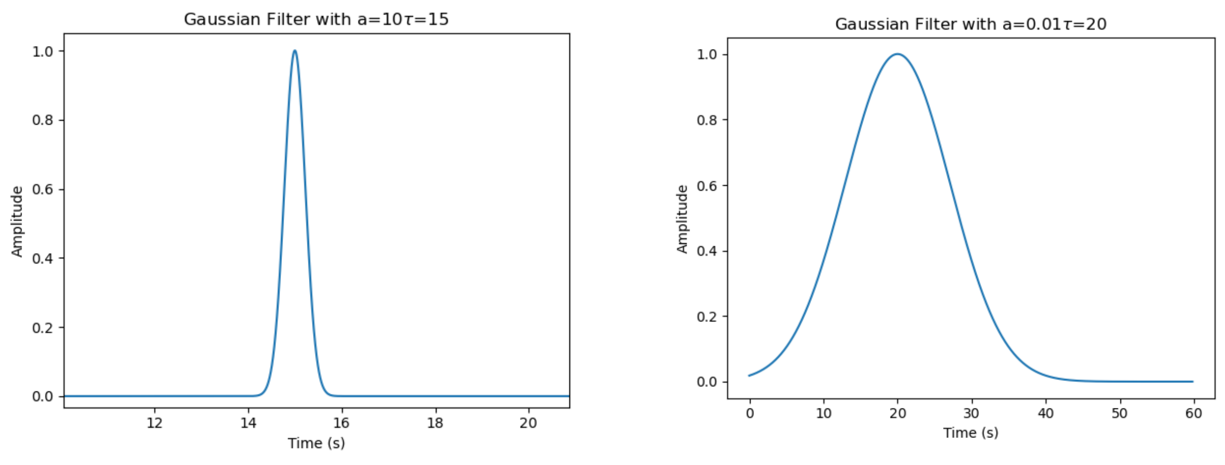

Figure 1. These two graphs show Gaussian filters with various values of the parameters a and τ. Note that a affects the width of the filter, and τ affects the center. Additionally, while the filter here is shown in the time domain, it can also be applied to the frequency domain.

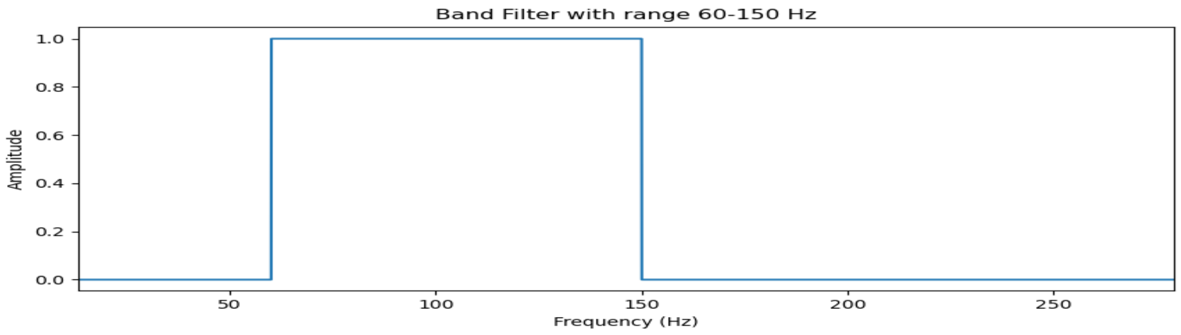

Figure 2. This shows a band pass filter with the lower limit of 60 Hz and an upper limit of 150 Hz. Here, all frequencies outside this range are completely attenuated, and those inside are unmodified.

3.5. Overtones

In the Pink Floyd sample, there were multiple instruments played. When an instrument plays a note, it does not sound only one frequency. Every instrument will have several overtones for each note. These overtones are multiples of the fundamental frequency. If a note played has the fundamental frequency f, the first overtone will have frequency $2f$, the second overtone will have frequency $3f$, and so on. Depending on the instrument, the overtones can have high enough amplitude to be mistaken for separate note. By filtering out overtones, we can hopefully isolate each instrument.

3.6. Band Pass Filter

In order to isolate the bass guitar in the Pink Floyd sample, we needed some way of ignoring all the other frequencies. Just as the Gaussian allows us to smoothly select some subset of our data, the band pass also lets us filter out unwanted frequencies. However, rather than gradually attenuating certain frequencies, the band pass filter will set the amplitude of any frequencies outside its range to 0, leaving the amplitude of those inside alone. A band pass filter can be visualized in Figure 2

4. Algorithm Implementation and Development

To start, since we are using Python, and no simple method of parsing m4a audio files exists, we use MATLAB instead to convert the audio file into a matrix and save the matrix. Then, we can load the files in Python.

First, we wanted to create the spectrogram and label the notes for the Guns N' Roses song. The code for this task is in the section labeled Part 1a in Appendix B. First, we calculate several values such as the total number of samples, the length of the song, the time values, and the wavenumbers. When scaling the wavenumbers, here we only scale by $1/(song\ length\ in\ seconds)$, as we want the frequency to be in Hz, and not the angular frequency, since we are comparing with musical notes. From the time values, we can construct a list of all the τ values we use. Then, we loop through each value of τ, constructing the Gaussian filter centered at that point, and then apply the filter to our data. Then, we take the Fourier transform of our data to get the frequencies present at the given time window around τ. Once we have done this for each τ, we have finished constructing our spectrogram. This code can be seen in the Helper Functions section, under the function `createSpectrogram()` in Appendix B.

Next, we rescale the spectrogram so that its maximum value is 1. Then, we discard most of the spectrogram outside of the frequencies 0 Hz to 1000 Hz. This is the range for which all the notes are visible and, given that displaying a large spectrogram is very intensive, this significantly helps performance. Once we have discarded the irrelevant frequencies, we can then graph the spectrogram.

Now, we have to label each of the notes. To do so, we find the location of the frequency with the maximal amplitude and translate this index into a frequency. Then, in order to ignore the periods of silence where no notes are played, we discard any slices where the maximal value is not above a certain threshold. In most cases, we found a value of .5 to be ideal. Then, we translate the list of frequencies into note names, and add them to our spectrogram plotting the names directly over where the note occurs. This process is detailed in the Helper Functions section, under the function `labelNotes()` in Appendix B.

We repeat this process for the Pink Floyd song. This code can be found in Part 1b of Appendix B. However, upon trying this, we found the performance to be extremely slow due the longer length of the audio clip. Thus, we decided it would be best to divide the clip into ten six-second clips and construct the spectrograms of each of those six-second clips, and then stitch all ten spectrograms together afterward. One of the main benefits is that we aren't wasting computational power on calculating the transform for a large amount of near-zero data. However, the one drawback is that the boundaries between clips won't have as accurate data. When our value of τ gets near to one of these boundaries (say at 6 seconds, 12 seconds, 18 seconds, etc.), any data immediately after the boundary is not included. Thus, the frequencies shown at these boundaries tend to be noisier. However, we found that this slight inaccuracy was worth the significant time improvement. The code for creating the partial spectrograms and combining them can be found in the Helper Functions section under the function `createSpectrogramSplit()` in Appendix B.

Applying this method, we split the data into 10 different segments, and then proceed as usual, creating the spectrogram for each individual segment and then stitching them together afterwards. We then clip the spectrogram to only include frequencies between 0 and 1000 Hz, and then can plot and label the notes on the spectrogram.

Next, we wanted to isolate the bass notes in the Pink Floyd sample. This is done in Part 2 of Appendix B. We found that the frequency of a bass guitar ranges from as low as 40 Hz to as high as 400 Hz. So, we first took the Fourier transform of the data so that we could apply a band pass filter to attenuate frequencies outside of this range. We then took the inverse Fourier transform to return to the time domain, where we then constructed the spectrogram. Once we did this, we were able to see that the notes had a smaller range, varying only from 60 Hz to 150 Hz, so we changed our band pass filter and repeated this. This gave us a clear list of all the bass notes and their frequencies, which will be used later.

Finally, we wanted to isolate the lead guitar solo in the Pink Floyd sample. This is done in the section labeled Part 3 in Appendix B. We started by now filtering out all frequencies below 150 Hz (and above 5000 Hz). There was still too much noise to reproduce the lead guitar part, so we wanted to filter out the overtones of the bass guitar. So, we took the list of bass notes, mapped these to the frequencies, and for each one calculated the overtones. At each time point, we created a Gaussian filter centered around each overtone, and then added all the Gaussian filters together. Since we want to filter out the overtones, we then took $1 - g(t)$, which can be visualized in Figure 3, to include all frequencies except the overtones. We then applied each of these filters to their corresponding time points.

Initially, we used the first five overtones, however this didn't help us isolate the lead guitar. We adjusted the filter so that the width of covered around 20 Hz, and still had no luck. We then considered the possibility that for some of the lower notes, 5 overtones might not be enough. We attempted to find out how the amplitude of overtones compared to the fundamental frequency, in the hopes that we could figure out how many overtones we would need to filter but found no existing research on this. Thus, we ended up trying even higher values, using 15 overtones, but found that this ended up nullifying lots of the actual notes we were trying to find. Thus, we decreased the strength of the filter for each successive overtone, by multiplying the individual filter by $1/(overtone\ number)$. Thus, the Gaussian filter for the 2nd overtone was multiplied by $1/2$, and so on. This yielded slightly better results, in that more notes were detected, however it was still difficult to discern the actual score. Finally, we tested out multiple values of exponents for the multiplier, such as multiplying by $n^{-1/3}$ instead of $1/n$, but settled on $1/n$. The final filter spectrogram, shown in Figure 4, was created from each of the bass notes. At this point, we decided to compare our results to the actual notes and were able to see some notes were properly discerned, specifically those that were sustained, played loudly, and not surrounded by ornaments (musical embellishments).

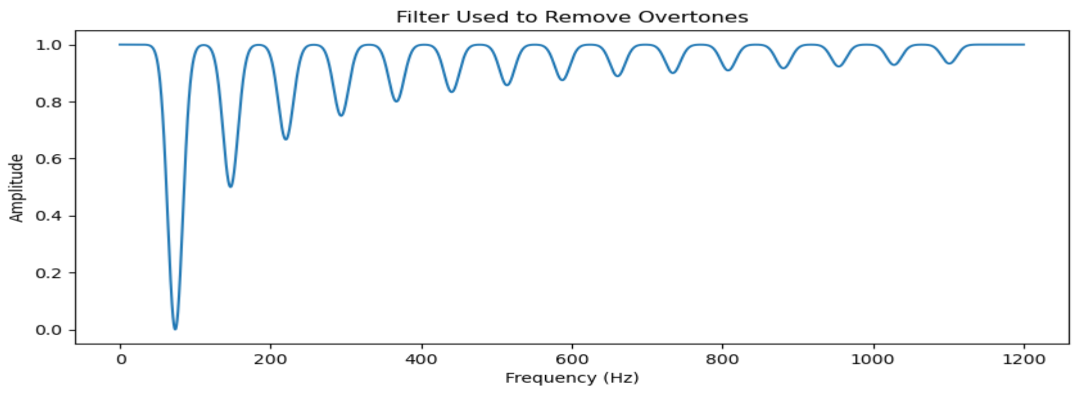

Figure 3.This shows the filter we used to remove overtones. This filter would look slightly different depending on the bass note it was constructed for. Note that all frequencies outside the overtones are left alone, and the frequencies near the overtones are attenuated.

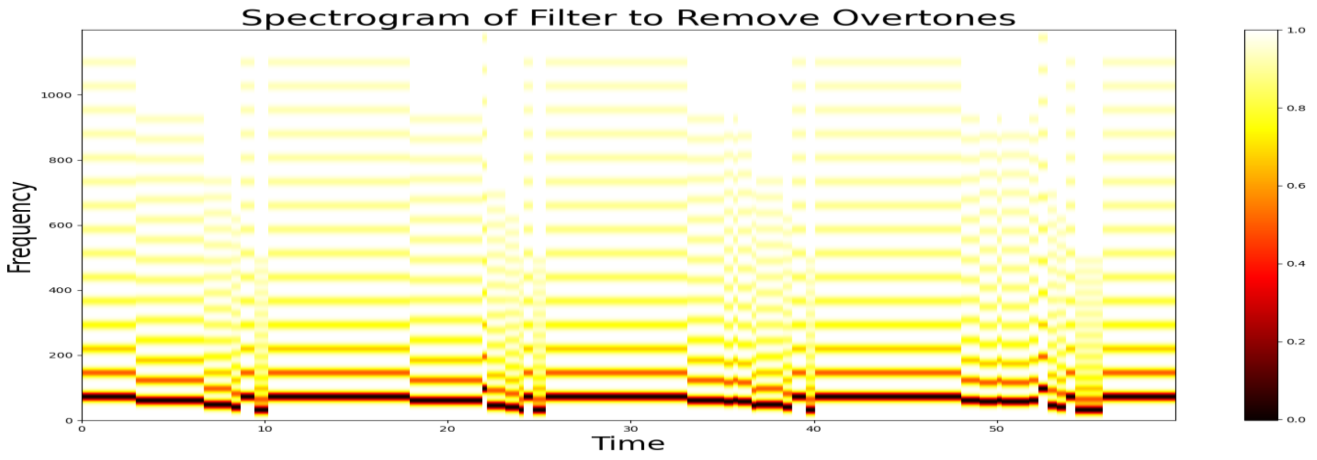

Figure 4. Here, we see the spectrogram of all the overtone filters for each point in time. Note that the overtones that get filter change as the bass note gets changed. Additionally, most of the spectrogram has a value very close to 1, showing that most of the frequencies are not modified. The spectrogram is also darker for lower frequencies, representing the usually larger amplitude of the lower overtones compared to the higher overtones.

5. Computational Results

Now, we will discuss the results achieved by applying the methods described in the previous section. First, the spectrogram of the Guns N' Roses sample is shown in Figure 5. The guitar notes are shown in the figure superimposed over the location of each note on the spectrogram. Written out, the notes for the Guns N' Roses sample are $C_4^\#, C_5^\#, G_4^\#, F_4^\#, F_5^\#, G_4^\#, F_5, G_4^\#$. This repeats once and is then followed up by $D_4^\#, C_5^\#, G_4^\#, F_4^\#, F_5^\#, G_4^\#, F_5, G_4^\#$, which also repeats. This is then followed by $F_4^\#, C_5^\#, G_4^\#, F_4^\#, F_5^\#, G_4^\#, F_5, G_4^\#$, which repeats. Finally, we return to the first phrase, which is $C_4^\#, C_5^\#, G_4^\#, F_4^\#, F_5^\#, G_4^\#, F_5, G_4^\#$. For the Pink Floyd sample, shown in Figure 6, the bass plays the following notes: $B_2, A_2, G_2, F_2^\#, E_2$. This phrase is repeated throughout the 60 second sample.

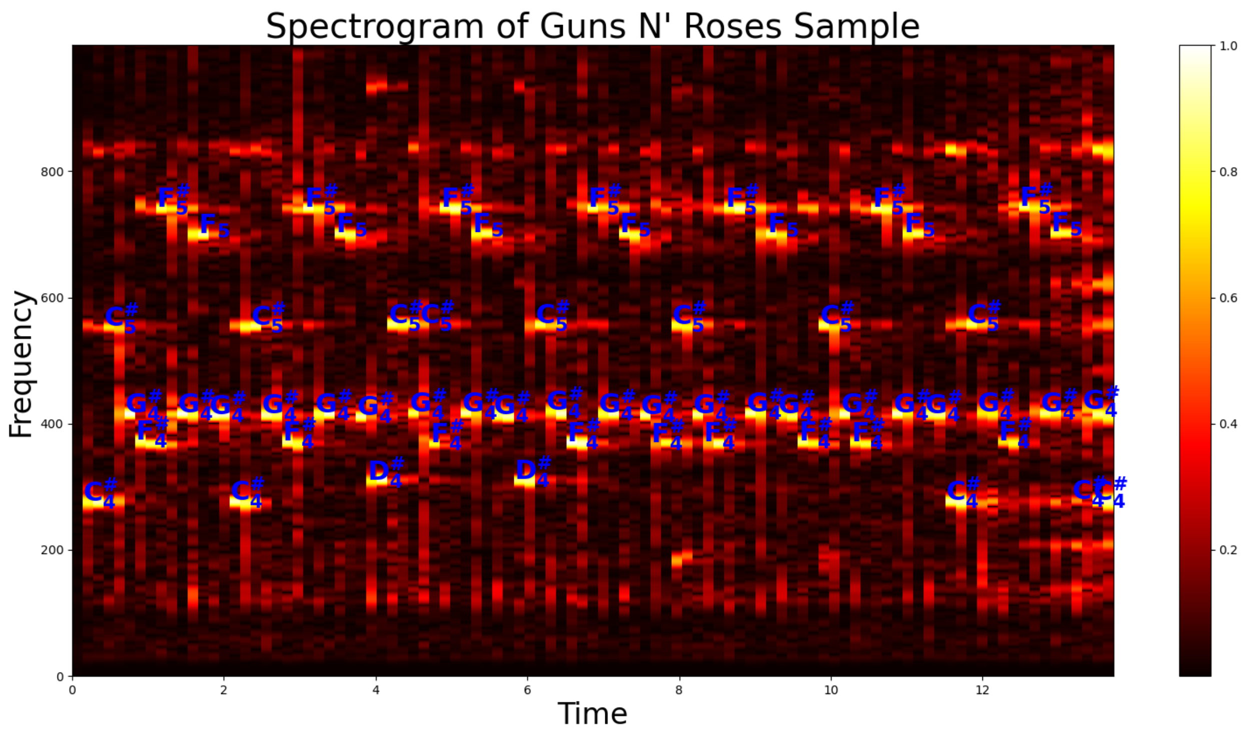

Figure 5. This is the spectrogram of the entire Guns N' Roses sample with the notes labeled, showing frequencies between 0 and 1000 Hz

Next, after applying the band pass filter, we got the spectrogram of only the bass notes, shown in Figure 7. The notes are much easier to decipher here since we are only seeing a small subset of the notes.

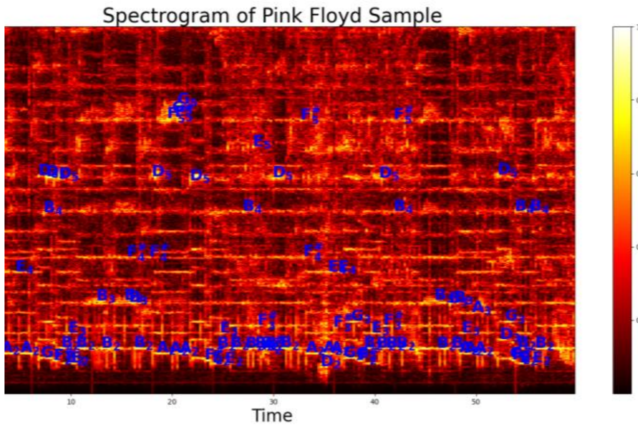

Figure 6. This figure shows the entire Pink Floyd sample's spectrogram. Note that the bass notes are the strongest in intensity, with only few of the guitar solo notes showing up.

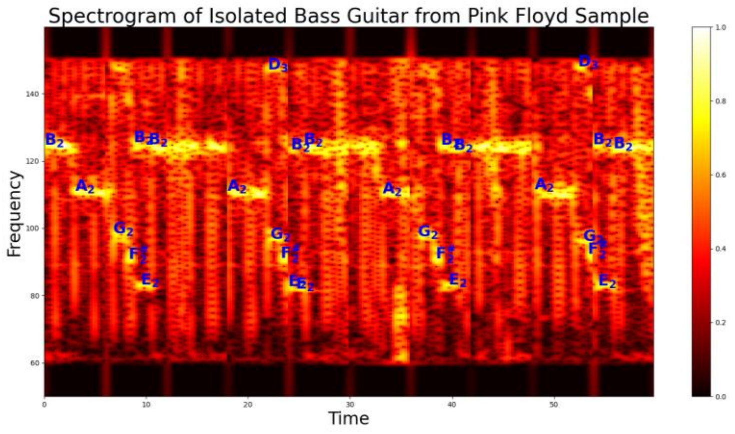

Figure 7. This is the spectrogram of the Pink Floyd sample with the bandpass filter applied from 60 Hz to 150 Hz. Here, it is much easier to see the notes that are played by the guitar.

Finally, we applied the overtone filter shown in Figure 4 and got the spectrogram shown in Figure 8. Compared to the spectrogram without the filter, there appear to be better results. We are able to see the first 3 notes of the solo, $F_4^\#, F_4^\#, E_4$. Additionally, looking at around 7 seconds, we see the sequence of notes $D_5, D_5, F_5^\#$, which is a simplified version of what is actually played. Further, at around 18 seconds, we see $D_5, F_5^\#, G_5$, which is similar to the actual notes, albeit simplified.

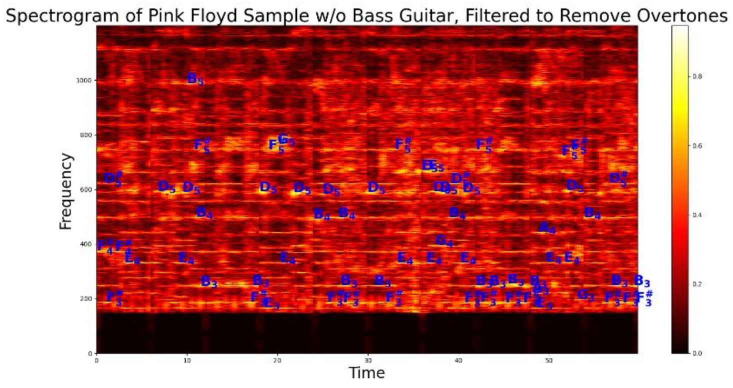

Figure 8. This shows the Pink Floyd song with the overtone filter applied. Note that more of the notes in the 400 – 800 Hz range are visible.

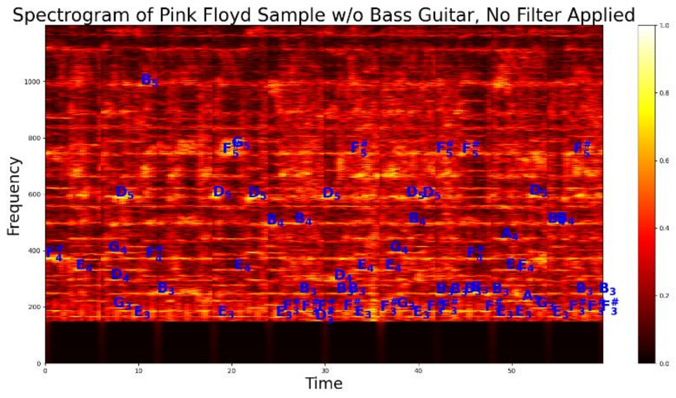

Figure 9. This figure shows the Pink Floyd sample without the overtone filter applied. Upper notes aren't found as easily.

6. Summary and Conclusions

Given two audio files, we were successfully able to isolate the notes of the melody from one, and the bass guitar from the other. We accomplished this using the Gabor transform in order to construct spectrograms. From the spectrograms, we were able to isolate the maximal frequencies at each point in time, and thus dictate the musical note being played at each point in time. However, we were not able to successfully isolate the melody from the second audio clip. This melody was a guitar solo that had very frequent note changes, pitch bends, and other instrumental techniques that lead to less isolated frequencies. Additionally, there was a high level of noise, such that it was difficult to discern peaks. Although overtone filtering did somewhat help, there is still more that could be done. Advanced denoising techniques could be applied which could make it easier to spot prevalent frequencies. We had to split the Pink Floyd sample into parts in order to create the spectrogram without large performance decreases. A more powerful computer could use narrower filters and many more windows, compared to the 100 or 250 windows we used. This would give more accuracy in the time domain. However, it should be noted that this still comes with the drawback of decreasing the frequency accuracy, as that is one of the mathematical limitations of the Fourier transform: the less time data you have, the more spread out the frequency data will be, and vice versa.

7. Appendix A: Functions Used

Python

NumPy

`np.abs` : Returns the absolute value. Used because we have complex data, but only care about the magnitude of each data point.

`np.amax` : Returns the largest value of a matrix. Used to normalize certain matrices.

`np.append` : Adds to matrices or vectors together.

`np.arange` : Generates a vector with consecutive integer values. We used this to generate the rescaled frequencies.

`np.argmax` : Gets the index of the maximal element

`np.exp` : Calculates the exponential. Used to generate the Gaussian filter.

`np.fft.fft` : Calculates the fast Fourier transform. Used to transform the data to the frequency space for constructing the spectrogram

`np.fft.fftshift` : Shifts the zero-frequency to the center of the vector. Used because of how the fast Fourier transform and inverse fast Fourier transform structure their return values.

`np.fft.ifft` : Calculates the n-dimensional inverse fast Fourier transform. Used to convert data from the frequency domain back to its original domain.

`np.fft.ifftshift` : Shifts the zero-frequency back when doing the inverse transform. Used again because of how the return values are structured after the inverse fast Fourier transform.

`np.linspace` : Creates a vector of evenly spaced numbers. Used to generate the τ values

MatPlotLib

`pyplot.figure` : Generates a figure to display graphs on.

`pyplot.plot` : Creates a line graph of the data. We used this to generate the trajectories

`pyplot.annotate` : Adds text to a graph. Used to add the note names to the plot.

SciPy

`io.loadmat` : Used to load a MATLAB matrix into Python

Matlab

`audioread` : Used to read in the audio files into a vector

`save` : Used to save the song vector and sample rate into a single *.mat file, to be later imported in Python

8. Appendix B: Code

MATLAB:

```
[y, Fs] = audioread('GNR.m4a');
save('GNR.mat', 'y', 'Fs')
[y, Fs] = audioread('Floyd.m4a');
save('Floyd.mat', 'y', 'Fs')
```

Python:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as sp
from numpy.fft import fft, fftshift, ifft, ifftshift

## Helper Functions

# Converts a given frequency into the corresponding note name
def freq2note(freq):
    if freq < 25 or freq > 4190:
        return ""
    key_num = int(round(12 * np.log2(freq / 440) + 49))
    octave = int(np.floor((key_num + 8) / 12))
    note_index = key_num % 12 - 1
    if note_index == -1:
        note_index = 11
    notes = ['A', 'A#', 'B', 'C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#']
    return notes[note_index] + str(octave)

# Converts a given note name into the frequency
def note2freq(note):
    if len(note) == 3:
        note_name = note[0:2]
        octave = int(note[2])
    elif len(note) == 2:
        note_name = note[0]
        octave = int(note[1])
    else:
        return 0
    notes = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B']

    note_index = notes.index(note_name)
    key_num = note_index + 4 + (12 * (octave - 1))
    return np.power(2, (key_num - 49) / 12) * 440

# Convert a frequency into the corresponding index in order to set limits
# on the y-axis of the spectrogram
def getFreqIndex(ks, freq):
    index = ks.searchsorted(freq)
```

```python
    index = np.clip(index, 1, len(ks) - 1)
    left = ks[index - 1]
    right = ks[index]
    index -= freq - left < right - freq
    return index

# Given a spectrogram and threshold value, returns the list of note names that have a amplitu
de
# larger than the threshold, inserting empty strings where no note is found
def getNotes(spectrogram, ks, threshold):
    max_freq_ind = np.argmax(spectrogram, axis=0)
    max_freq = np.max(spectrogram, axis=0)
    max_freq_ind[max_freq < threshold] = 0
    note_frequencies = ks[max_freq_ind]
    notes = np.array([freq2note(frequency) for frequency in note_frequencies])
    return notes, note_frequencies

# Discards parts of the spectrogram outside of the given min and max frequencies
def clipSpectrogram(spectrogram, ks, min_freq, max_freq):
    min_freq_ind = getFreqIndex(ks, min_freq)
    max_freq_ind = getFreqIndex(ks, max_freq)

    spectrogram = spectrogram[min_freq_ind:max_freq_ind, :]
    ks = ks[min_freq_ind:max_freq_ind]

    return spectrogram, ks

# Takes the song matrices imported from matlab and returns the song vector and sample rate.
def getSongMatData(song):
    y = song['y'].flatten()
    Fs = song['Fs'].flatten()[0]
    return y, Fs


# Takes song vector and sample rate and returns the total number of samples,
# length in seconds, time vector, and shifted frequency vector
def getSongData(y, Fs):
    n = len(y)
    song_length = n / Fs
    t = np.arange(0, n) / Fs
    k = np.append(np.arange(0, n/2), np.arange(-n/2, 0)) / song_length
    ks = fftshift(k)
    return n, song_length, t, ks

# Creates a spectrogram by first splitting it into parts and then stitching those partial
# spectrograms together
def createSpectrogramSplit(y, Fs, num_parts, num_windows, filter_width, log_transform, debug_
plot = False):
    y_parts = np.split(y, num_parts)
    spectrogram = np.zeros((len(y_parts[0]), 0))
    all_taus = np.array([])
    for i, yi in enumerate(y_parts):
        print("Part: " + str(i + 1) + " / " + str(num_parts))

        _, song_length, _, _ = getSongData(yi, Fs)

        spectrogram_partial, taus, ks = createSpectrogram(yi, Fs, num_windows, filter_width,
log_transform, debug_plot)
        all_taus = np.hstack((all_taus, taus + song_length * i))

        spectrogram = np.hstack((spectrogram, spectrogram_partial))

    return spectrogram, all_taus, ks

# Creates a spectrogram of a given song vector
def createSpectrogram(y, Fs, num_windows, filter_width, log_transform, debug_plot = False):
    n, _, t, ks = getSongData(y, Fs)

    taus = np.linspace(0, t[-1], num_windows)
    spectrogram = np.zeros((n, num_windows))

    for i, tau in enumerate(taus):
        print("Progress: " + str(i + 1) + " / " + str(num_windows))
        gauss = np.exp(-filter_width * (t - tau)**2)
        window = y * gauss
        y_transform = fftshift(fft(window))

        if log_transform:
            spectrogram[:, i] = np.log(np.abs(y_transform) + 1)
        else:
            spectrogram[:, i] = np.abs(y_transform)

        if debug_plot and i == int(num_windows / 4):
            plt.plot(t, gauss)
            plt.show()

            plt.plot(t, y * gauss)
            plt.show()

            plt.plot(ks, y_transform)
            plt.show()

    return spectrogram, taus, ks

# Plots a spectrogram
def plotSpectrogram(spectrogram, taus, ks, title=""):
    plt.pcolormesh(taus, ks, spectrogram, shading='gourad', cmap='hot')
    plt.colorbar()
    plt.title(title, size=28)
    plt.xlabel("Time", size=24)
    plt.ylabel("Frequency", size=24)

# Adds text annotations of note names to a spectrogram
def labelNotes(notes, taus, note_frequencies):
    prev_note = ""
    for i, note in enumerate(notes):
        if note != prev_note:
            note_txt = note
            if len(note) == 3:
                note_txt = "$\mathregular{" + note[0] + "^\\" + note[1] + "_" + note[2] + "}$
"
            elif len(note) == 2:
                note_txt = "$\mathregular{" + note[0] + "_" + note[1] + "}$"
            plt.annotate(note_txt, (taus[i], note_frequencies[i]), color='b', weight='bold',
fontsize=22)
        prev_note = note

# Plots a spectrogram with note names added
def plotSpectrogramWithNotes(spectrogram, taus, ks, threshold, title=""):
    plotSpectrogram(spectrogram, taus, ks, title)
    notes, note_frequencies = getNotes(spectrogram, ks, threshold)
    labelNotes(notes, taus, note_frequencies)
    plt.show()

# Load data from MATLAB matrices
gnr = sp.loadmat('GNR.mat')
floyd = sp.loadmat('Floyd.mat')

## Part 1a
# Create Spectrogram and Label Notes for GNR
y, Fs = getSongMatData(gnr)

spectrogram, taus, ks = createSpectrogram(y, Fs, num_windows=100, filter_width=500, log_trans
form=True)

# Normalize spectrogram
spectrogram = spectrogram / np.amax(spectrogram)

# Throw away values outside normal music range for performance reasons
spectrogram, ks = clipSpectrogram(spectrogram, ks, min_freq=0, max_freq=1000)

plotSpectrogramWithNotes(spectrogram, taus, ks, threshold=.3, title="Spectrogram of Guns N' R
oses Sample")


## Part 1b
# Create Spectrogram and Label Notes for Floyd
y, Fs = getSongMatData(floyd)
y = y[0:-1]
```

```
spectrogram, taus, ks = createSpectrogramSplit(y, Fs, num_parts=10, num_windows=25, filter_wi
dth = 50, log_transform=True)

# Normalize spectrogram, crop to desired area
spectrogram = spectrogram / np.amax(spectrogram)
spectrogram, ks = clipSpectrogram(spectrogram, ks, min_freq=0, max_freq=1000)

plotSpectrogramWithNotes(spectrogram, taus, ks, threshold=.5, title="Spectrogram of Pink Floy
d Sample")


## Part 2
y, Fs = getSongMatData(floyd)
y = y[0:-1]

n, _, _, ks = getSongData(y, Fs)

# Take Fourier Transform and apply a band pass filter to isolate the bass
y_transform = fftshift(fft(y))
lf = 60
hf = 150
band_filter = np.zeros(n)
band_filter[getFreqIndex(ks, lf):getFreqIndex(ks, hf)] = 1

y_transform_filtered = y_transform * band_filter
y_filtered = ifft(ifftshift(y_transform_filtered))

# Now create the spectrogram as above, except using the filtered version
spectrogram_bass, taus, ks = createSpectrogramSplit(y_filtered, Fs, num_parts=10, num_windows
=25, filter_width=5, log_transform=True)

# Normalize spectrogram, crop to desired area
spectrogram_bass = spectrogram_bass / np.amax(spectrogram_bass)
spectrogram_bass, ks = clipSpectrogram(spectrogram_bass, ks, min_freq=50, max_freq=160)

plotSpectrogramWithNotes(spectrogram_bass, taus, ks, threshold=.5, title="Spectrogram of Isol
ated Bass Guitar from Pink Floyd Sample")

## Part 3
y, Fs = getSongMatData(floyd)
y = y[0:-1]
n, _, _, ks = getSongData(y, Fs)

y_transform = fftshift(fft(y))
lf = 150
hf = 20000
band_filter = np.zeros(n)
band_filter[getFreqIndex(ks, lf):getFreqIndex(ks, hf)] = 1

y_transform_filtered = y_transform * band_filter
```

```
y_filtered = ifft(ifftshift(y_transform_filtered))

# Now create the spectrogram as above, except using the filtered version
spectrogram, taus, ks = createSpectrogramSplit(y_filtered, Fs, num_parts=10, num_windows=25,
filter_width=5, log_transform=True)

# Normalize spectrogram, crop to desired area
spectrogram = spectrogram / np.amax(spectrogram)
spectrogram, ks = clipSpectrogram(spectrogram, ks, min_freq=0, max_freq=1200)

# Get the bass note names
bass_notes, _ = getNotes(spectrogram_bass, ks, threshold=.5)

overtone_filters = np.zeros(spectrogram.shape)

for i, note in enumerate(bass_notes):
    # Find the frequency of each note to construct the overtones from
    freq = note2freq(note)

    num_overtones = 15
    gauss_filters = np.zeros((spectrogram.shape[0], num_overtones))
    # Construct a Gaussian filter for each overtone
    for overtone_num in range(num_overtones):
        filter_width = .005
        center_frequency = freq * (overtone_num + 1)
        gauss_filters[:, overtone_num] = np.exp(-
filter_width* (ks - center_frequency)**2) / (overtone_num + 1)
    # Add up all the Gaussian filters and invert it
    combined_filter = 1 - np.sum(gauss_filters, axis=1)

    # Add the filter for each time point
    overtone_filters[:, i] = combined_filter

# Apply the filter to the spectrogram
spectrogram_no_overtones = spectrogram * overtone_filters

plotSpectrogram(overtone_filters, taus, ks, title="Spectrogram of Filter to Remove Overtones"
)
plt.show()

plotSpectrogramWithNotes(spectrogram_no_overtones, taus, ks, threshold=.25, title="Spectrogra
m of Pink Floyd Sample w/o Bass Guitar, Filtered to Remove Overtones")

plotSpectrogramWithNotes(spectrogram, taus, ks, threshold=.5, title="Spectrogram of Pink Floy
d Sample w/o Bass Guitar, No Filter Applied")
```