Floyd to Fourier: Analyzing Music with Gabor Transforms

Alex Mallen

Abstract

In this paper I will be showing how the short-time Fourier transform, otherwise known as the Gabor transform, can be used to reproduce the musical score of a song. Unlike the traditional Fourier transform which only relays how much of each frequency is present in a signal, the Gabor transform also relays information about when a certain frequency was present in the signal. This is crucial to reconstructing a song's melody.

1 Introduction

The overall goal of this project was to reproduce the musical score for specific instruments in Guns n' Roses's *Sweet Child O' Mine* and Pink Floyd's *Comfortably Numb* using only the audio of each song.

Describing a song with a musical score can be thought of as a form of audio compression. It relays to the reader what notes, or frequencies, are present in the song at every beat rather than indicating the exact amplitude of the audio at very frequent intervals. As is common in compression, some information is lost such as the timbre of the notes being played.

This process of compression can also be described in the language of Fourier analysis. First, a spectrogram of the music is computed to tell us which frequencies are in the song at which times. This process is nearly lossless as it can be inverted to obtain approximately the initial audio. Going from the spectrogram to the musical score is the lossy step as it involves removing all but the most dominant frequencies at every point in time. By converting these frequencies into their corresponding musical notes, we can construct a piece of audio's musical score.

2 Theory

In the previous paper, I introduced the idea of the Fourier transform, which can take any data and transform it into a sum of sinudoidal functions with different frequencies [2]. One important property of the Fourier transform was that it caries very little information about the location within the original signal of the frequencies. This was useful for denoising

signals from a moving source. However, the timing of each note is important information for the construction of a musical score.

A simple adjustment can be made to the FFT algorithm to allow it to communicate when the frequency appeared in the song. We can take our original audio $\mathbf{x}$ of length T and partition it into a list of m clips $X = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_{m-1}]$. Then, we can compute the FFT of each $\mathbf{x}_i$ separately to obtain $\hat{X} = [\hat{\mathbf{x}}_0, \hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_i, \ldots, \hat{\mathbf{x}}_{m-1}]$. Let $\Delta t = T/m$ be the time interval spanned by each clip. With this framework, we are certain that all of the frequencies in each $\hat{\mathbf{x}}_i$ occurred within that clip.

$$i\Delta t < t < (i+1)\Delta t \tag{1}$$

This method is called the short-time Fourier transform, otherwise known as the Gabor transform [3].

$\hat{X}$ is known as the spectrogram of $\mathbf{x}$ and can be displayed as an image that shows the frequency spectrum of the song vertically at every point in time. The precision with which we can determine this time t depends on Δt according to equation 1. The most time precision occurs with small Δt. However, if the Δt is smaller than the period of the lowest note, then that note will be lost. More generally, the smaller Δt, the fewer data points there are off of which to base your frequency analysis, and the worse your frequency resolution. This is known as the Heisenberg uncertainty principle. This fundamental tradeoff in time-frequency analysis means that it is critical to pick the correct value of Δt.

Once the frequencies at each subinterval of time have been discovered, notes should appear as dominant modes in the spectrogram.

3 Implementation

I wrote all of my code in Python, and made use of a Jupyter Notebook to rapidly test variations on the many parameters that affect the quality of the output. All source code can be found in [1].

3.1 GNR Guitar and Pink Floyd Bass Extraction

With the music loaded as a numpy array of amplitudes, I first computed the Gabor transform of each song to produce a spectrogram as described in section 2. Details for my implementation of `gabor` can be found in Appendix A. With equation 1 in mind, I chose the time window length that created the most distinct modes in the frequency domain so I could accurately represent the notes while still understanding the relative order in which they appear. I then computed the frequency associated with each index in the frequency axis of my spectrogram with $1/L * \texttt{np.arange}(-n//2, n//2)$ where L is the length, in seconds, of each Gabor window, and n is the number of elements in each Gabor window. The prefactor of $1/L$ is present because the lowest magnitude nonzero frequency possible in

an L-periodic function is $1/L$. There is no factor of 2π because we are dealing with cycles per second, not radians per second, in order to make converting to musical notes easier.

My next step was to clean up the spectrogram by taking the sum of the absolute value of the Fourier coefficients associated with each frequency and its negative. I then applied an edge-detecting convolution to the spectrogram to accentuate the dominant Fourier coefficients. I used the `scipy.ndimage.convolve` implementation of convolution with edge wrapping, which is described in Appendix A. There were many iterations of kernels I went through to produce a spectrogram where the correct frequencies have the largest coefficients with minimal noise. My original idea was to apply a kernel that looked similar to the signal I wanted to accentuate, with two rows of positive values in the middle and two rows of negative values on the edges, but I actually found that a standard horizontal edge detection kernel performed better, especially when preceded by a small Gaussian kernel.

The final step was to extract the notes from the spectrogram. This involved taking the argmax Fourier mode of every window and converting the frequency in Hz into its corresponding musical note. I wrote a short function that maps every frequency in a valid range to the musical note with the nearest frequency. While the argmax value did not always correspond to the correct note played at that time, the correct note was usually identifiable from context. However, in order to be able to filter out the bass in *Comfortably Numb* I produced a clean score by using context to manually denoise the few outlier points. It should be noted that the bass in *Comfortably Numb* sometimes played two notes at once, so these were manually introduced according to their clear presence in the spectrogram. To

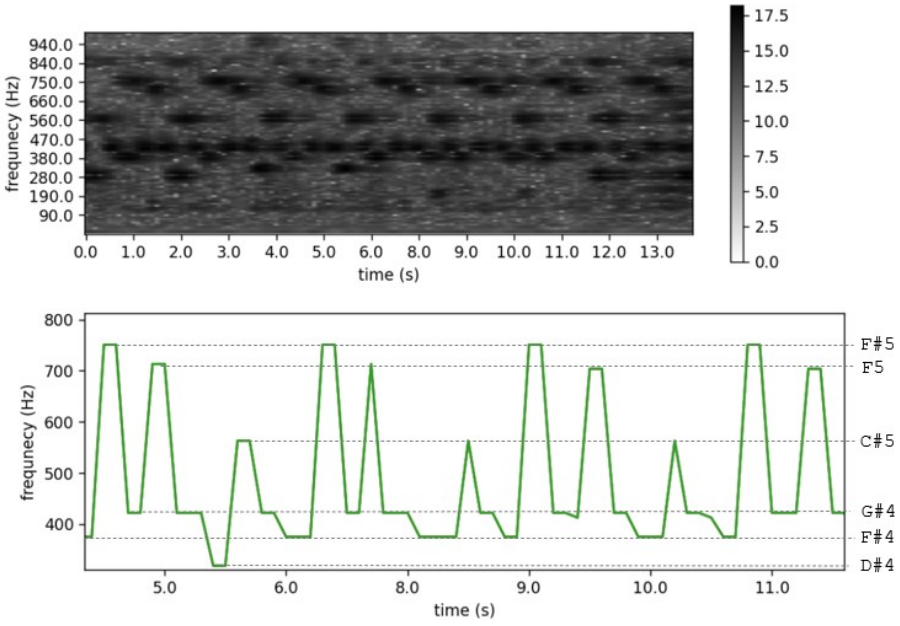

FIGURE 1: TOP: This is the spectrogram of *Sweet Child O' Mine* after convolving it with a Gaussian then edge detecting kernel. BOTTOM: This depicts which notes were being played at each point in time from around 4 to 12 seconds. The rest of the song looks very similar.

filter out the bass I iterated over all my time windows and applied a Gaussian filter around all of the cleaned dominant frequencies.

3.2 Pink Floyd Guitar Extraction

I attempted to extract the notes of the guitar solo by subtracting out the more dominant bass notes and their overtones. This code iterated over each Gabor window and subtracted off β^j times each harmonic j of the bass note(s) played at that time, where β is a decay parameter slightly less than 1. I constructed a Gaussian filter around each of those frequencies and subtracted off the filter applied to the Fourier coefficients.

With a bass-removed spectrogram, I first attempted to extract notes using an edge-detecting kernel and an argmax unsuccessfully, so I instead looked at the spectrogram while listening to the song for prominent guitar notes, and read the frequency from the spectrogram in order to find the which note was being played.

4 Results

Figure 1 depicts the guitar from *Sweet Child O' Mine* as a spectrogram and shows extracted notes. The following is the sequence of notes I recovered from the song. Each row is almost identical apart for two notes, which have been bolded.

C#4 C#5 G#4 F#4 F#5 G#4 F5 G#4 **C#4** C#5 G#4 F#4 F#5 G#4 F5 G#4
D#4 C#5 G#4 F#4 F#5 G#4 F5 G#4 **D#4** C#5 G#4 F#4 F#5 G#4 F5 G#4
F#4 C#5 G#4 F#4 F#5 G#4 F5 G#4 **F#4** C#5 G#4 F#4 F#5 G#4 F5 G#4
C#4 C#5 G#4 F#4 F#5 G#4 F5 G#4 **C#4** C#5

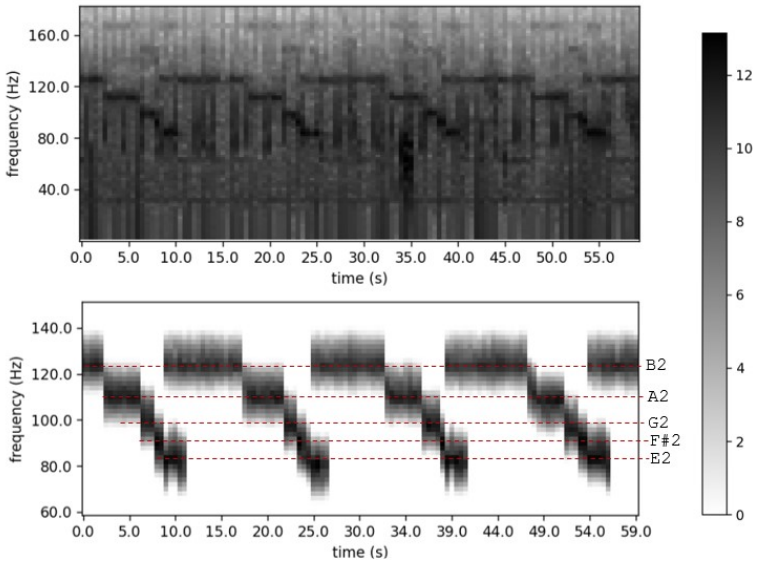

FIGURE 2: TOP: This is the spectrogram of *Comfortably Numb* centered around the bass frequencies. BOTTOM: This is the spectrogram filtered around the most dominant notes.

Extracting the bass from Pink Floyd involved a longer window than for Guns n' Roses because the low notes of the bass have a longer period. Unfortunately, even after increasing the window length to two seconds, which is already longer than the shortest bass note, the lowest note B1≈62 Hz did not appear in the spectrogram. I expect this is because B2≈123 Hz is played at the same time, and since it is a harmonic of B1, it overshadowed its presence. Figure 2 provides a complete description of the results of performing short-time Fourier analysis to extract the Pink Floyd bass.

The guitar solo of *Comfortably Numb* proved the most challenging to analyze for both its complexity and subtlety in the spectrogram. Subtracting off multiples of the bass notes did not successfully remove overtones because there was a small margin of error in the discovered frequencies that was amplified in the process. The plot of notes obtained by taking the argmax of the spectrogram is omitted because it is too noisy; instead, see figure 3.

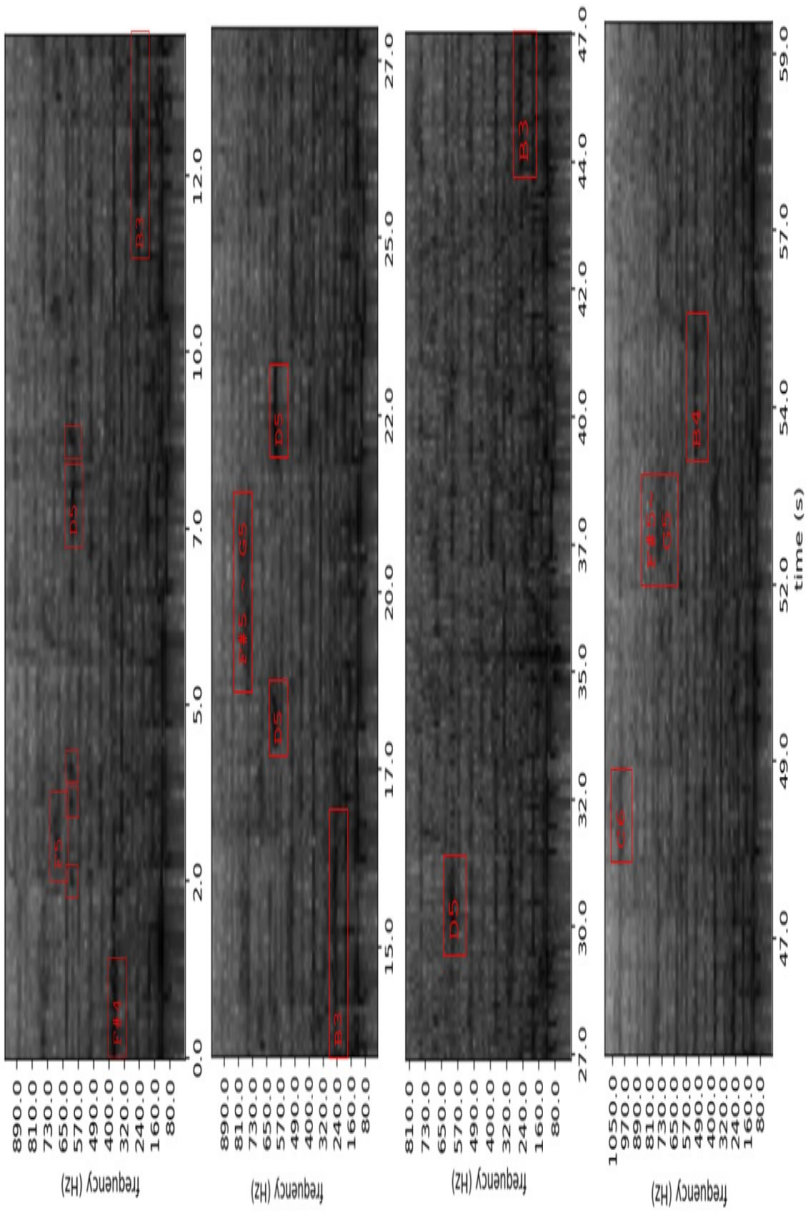

FIGURE 3: This is the bass-filtered spectrogram of *Comfortably Numb* with several guitar notes highlighted. The song is divided into 4 time intervals so that it fits on a page. Note that at 20 and 53 seconds the frequency is oscillating between an F#5 and a G5. The long-lasting notes were the easiest to identify, and few short notes were identified.

5 Conclusion

Both areas of failure come back to the Heisenberg uncertainty principle: I wasn't able to resolve the low notes of the bass because they didn't last long enough, and the short notes of the guitar were likewise too short-lived to see in the spectrogram. Overtones, different instruments, and general noise all contributed to the difficulty of this task. The under-performance of Fourier analysis makes me wonder how humans are able to make such clear distinctions in the pattern of the notes where Fourier fails. Perhaps the wavelet framework would outperform this because it was designed with the Heisenberg uncertainty principle in mind.

Nevertheless, short-time Fourier analysis proved to be a useful tool to recognize dominant frequencies in data. While I already have a difficult time parsing which note is being played at which time in the music, signals which are not explicitly musical would be even more challenging to analyze without the Fourier transform.

Appendix A

Important functions and their implementations

`numpy.fft.fft`

Takes a 1-dimensional array of complex data as input and returns a complex array of the same length representing how much of each frequency is present in the data. Assumes that the interval spanned the data is 2π and that the data is periodic in 2π. See [4].

`gabor`

Otherwise know as the short-time Fourier transform. Takes music: 1D numpy array representing the music, rate: sampling rate in samples per second used to generate this sound, window_time: time interval in seconds over which to compute the short-time Fourier transform, and scale: standard deviation of the Gaussian filter to remove overtones. `gabor` returns a 2D numpy array of the Fourier transform for each window along the song. The width times height of this array will always be equal to the length of the input music (barring truncation) due to the Heisenberg uncertainty principle. See the docstring of this function in my Python code in Appendix B.

`gabor` calls `numpy.fft.fft` on each Gabor window, multiplies it by a Gaussian filter centered at 0, and sets one column of the spectrogram matrix to the resulting value.

`scipy.ndimage.convolve`

Takes an image and a kernel as input and returns the image convolved with the given kernel.

The result will have the same size as the original image, and the way it handles edge pixels is determined by the mode parameter. Taking the convolution involves setting every pixel to some linear combination of its neighbors according to the weights given by the kernel. See [5].

Appendix B

Python code

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
from numpy.fft import fft, ifft, fftshift, ifftshift, fftfreq
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'notebook')
from scipy.io import wavfile
import IPython.display as ipd
from scipy.ndimage import convolve


# In[2]:


rate, gnr_song = wavfile.read("GNR.wav") # rate in Hz
gnr_song = gnr_song.astype(np.float32)[:, 0]

_, floyd_song = wavfile.read("Floyd.wav") # rates of the two files are the
    same
floyd_song = floyd_song.astype(np.float32)[:, 0]


# In[3]:


ipd.Audio(gnr_song, rate=rate)


# In[4]:


ipd.Audio(floyd_song, rate=rate)


# # Question 1
# https://en.wikipedia.org/wiki/Overtone: To get rid of overtones filter
    around 0.

# In[5]:
```

```
def gabor(song, rate, window_time=0.01, scale=np.inf):
    """
    Otherwise know as the short-time Fourier transform

    music: 1D numpy array representing the music
    rate: sampling rate, in samples per second, used to generate this sound
    window_time: time in seconds over which to compute the short-time
        fourier transform
    scale: standard deviation of the gaussian filter to remove overtones.
        defaults to infinite (ie. no filtering)

    returns: 2D numpy array of the Fourier transform for each window along
        the song
            axis 0 represent frequencies and axis 1 represent time (window
                number)
            note: the frequency axis has been fftshifted, so it will be in
                the original order
    """
    window_width = int(window_time * rate)
    num_windows = len(song) // window_width

    if np.isfinite(scale):
        k = fftfreq(window_width)
        kernel = np.exp(-k**2 / (2 * scale**2))

    spectrogram = np.zeros((window_width, num_windows), dtype=np.complex128)

    for i in range(num_windows):
        window = song[i * window_width:(i + 1) * window_width]
        fft_window = fft(window)
        if np.isfinite(scale):
            fft_window *= kernel
        spectrogram[:, i] = fft_window

    return fftshift(spectrogram, axes=[0])


# In[6]:


def freq_to_note(freq):
    """
    Converts a frequency into a note. If the frequency is invalid it
        returns None
```

```
    freq: frequency in Hz of the note

    returns: str of the form "<octave>:<note>"
    """
    freqs = [65, 69, 73, 78, 82, 87, 92, 98, 104, 110, 117, 123, 131, 139,
        147, 156, 165, 175, 185, 196, 208, 220, 233, 247, 262, 277, 294,
        311, 330, 349, 370, 392, 415, 440, 466, 494, 523, 554, 587, 622,
        659, 698, 740, 784, 831, 880, 932, 988]
    notes = list()
    for i in range(2, 6):
        notes.extend([f"{i}:C", f"{i}:C#", f"{i}:D", f"{i}:D#", f"{i}:E",
            f"{i}:F", f"{i}:F#", f"{i}:G", f"{i}:G#", f"{i}:A", f"{i}:A#",
            f"{i}:B"])

    if freq < min(freqs) - 5 or freq > max(freqs) + 100:
        return None

    idx = np.argmin(abs(freq - freqs))
    return notes[idx]


# # GNR guitar

# In[181]:


L = 13 / 122
gnr_spectrogram = gabor(gnr_song, rate, window_time=L, scale=0.018)
n = gnr_spectrogram.shape[0]
k_gnr = 1 / L * np.arange(-n // 2, n // 2) # the smallest magnitude
    frequency for an L-periodic function should be 1/L


# In[213]:


plt.figure()
fl_gnr = abs(gnr_spectrogram[:gnr_spectrogram.shape[0] // 2 + 1]) +
    abs(gnr_spectrogram[gnr_spectrogram.shape[0] // 2:][::-1])

kernel = np.array([[1, 2, 1],
                   [2, 8, 2],
                   [1, 2, 1]])
conv_gnr = convolve(fl_gnr, kernel, mode="wrap")
kernel = np.array([[-2,-8,-2],
                   [ 2, 8, 2]])
conv_gnr = convolve(conv_gnr, kernel, mode="wrap")
```

```
plt.subplot(2,1,1)
plt.imshow(np.log(abs(conv_gnr) + 1), aspect=0.4, cmap="binary")
plt.xlabel("time (s)")
plt.ylabel("frequnecy (Hz)")
plt.xticks(ticks=range(0, fl_gnr.shape[1], 10), labels=np.round(13 / 120 *
    np.arange(0, fl_gnr.shape[1], 10)))
plt.yticks(ticks=range(fl_gnr.shape[0] - 400, fl_gnr.shape[0], 10),
    labels=np.round(-k_gnr[range(fl_gnr.shape[0] - 400, fl_gnr.shape[0],
    10)], -1))
plt.ylim([fl_gnr.shape[0] - 0, fl_gnr.shape[0] - 500])
plt.colorbar()


# In[312]:


gnr_freqs = -k_gnr[np.argmax(conv_gnr, axis=0)]


# In[216]:


plt.subplot(2,1,2)
# plt.scatter(np.arange(len(gnr_freqs)), gnr_freqs)
plt.plot(gnr_freqs)
plt.xlabel("time (s)")
plt.xticks(ticks=range(0, gnr_freqs.shape[0], 10), labels=np.round(13 /
    120 * np.arange(0, gnr_freqs.shape[0], 10)))
plt.ylabel("frequnecy (Hz)")
plt.show()


# In[ ]:




# In[217]:


print(list(freq_to_note(freq) for freq in gnr_freqs[::2]))


# # Floyd bass
```

```
# In[250]:


L = 2
floyd_spectrogram = gabor(floyd_song, rate, window_time=L, scale=0.001)
n = floyd_spectrogram.shape[0]
k_floyd = 1 / L * np.arange(-n // 2, n // 2) # the smallest magnitude
    frequency for an L-periodic function should be 1/L


# In[251]:


plt.figure()
fl_floyd_spect = abs(floyd_spectrogram[:floyd_spectrogram.shape[0] // 2])
    + abs(floyd_spectrogram[floyd_spectrogram.shape[0] // 2:][::-1])
filt_window = np.arange(fl_floyd_spect.shape[0])
cleaned_floyd = fl_floyd_spect

kernel = np.array([[-4,-8,-10,-8,-4],
                   [ 4, 8, 10, 8, 4]])
convolved_floyd = convolve(cleaned_floyd, kernel, mode="wrap")

plt.subplot(2,1,1)
plt.imshow(np.log(abs(cleaned_floyd) + 1), aspect="auto", cmap="binary")
plt.xlabel("time (s)")
plt.yticks(ticks=range(cleaned_floyd.shape[0] - 400,
    cleaned_floyd.shape[0], 20),
    labels=np.round(-k_floyd[range(cleaned_floyd.shape[0] - 400,
    cleaned_floyd.shape[0], 20)], -1))
plt.xticks(ticks=range(0, cleaned_floyd.shape[1], 10), labels=np.round(L *
    np.arange(0, cleaned_floyd.shape[1], 10)))
plt.ylabel("frequency (Hz)")
plt.ylim([cleaned_floyd.shape[0] - 0, cleaned_floyd.shape[0] - 500])
plt.colorbar()


# In[159]:


cleaned_floyd_freqs = list([freq] for freq in floyd_freqs)
print(list((i, cleaned_floyd_freqs[i]) for i in range(110, 121)))


# In[160]:
```

```
# this cleans the frequencies so that I can filter around them, and it
    also allows for multiple notes at once. Based on L=13/123
cleaned_floyd_freqs[5] = cleaned_floyd_freqs[6]
cleaned_floyd_freqs[18].append(cleaned_floyd_freqs[24][0])
cleaned_floyd_freqs[19].append(cleaned_floyd_freqs[24][0])
cleaned_floyd_freqs[20].append(cleaned_floyd_freqs[24][0])
cleaned_floyd_freqs[21].append(cleaned_floyd_freqs[24][0])
cleaned_floyd_freqs[22].append(cleaned_floyd_freqs[24][0])

cleaned_floyd_freqs[50].append(cleaned_floyd_freqs[63][0])
cleaned_floyd_freqs[51].append(cleaned_floyd_freqs[63][0])
cleaned_floyd_freqs[52].append(cleaned_floyd_freqs[63][0])
cleaned_floyd_freqs[53].append(cleaned_floyd_freqs[63][0])
cleaned_floyd_freqs[55] = cleaned_floyd_freqs[63]
cleaned_floyd_freqs[56] = cleaned_floyd_freqs[63]
cleaned_floyd_freqs[57] = cleaned_floyd_freqs[63]
cleaned_floyd_freqs[58] = cleaned_floyd_freqs[63]
cleaned_floyd_freqs[59] = cleaned_floyd_freqs[63]

cleaned_floyd_freqs[79].append(cleaned_floyd_freqs[91][0])
cleaned_floyd_freqs[80].append(cleaned_floyd_freqs[91][0])
cleaned_floyd_freqs[81].append(cleaned_floyd_freqs[91][0])
cleaned_floyd_freqs[82].append(cleaned_floyd_freqs[91][0])
cleaned_floyd_freqs[83].append(cleaned_floyd_freqs[91][0])
cleaned_floyd_freqs[89] = cleaned_floyd_freqs[91]
cleaned_floyd_freqs[90] = cleaned_floyd_freqs[91]

cleaned_floyd_freqs[110].append(cleaned_floyd_freqs[117][0])
cleaned_floyd_freqs[111].append(cleaned_floyd_freqs[117][0])
cleaned_floyd_freqs[112].append(cleaned_floyd_freqs[117][0])
cleaned_floyd_freqs[113].append(cleaned_floyd_freqs[117][0])
cleaned_floyd_freqs[114].append(cleaned_floyd_freqs[117][0])
cleaned_floyd_freqs[118] = cleaned_floyd_freqs[117]
cleaned_floyd_freqs[119] = cleaned_floyd_freqs[117]
cleaned_floyd_freqs[120] = cleaned_floyd_freqs[117]


# In[161]:


plt.figure()
xl = []
yl = []
for i in range(len(cleaned_floyd_freqs)):
    for j in range(len(cleaned_floyd_freqs[i])):
        xl.append(i)
        yl.append(cleaned_floyd_freqs[i][j])
```

```
plt.scatter(xl, yl)
# plt.plot(cleaned_floyd_freqs)
plt.show()


# In[225]:


print(list(freq_to_note(freq) for freq in floyd_freqs[::2]))


# ## Question 2
# ### Filtering the bass out of Pink Floyd

# In[163]:


cleaned_floyd = fl_floyd_spect
floyd_bass = np.zeros(cleaned_floyd.shape)
filt_window = -k_floyd[range(cleaned_floyd.shape[0])]

for t in range(cleaned_floyd.shape[1]):
    for freq in cleaned_floyd_freqs[t]:
        filt = np.exp(-(filt_window - freq)**2 / 20)
        floyd_bass[:, t] += cleaned_floyd[:, t] * filt


# In[218]:


plt.figure()
plt.imshow(np.log(abs(floyd_bass) + 1), aspect=1, cmap="binary")
plt.xlabel("time (s)")
plt.ylabel("frequency (Hz)")
plt.xticks(ticks=range(0, floyd_bass.shape[1], 10), labels=np.round(59 /
    120 * np.arange(0, floyd_bass.shape[1], 10)))
plt.yticks(ticks=range(floyd_bass.shape[0] - 400, floyd_bass.shape[0],
    10), labels=np.round(-k_floyd[range(floyd_bass.shape[0] - 400,
    floyd_bass.shape[0], 10)], -1))
plt.ylim([floyd_bass.shape[0] - 20, floyd_bass.shape[0] - 90])
plt.colorbar()


# In[143]:


bassline = []
```

```
floyd_spect = gabor(floyd_song, rate, window_time=L)
for t in range(floyd_bass.shape[1]):
    section = list(fftshift(ifft(ifftshift(np.concatenate([floyd_bass[:,
        t], np.flip(floyd_bass[:, t])])))))
#     section = list(fftshift(ifft(ifftshift(floyd_spect[:, t]))))
    bassline.extend(section)


# In[144]:


bassline = abs(np.array(bassline)) * np.max(floyd_song) / np.max(bassline)
# bassline = (255 / (np.max(bassline) - np.min(bassline)) * bassline +
    np.min(bassline))
print(bassline)
print(np.max(floyd_song))


# # Floyd Guitar

# In[283]:


L = (len(floyd_song) / rate) / (4 * len(cleaned_floyd_freqs)) # relative
    to clean floyd freqs so it can be filtered easily
floyd_spectrogram = gabor(floyd_song, rate, window_time=L, scale=0.01)
n = floyd_spectrogram.shape[0]
k_floyd = 1 / L * np.arange(-n // 2, n // 2) # the smallest magnitude
    frequency for an L-periodic function should be 1/L


# In[311]:


plt.figure()
fl_floyd_spect = abs(floyd_spectrogram[:floyd_spectrogram.shape[0] // 2])
    + abs(floyd_spectrogram[floyd_spectrogram.shape[0] // 2:][::-1])
cl_floyd = fl_floyd_spect[:, :]
# kernel = np.array([[1, 2, 1],
#                    [2, 8, 2],
#                    [1, 2, 1]])
# cl_floyd = convolve(cl_floyd, kernel, mode="wrap")
# kernel = np.array([[-2,-8,-2],
#                    [ 2, 8, 2]])
# cl_floyd = convolve(cl_floyd, kernel, mode="wrap")
ups_floyd_freqs = np.repeat(cleaned_floyd_freqs, 4)
floyd_guitar = cl_floyd[:, :]
```

```
filt_window = -k_floyd[range(cl_floyd.shape[0])]
for t in range(cl_floyd.shape[1]):
    for base_freq in ups_floyd_freqs[t]:
        for harmonic in range(1, 10):
            freq = base_freq * harmonic
            filt = np.exp(-(filt_window - freq)**2 / 20) * 0.6**harmonic
            # print(np.round(filt[-100:], 2))
            floyd_guitar[:, t] -= cl_floyd[:, t] * filt

plt.subplot(2,1,2)
plt.imshow(np.log(abs(cl_floyd) + 1), aspect="auto", cmap="binary")
plt.xlabel("time (s)")
plt.xticks(ticks=range(0, cl_floyd.shape[1], 20), labels=np.round(L *
    np.arange(0, cl_floyd.shape[1], 20)))
plt.yticks(ticks=range(cl_floyd.shape[0] - 400, cl_floyd.shape[0], 10),
    labels=np.round(-k_floyd[range(cl_floyd.shape[0] - 400,
    cl_floyd.shape[0], 10)], -1))
plt.ylabel("frequency (Hz)")
plt.ylim([cl_floyd.shape[0], cl_floyd.shape[0] - 200])
plt.colorbar()


# In[303]:


argmax_fguitar = np.argmax(abs(floyd_guitar), axis=0)
guitar_freqs = -k_floyd[argmax_fguitar]
plt.figure()
plt.scatter(np.arange(len(argmax_fguitar)), guitar_freqs)
plt.plot(guitar_freqs)
plt.show()
```

References

[1] Mallen, Alex Troy. GitHub source code.
https://github.com/AlexTMallen/AMATH-582/tree/master/HW2

[2] Mallen, Alex Troy. Fourier Analysis to Locate Moving Submarine from Noisy Data.
https://github.com/AlexTMallen/AMATH-582/tree/master/HW1

[3] Kutz, Jose Nathan. Time Frequency Analysis & Fourier Transforms.
https://www.youtube.com/watch?v=wiMge5T3d9I&feature=youtu.be

[4] Numpy fft.
https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html

[5] Scipy convolve.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.convolve.html