Max Woo

004330742

War against Gabor

Introduction

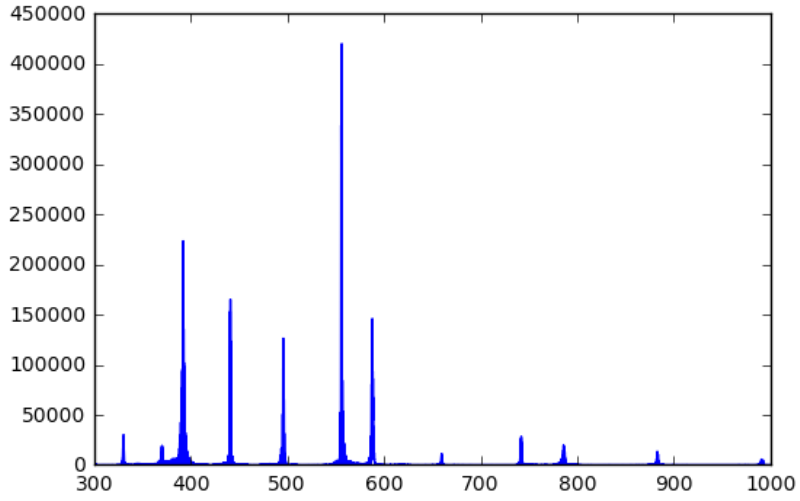
Initially, the proposed goal of this project was to try and classify musical instruments inside a song. Every instrument has a unique "timbre", a unique distribution in the frequency domain, so by analyzing the frequency distributions of each note in a song, one could try to find patterns and group notes that have similar timbres. However, I ran into many complex issues just trying to isolate the notes in relatively simple music samples, so that became the main focus of this project.

Note that this project factors into a personal goal of mine that involves creating a framework for real-time music analysis and visualization. Thus, the analysis techniques used during this project were created with the following restrictions in mind: that I can't preprocess the entire song (to determine things like tempo and key), and that my algorithms have to be relatively fast to keep up with the average frame rate of 44100 samples per second.

Also note that I included a Jupyter notebook called "Music Samples" which only contains the music samples discussed throughout the report. The other three Jupyter notebooks contain the full extend of my research and experiments.

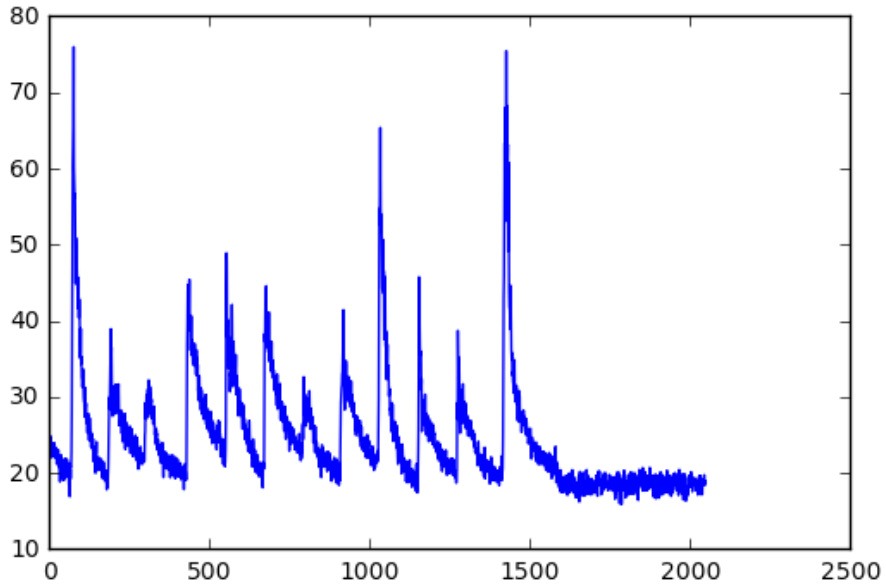
First Steps: The D Major Scale

I started with an extremely simple music sample: a piano playing the D major scale (pianoDmajorscale.wav). This song, along with every other song I analyzed in the project, has a frame rate of 44100 samples/second. Cropping the song size to the nearest power of 2, and then applying a fast fourier transform gives us:

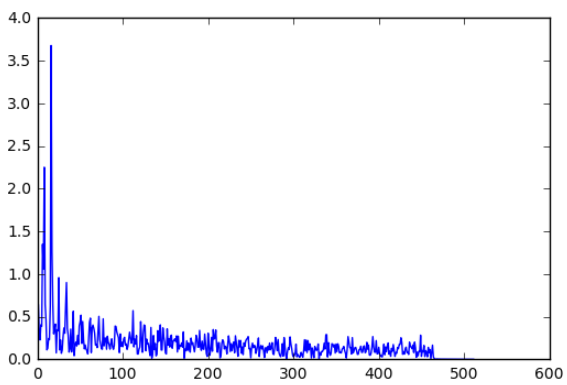


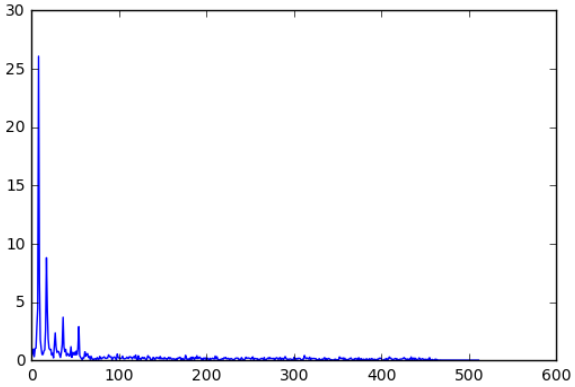
Immediately the main issue becomes apparent. While the piano only plays one note at a time, when we take the fourier transform of the entire sample, the timbres of all the notes get mixed together. This is because the concept of a "note" is based on both time domain properties and frequency domain properties. Thus, we can't simply look at the frequency domain of a song to isolate each note. However, obviously we can't just look at the time domain either, because it would be a chaotic mess of intermingled cosine waves, not to mention all the random noise. Thus, we somehow have to analyze both the frequency and time domain at the same time.

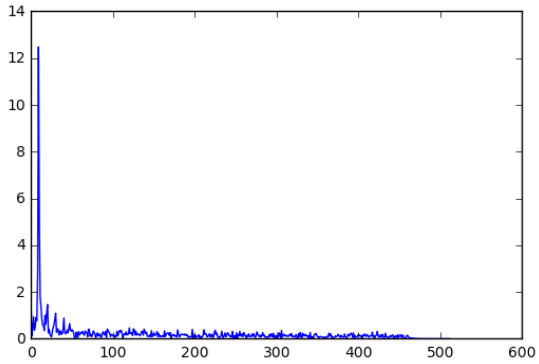
My first thought was to take a small window size of 256, and shift the window sample by sample in the time domain, taking the fourier transform at each position and looking at how the total power in the frequency domain changes over time. The result looked like this:



Now it's extremely clear where the notes are. Taking the first few notes, I took a look at the timbre of each one:







However, it is much too soon to begin classifying, after all we only have one instrument in this song! Thus, we take a look at more complex songs.

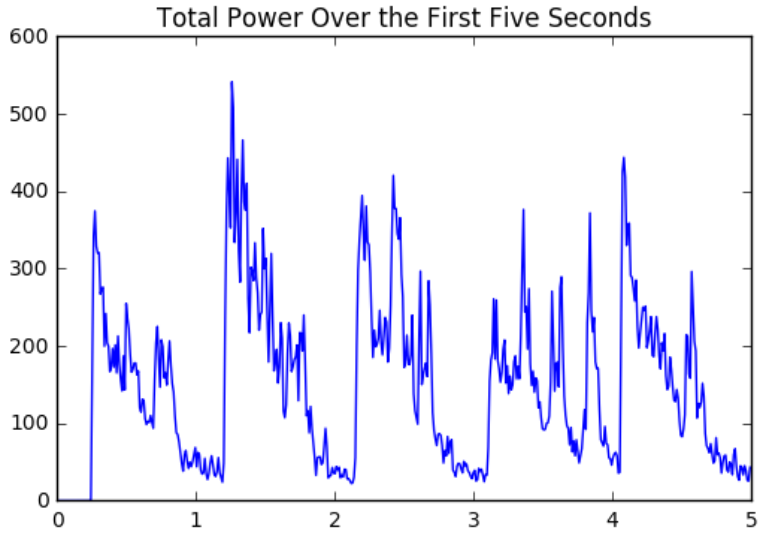
Multiple Instruments

I decided to stick with orchestral music because I thought they would be simpler with less of the fancy effects that modern music has (like reverb, sawtooth waves, vocal chopping, etc). I chose the song Sakae in Action (sakaeinaction.wav) from the movie Summer Wars because it starts off with very distinct piano notes and violin plucks, and then slowly gets more complex and adds more instruments.

We start with the first 5 seconds, which contains 3 sets of one piano note and two violin plucks, and then a final set of one piano chord and three violin plucks, something like

PVV PVV PVV PVVV

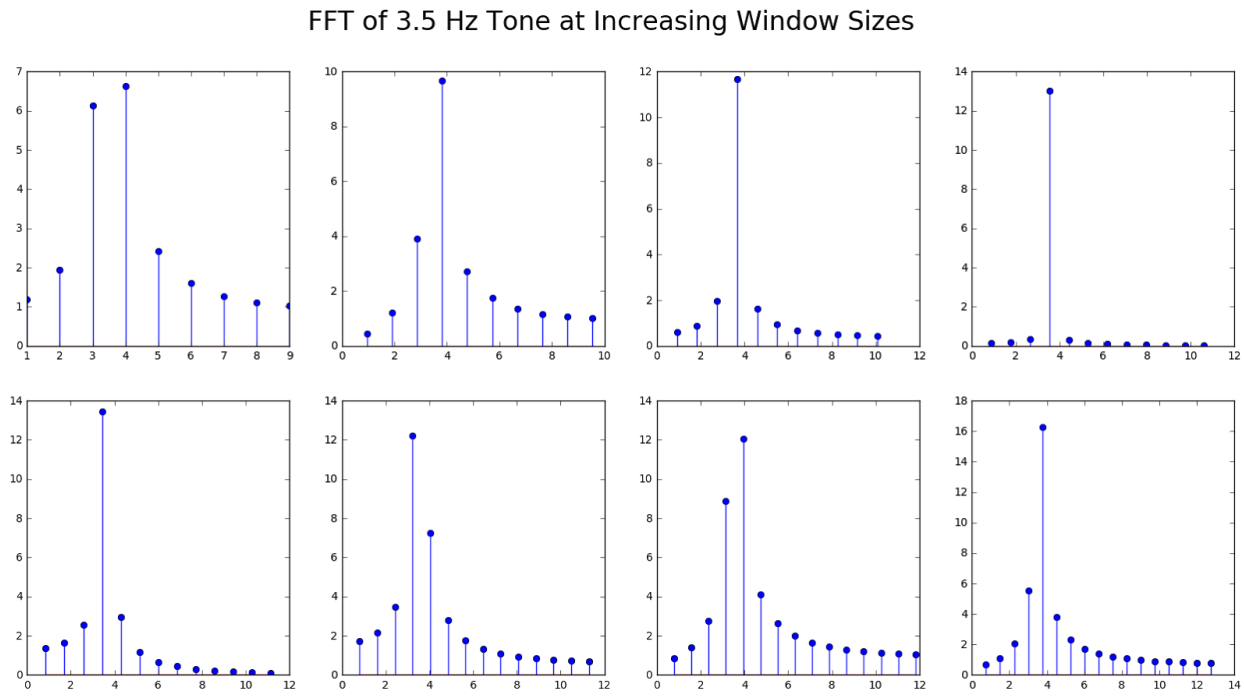
where P represents a piano chord and V represents a violin pluck. Just like with the D major scale sample, we look at the total power over time:



The pattern is sort of evident, but it is clearly much more difficult to extract the notes from this sample than it was for the D major scale.

After a bit of online research on spectral analysis I came across the concept of the “Gabor limit”. Essentially, the idea is that the more precision you have in the frequency domain, the less information you have about the time domain, and vice versa. Remember that for our case, in order to identify and classify instruments, we first need to find notes in the music. Musical notes have both a shape in the frequency domain (the “timbre”) and a shape in the time domain (the “attack”).

If we focus on the frequency domain, we can see that choosing a large window size can give a higher “resolution” because it increases the number of measured frequencies without increasing the highest measured frequency. This is demonstrated below, where the FFT of a 3.5 hz tone is taken at different window sizes. Notice that when the tone falls directly between two measured frequencies, like in the top left graph, the distribution looks more like a sinc() function and not a sharp spike. However, in general, as the spacing between measured frequencies decreases, the spike becomes sharper. There is also a “sweet spot” at window size 24 (the top right graph), because one of the measured frequencies happens to be extremely close to 3.5



However, there is a downside to constantly increasing window size and increasing resolution. If a song has multiple notes played in quick succession, than using a window size that encompasses all those notes will “blur” the shapes of those notes together, rendering the instruments indistinguishable from each other. In addition, the larger the window size, the more the “attack” of a note (it’s envelope in the time domain) begins to influence the fourier transform of the signal.

However, there seems to be quite a bit of recent research[1] that show how our ears transcend the Gabor limit, analyzing songs much better than fourier analysis could, and suggesting that our ears and brain use processing techniques much more complex and effective as compared to traditional frequency analysis.

Perhaps the key behind the human hearing is having a structure, the cochlea, that can capture multiple resonance frequencies at the same time (higher frequencies towards the base and lower frequencies towards the apex). Having a structure fine tuned to resonate with a certain frequency means that it only needs enough "data samples" for one period of its resonance frequency to get a magnitude reading on that frequency. While a 4096-point Fourier transform would take 1/10 of a second to get enough data to examine a one kHz tone (at a standard 44100 samples/second), a resonating structure would theoretically be able to detect it instantly, only needing 1/1000 of a second (the period of a 1 kHz tone).

We can simulate this with multiple "samplers" simultaneously computing the fourier transform of the signal, each with a different window size corresponding to the minimum window size needed to capture each frequency. Now, ideally, we would have a sampler for every single frequency, but that would require thousands of samplers, each running their own O(nlogn) FFT algorithm on each sample, at a rate of 44100 samples per second. This is clearly not practical, but in hopes of getting comparable results, we try the following procedure:

- Use a preliminary sampler with a relatively large window size, capturing 20 Hz as its lowest frequency (commonly accepted as the lowest pitch distinguishable by the human ear)
- From the FFT of the preliminary sampler's samples, examine the power of each frequency, and determine the dominant frequencies
- For each of the dominant frequencies, dynamically spawn a sampler tuned to that frequency (window_size = sample_rate/frequency)
- For each of these samplers, compute a running FFT for every window along the data, and only examine the power of the frequency that sampler was made for
- When the power of that frequency "peaks", the note is probably at its loudest, so extract the shape of the fourier transform at that time to get the optimal "timbre"

The Sliding DFT

Note that, in addition to focusing only on dominant frequencies, in order to efficiently compute the DFT for a moving window, we use a variation called the sliding DFT, which is able to compute the fourier transform of a "sliding window" in O(N) time for each shift (instead of recomputing the entire DFT, a O(nlogn) operation):

[1] Byrne, Michael. "How Your Ears Do Math Better Than Mathematicians" http://motherboard.vice.com/blog/how-your-ears-do-math-better-than-mathematicians

$$\mathcal{F}[x_0, x_1, x_2, x_3] = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} \quad \text{where} \quad \omega = e^{-2\pi j \frac{1}{N}}$$

$$\mathcal{F}[x_1, x_2, x_3, x_4] = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} (X_0 - x_0) + x_4 \\ (X_1 - x_0)/\omega + \omega^3 x_4 \\ (X_2 - x_0)/\omega^2 + \omega^2 x_4 \\ (X_3 - x_0)/\omega^3 + \omega x_4 \end{pmatrix}$$

By precomputing the "w_1 vector" ([1, w^k, w^{2k}, … w^{k*N-1}]) and the "w_{N-1} vector" [1, w^{k*N-1}, w^{k*N-2}, … w^k], the code for this shifting operation is simple:

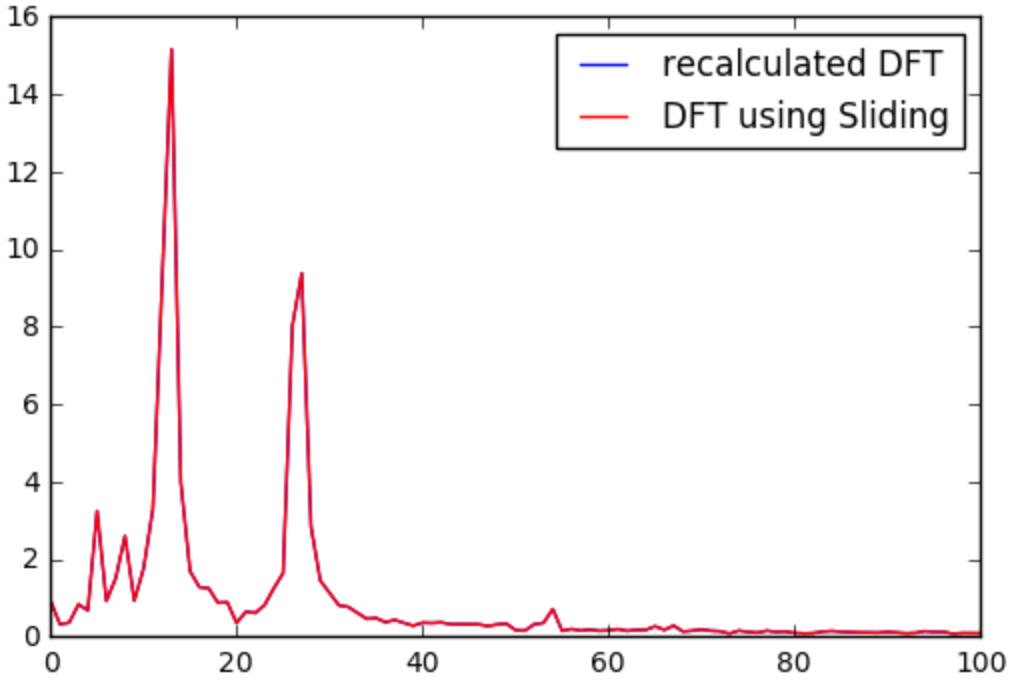
```
new_dft = (old_dft-old_sample)/w1_vec + new_sample*wN_vec
```

Using this, I constructed a Python class that would quickly give me the DFT of a shifting window:

```
class Sampler:
    def __init__(self, song, N, offset=0, Fs=44100.):
        self.song = song
        self.N = N # sample size
        self.Fs = Fs
        self.w1_vec = np.fft.fft(np.arange(N) == 1) # extract the [1 w w^2 w^3...] vector
using an fft of [0 1 0 0 0 ...]
        self.wN_vec = np.fft.fft(np.arange(N) == N-1) # extract the [1 w^(N-1) w^2(N-1) ...]
using an fft of [0 0 0 ... 0 1]
        self.frequencies = frequencies(np.arange(self.N), self.Fs)
        self.reset(offset)

    def next(self):
        if (len(self.song) <= self.offset+self.N):
            return np.zeros(self.N) # song finished, just return zeros

        old_datum = self.song[self.offset]
        new_datum = self.song[self.offset+self.N]
        self.offset += 1
        self.dft = (self.dft-old_datum)/self.w1_vec + new_datum*self.wN_vec
        return self.dft

    def reset(self, offset=0):
        self.offset = offset
        self.dft = np.fft.fft(self.song[self.offset:self.offset+self.N])
        return self.dft

    def current_sample(self):
        return self.song[self.offset:self.offset+self.N]
```

One of the possible problems with this approach is "drift", small floating point errors accumulating over time to cause larger errors later on. Because with the sliding DFT, each result is dependent on the previous result (whereas normally each DFT is calculated independently), there is a large possibility of drift. To measure this, we use a window size of 1024 and shift the

window by 102,400 samples using the sliding DFT algorithm. Then we manually calculate the fourier transform of the 102,400th sample and compare the results:

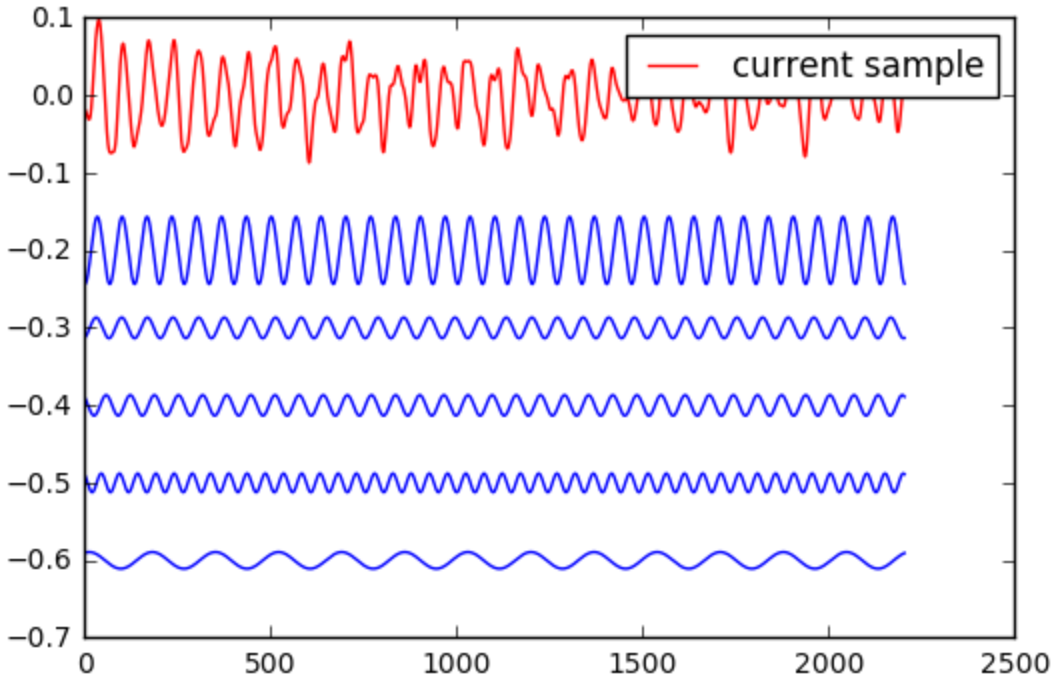


Surprisingly, it looks like there is zero drift. To test speed, we calculate the sliding DFT for 51,200 shifts and compared it with manually calculating the fast fourier transform of each shift. There is a slight improvement, 1.177 seconds for the sliding DFT and 1.304 seconds for manual recalculation.

Note that there is a distinct advantage of the sliding DFT over manual recalculation. The FFT gets its speed from factoring the window into smaller and smaller sizes, whereas the sliding DFT does not. Thus, our window size can be a prime number, which will only significantly slow down the initial fourier transform and not the shifts. With a window size of 1009 and a total of 1009 shifts, FFT recalculation took 2.0256 seconds while the sliding DFT took 0.0378 seconds, a significant improvement of about 100x!

Extracting Dominant Frequencies

Following the procedure outlined earlier, we create a sampler with a window size of 44100/20=2205, the minimum size needed to capture 20 Hz. We also offset the Sakae sound clip to start at 0:01 (between first “PVV” and second “PVV” sets). Then we extract the top 5 frequencies of the first window. Below is the time domain of the first window (2205 samples), and cosine waves of the top 5 frequencies:



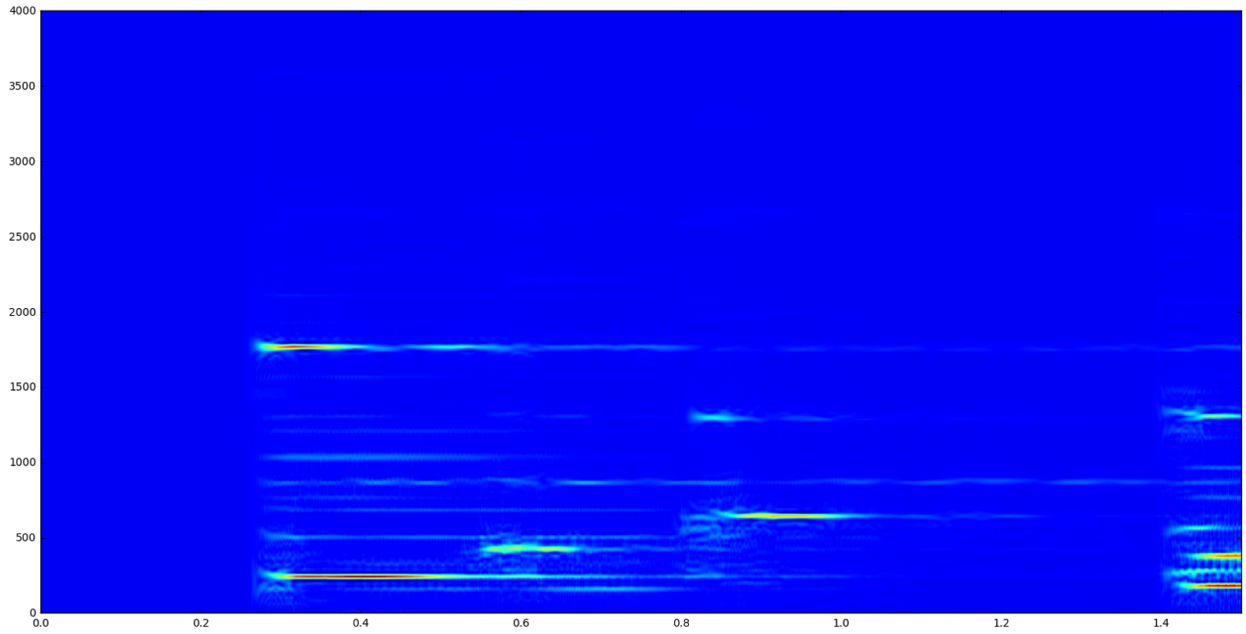
The frequencies are 660 Hz, 640 Hz, 680 Hz, 900 Hz, and 260 Hz.

Note that these are the sounds lingering after the first “PVV” set. Listening to the music by ear, the first PVV set is

- Piano chord : F4 (350 Hz), A6 (1760 Hz)
- violin: A4 (440 Hz), E5 (660 Hz)

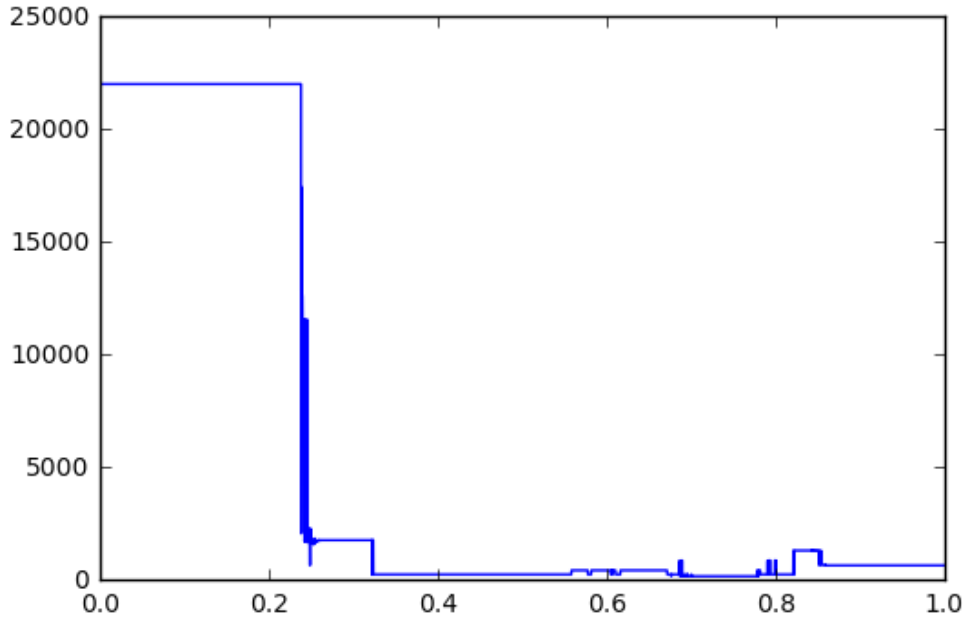
Of the top 5 frequencies detected, 660, 640, and 680 are all around 660 Hz (E5), which makes sense because E5 is the last note played. Note that 900 Hz is about ½ of 1760 Hz (A6), probably the loudest note of the first “PVV” set. Because strings often vibrate in harmonics (multiples of the main pitch), this also makes sense.

To get a better visual of the first “PVV” set, we can create a spectrogram by shifting the window along the first second and plotting the fourier transforms as columns of an image (with red indicating frequencies at a higher power and blue indicating frequencies at lower power):



I capped the top frequency at 4000 Hz and averaged every 10 columns to shrink the image to a reasonable size. From this, one can clearly see the first piano chord, and then the two violin notes following it. We extract the dominant frequencies at a 1 second offset, time "1.0" on the spectrometer. The dominant frequencies in the spectrometer at that point also seem to match the ones we computed earlier.

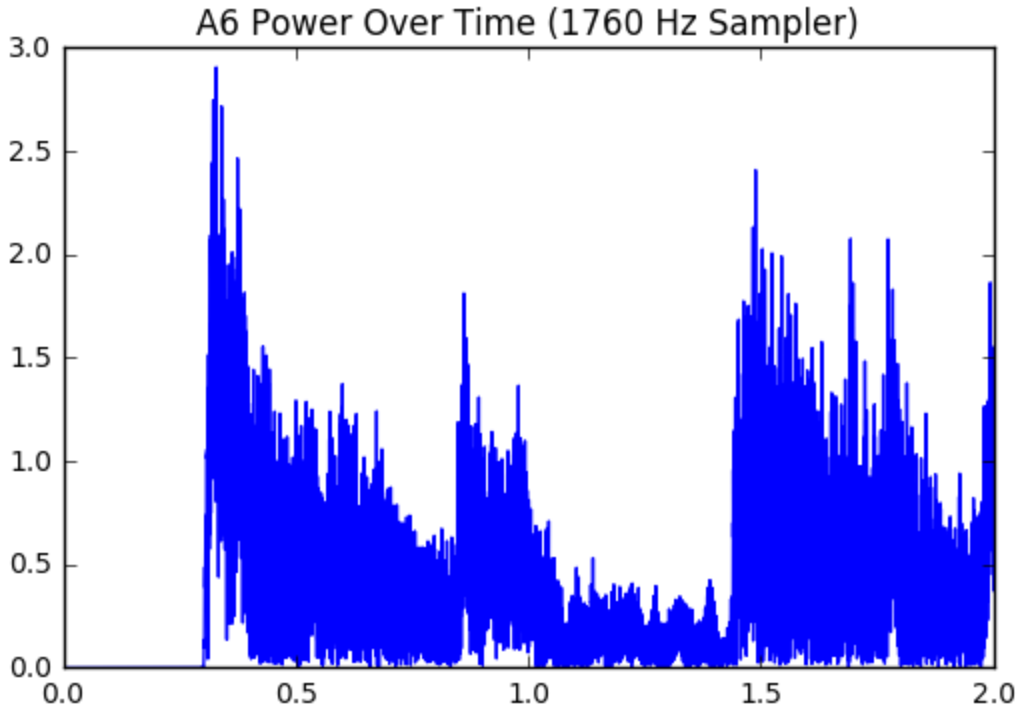
This technique is starting to look promising, so let's take a look at the top frequency over time. That is, for every window up to one second, we plot the frequency of the strongest frequency in the fourier domain:



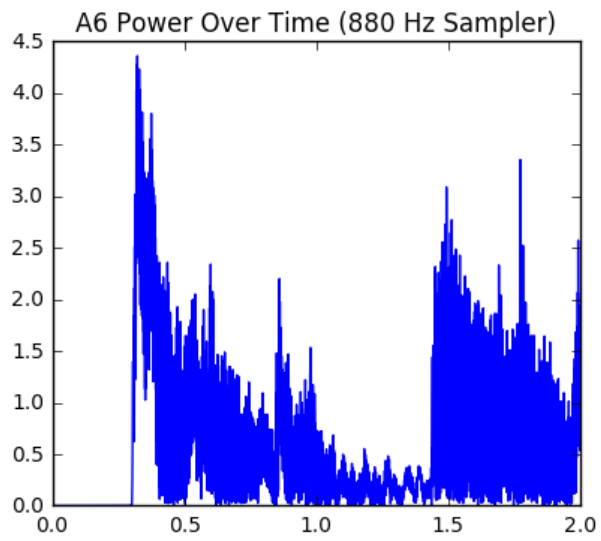
While we can still see 1760 Hz dominating at 0.3 seconds and 350 Hz dominating a short while after (because higher frequencies tend to die out faster), there is clearly a lot of other frequencies we don't want, especially the ~22500 Hz frequency dominating before the song even started. Thus, it's clearly not enough to focus on dominant frequencies, we also have to take a look at other factors.

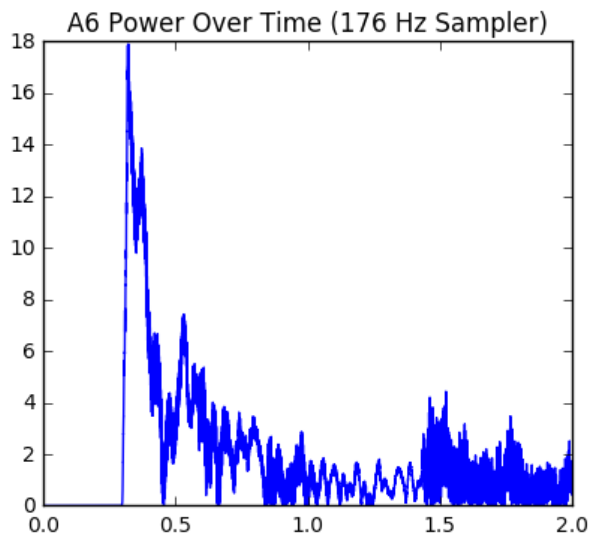
Finding the Optimal Window Size

Before we moved on to other methods of note extraction, we first examine the "fine-tuned samplers" idea. I chose to focus on the top note of the first piano chord, the A6, with a frequency of 1760 Hz. This is a good test for the samplers idea because the frequency is relatively high, so the 20 Hz sampler we were using before would lose much of the time domain data, while the multiple-samplers method hopefully wouldn't. The minimum window size needed to capture this frequency is 44100/1760 ≈ 25 samples. After creating a sampler with this window size, we use the sampler to extract the magnitude of the 1760 Hz frequency for every window up to 1 second. The plot is shown below:

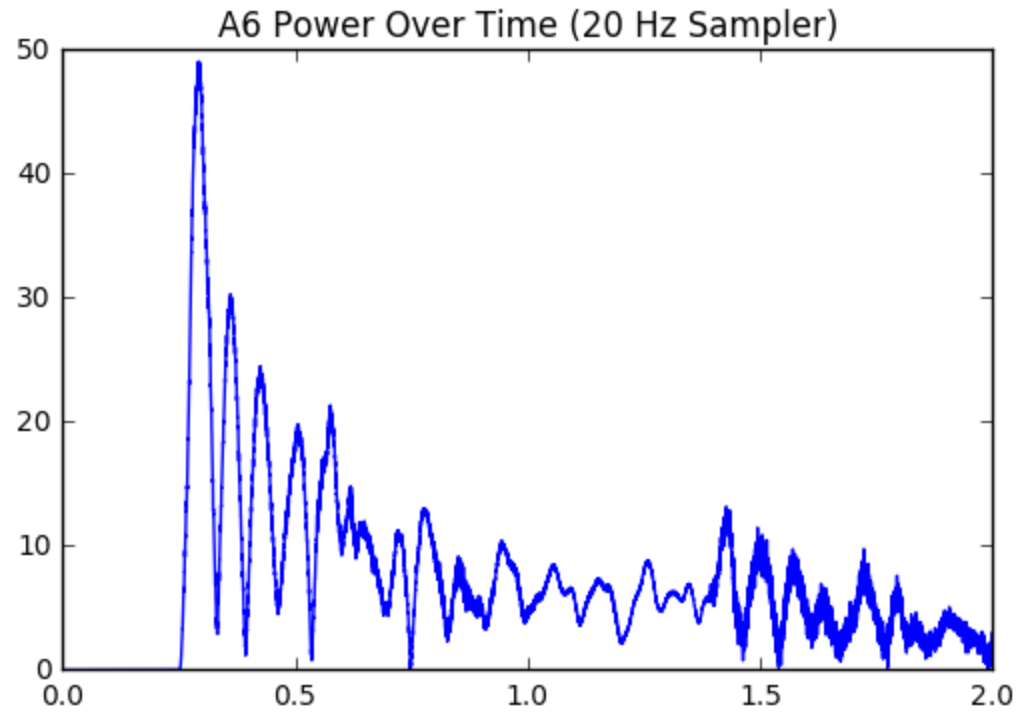


Notice how noisy the graph looks, indicating that perhaps the technique isn't as effective as I'd hoped. We can lower the sampling rate to capture two periods and 10 periods, corresponding to a 880 Hz sampler and a 176 Hz sampler

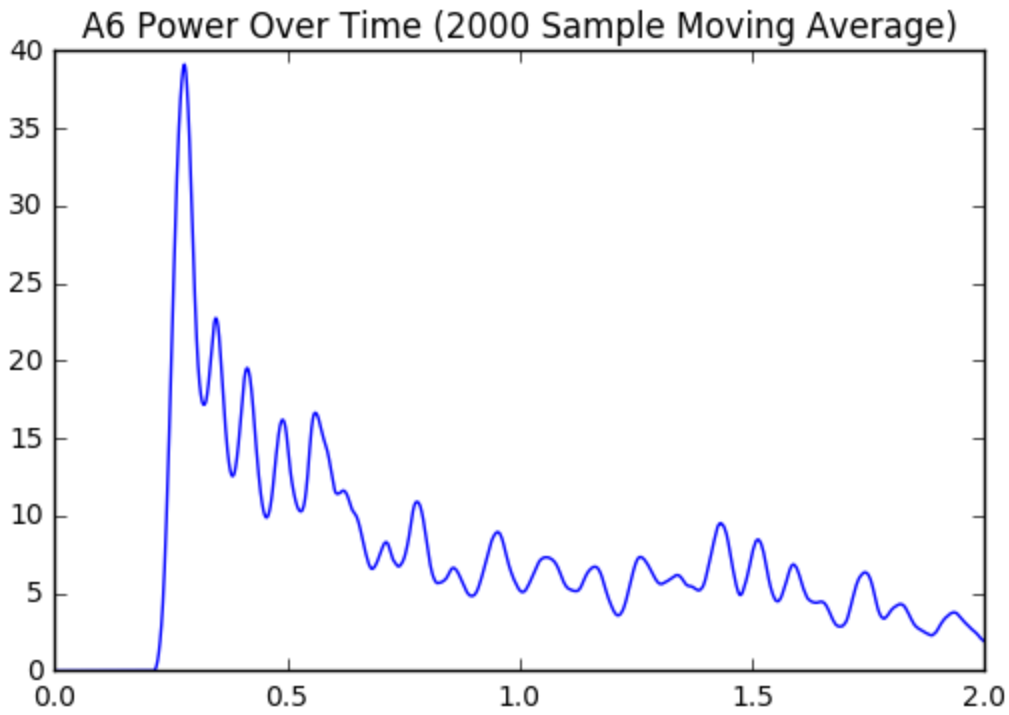




The graph seems to be getting better. Let's try 20 Hz, the sampler earlier used to extract dominant frequencies ("tuned" for human hearing, which detects 20 Hz at the lowest)

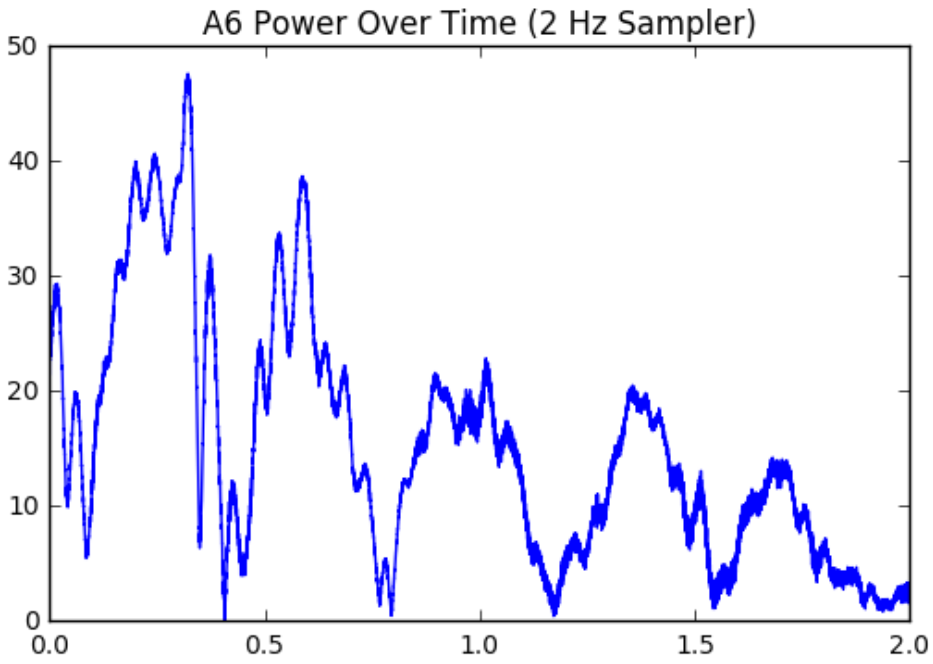


This graph actually looks better than the graph we got from the "tuned" 1760 Hz sampler. The spikes could easily be smoothened out with an averaging mechanism. For example, a 2000 sample moving average seemed to smoothen out the graph quite a bit:



Thus, it is probably sufficient to simply use the 20 Hz sampler from now on. Unfortunately this means scrapping the idea for multiple frequencies, but it is possible that we can refine it later on.

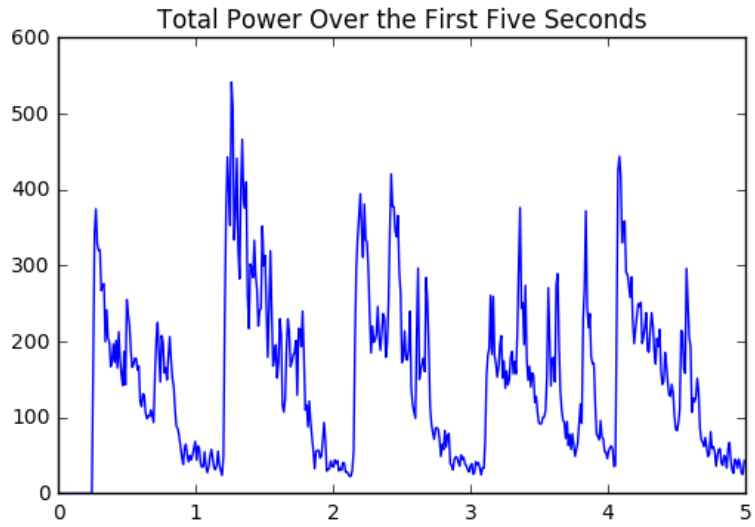
Lastly, we look at what a 2 Hz sampler would give:



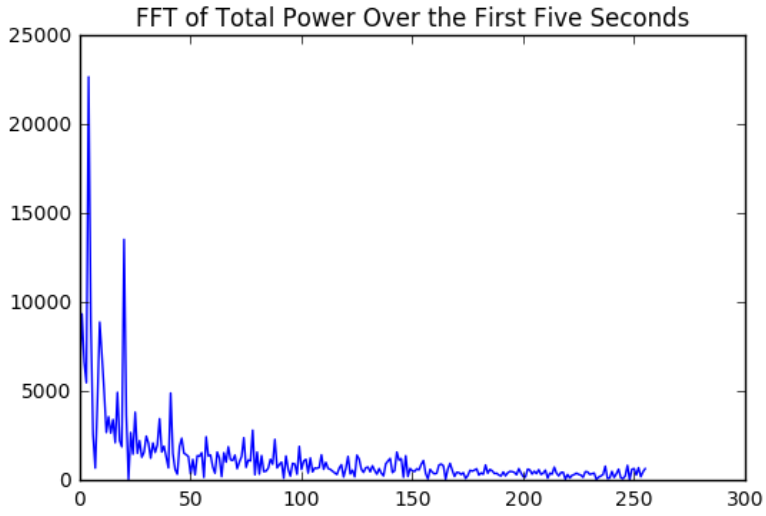
As suspected, and as the Gabor limit suggests, as the window size continues to increase the graph starts to get worse. Thus, it did still seem like there was an optimal window size for each frequency. However, for simplicity we stick with the 20 Hz sampler.

Extracting Power Peaks

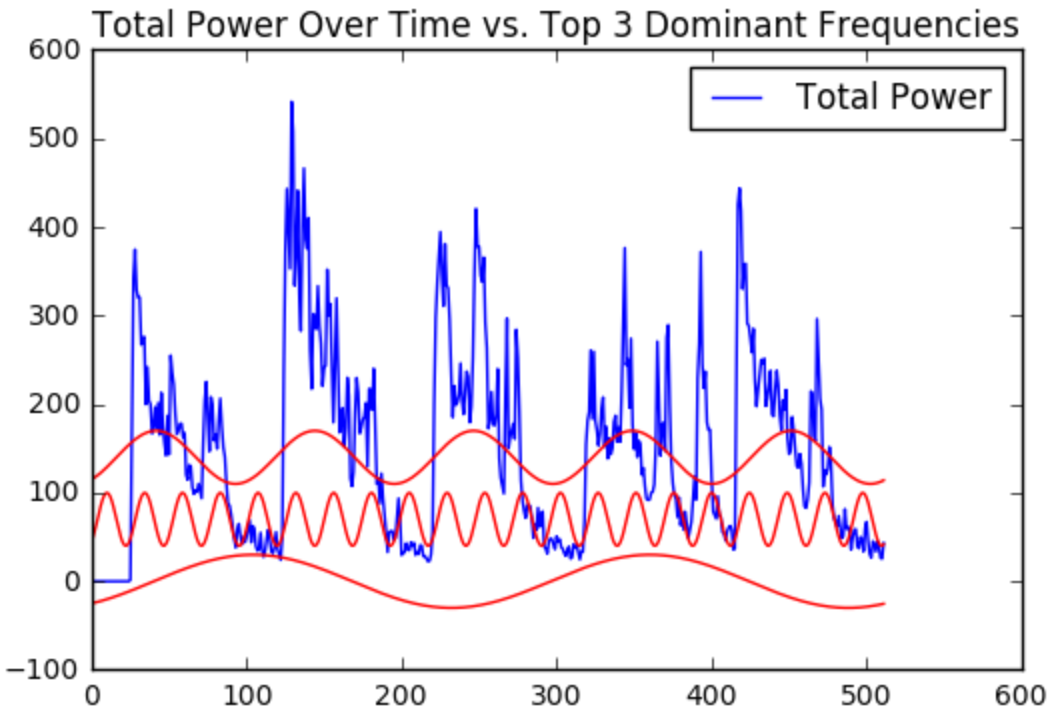
As shown previously, merely extracting dominant frequencies is not enough to isolate notes. An alternate method is to look at when the total power is at a "peak". Let's look at the total power of the first five seconds



Remember that this follows the pattern "PVV PVV PVV PVVV" (and starts over at t=4 seconds). What is immediately noticeable is the periodic nature of the graph. Building off of this observation, one could analyze the tempo and bpm (beats per minute) of the music in order to get a general sense of where the notes should be. Thus, we take the fourier transform of the total power over tim

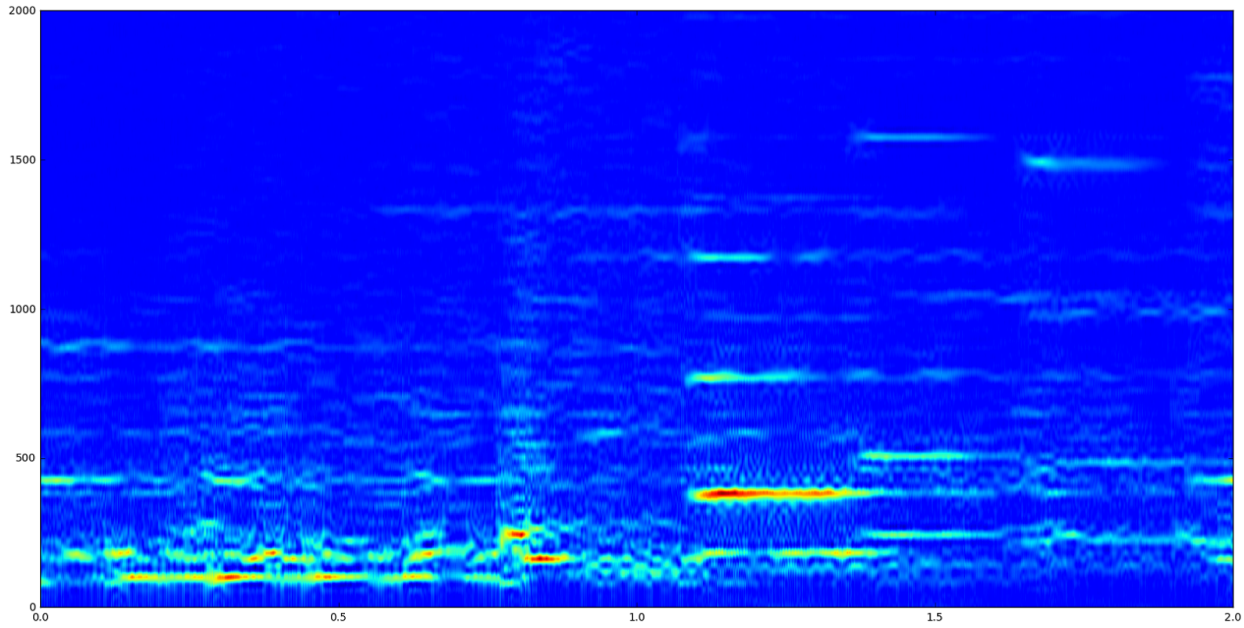


There are clearly a few peaks in this graph, so we extract these dominant frequencies and overlay them over the graph of the total power over time to see if they line up, making sure to calculate the phase of each frequency and factoring that in as well. The graph shows the original total power graph in blue, and cosine waves of the top 3 frequencies in red, with the most dominant frequency offset towards the top.

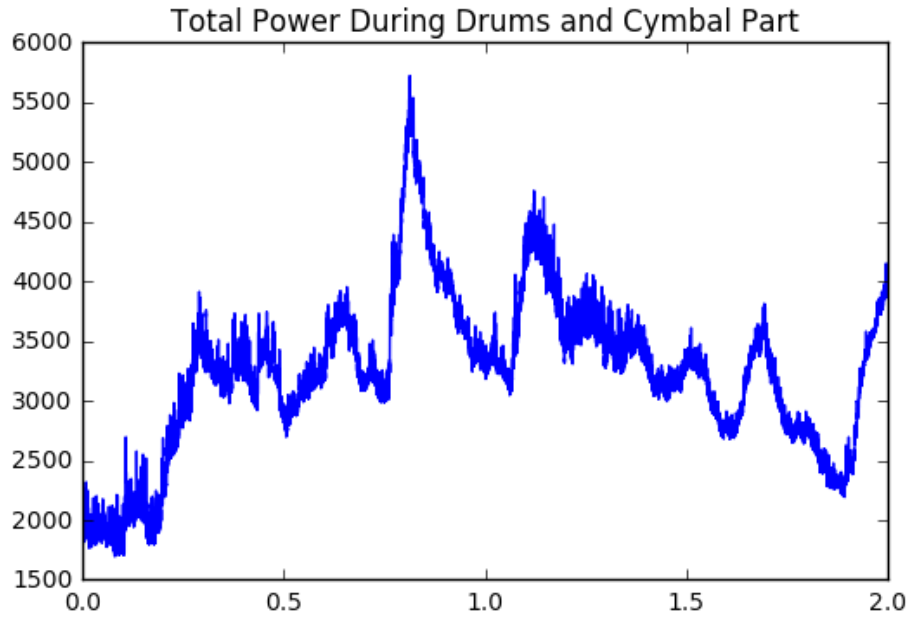


We can see that the dominant frequency lines up well with each set, while the second dominant frequency lines up with individual notes. However, the third dominant frequency does not really line up with anything. So to use this method effectively, I would need to determine a threshold to determine which dominant frequencies are good indicators of tempo, and which are not.

In addition, note that we are taking advantage of the unique periodic nature of this song clip. However, there are plenty of songs that do not have a consistent tempo, or that change tempo part way (eg jazz improvisations). In fact, there are many clips in this song itself where there is no noticeable pattern, and the tempo might be unclear. For example, at 2:34-2:36 in the song, there is a gradual cymbals build up to a large “clash” of drums and cymbals, and then the piano picks back up. Plotting the spectrogram we get:



An important thing to look at here is the “clash” at ~0.8 seconds. Compared to the cymbals build up on the left and the piano notes on the right, it does not look like much. However, listening to the sound clip it stands out above the rest. Plotting the total power over time of this graph gives a good indication why:



While none of the frequencies in the “clash” are particularly strong, there are so many frequencies involved that the total power ends up higher than the rest. In fact, this large distribution of low magnitude frequencies seem to be a characteristic of many percussion instruments, which is why they can sound loud even without having a discernable “pitch”. This also highlights another flaw in the dominant-frequency-extraction method used earlier. While

that method may work well for “pitch” based instruments like the violin or piano, it would not be able to detect percussion instruments.

Another important thing to notice is the total power of the three piano notes after the “clash”. From listening to the music, they clearly occur at evenly spaced intervals between 1 and 2 seconds. This seems to match up with the spectrogram, where there are three distinct sets of stripes between 1.0 and 2.0. However, in the total power graph, there are 6 spikes with widely varying sizes and no apparent pattern. In order to correctly isolate the notes we have to somehow group certain spikes together while leaving others separated. In fact, contrast this with the cymbals that build up to the “clash”. While there are multiple spikes in this region, from the music and spectrogram it sounds/looks like a single “note”, albeit a very messy one without a pitch. Thus, it is not sufficient to purely look at the total power either, and we will have to somehow combine dominant frequency analysis and the total power analysis.

Finding the Difference

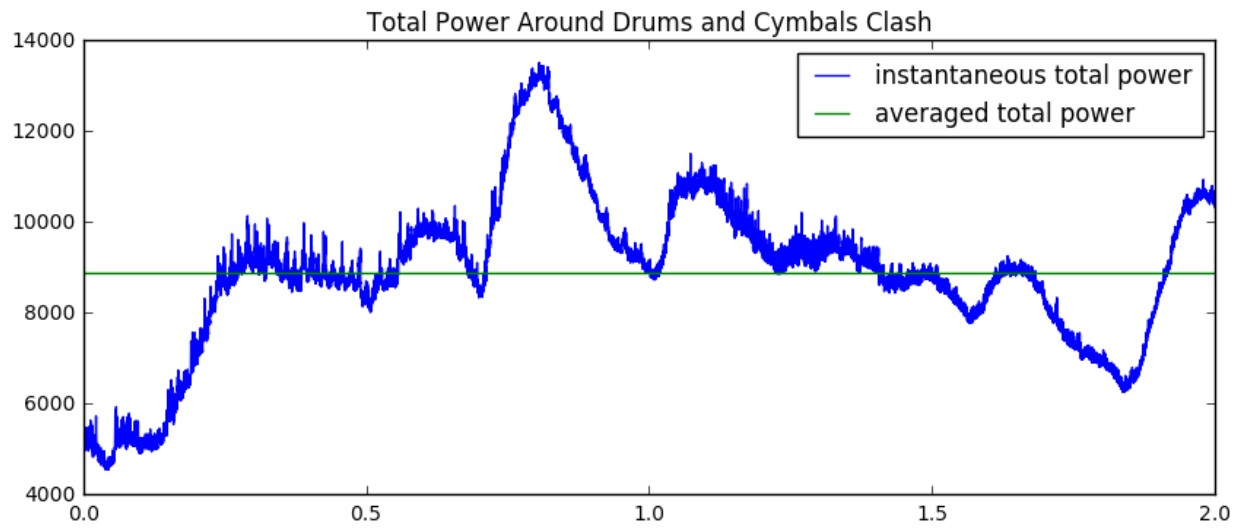
One of the key observations I made in order to solve the aforementioned issue is that in the total power graph, when two spikes are close together and have the same dominant frequencies they should "blur" together into one note, but spikes that are close together with very different frequencies should not. Thus, we can leverage this with the following procedure:

1. Keep track of the last-registered note’s dominant frequencies
2. As we shift the window, find the difference between the current frequency distribution and the previous notes distribution.
3. When the total power of the difference exceeds a threshold, record that as the new “last-registered note”, and repeat
4. At the same time, keep track of the last-registered note’s current power, and unregister it when it dips below a certain threshold

There are many ways to define the “difference” between a frequency distribution and a note. A naive implementation might simply subtract the two. However, in order to factor in the possible noise in the frequency distribution of the note, I opted to find the note’s dominant frequencies, and exclude those frequencies from the current frequency distribution. This also allows the current note to increase in volume after exceeding the threshold without registering as a new note.

What about the first note, where there is no “last-registered note”? With no dominant frequencies to exclude, this would simply be the total power of the frequency distribution. So in order to find the first note, we simply look at the unmodified total power distribution.

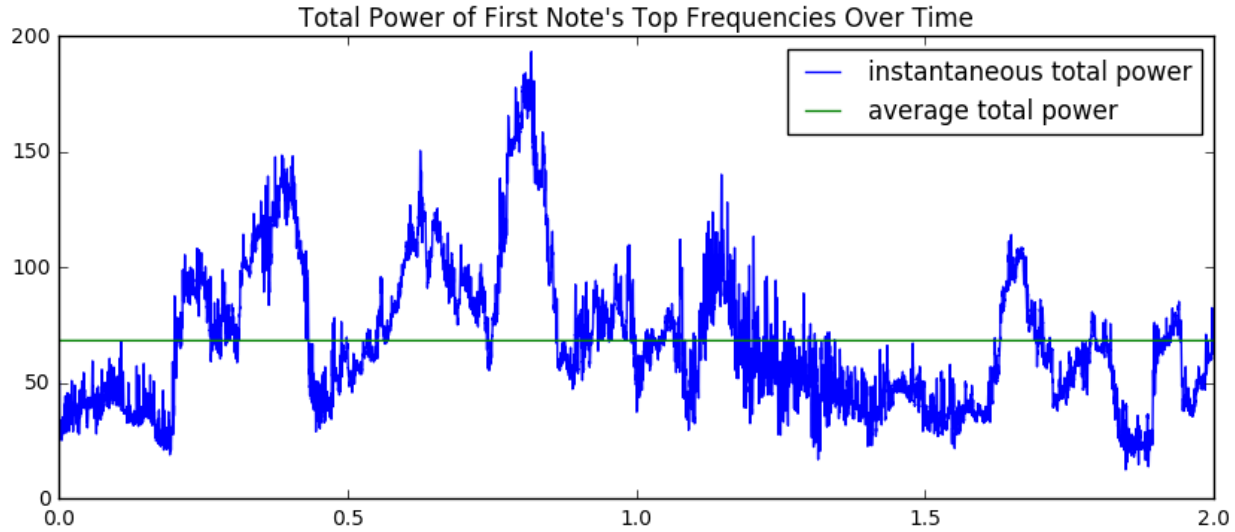
The last thing we need to define are the thresholds. One of the practices I try to avoid is to simply find a threshold through trial and error. While ultimately there are going to need to be some hard-coded constants or functions, I prefer a flexible and dynamic technique in order to better account for variation and noise. Thus, one of my first instincts was to take the average of the total power and then compare it with the total power graph:



We can see that the first note begins around 0.2 seconds. Tracing the first time the instantaneous power tops the average power gives us an exact time of 0.211 seconds. From here, we find the top 5 frequencies, which come out to be:

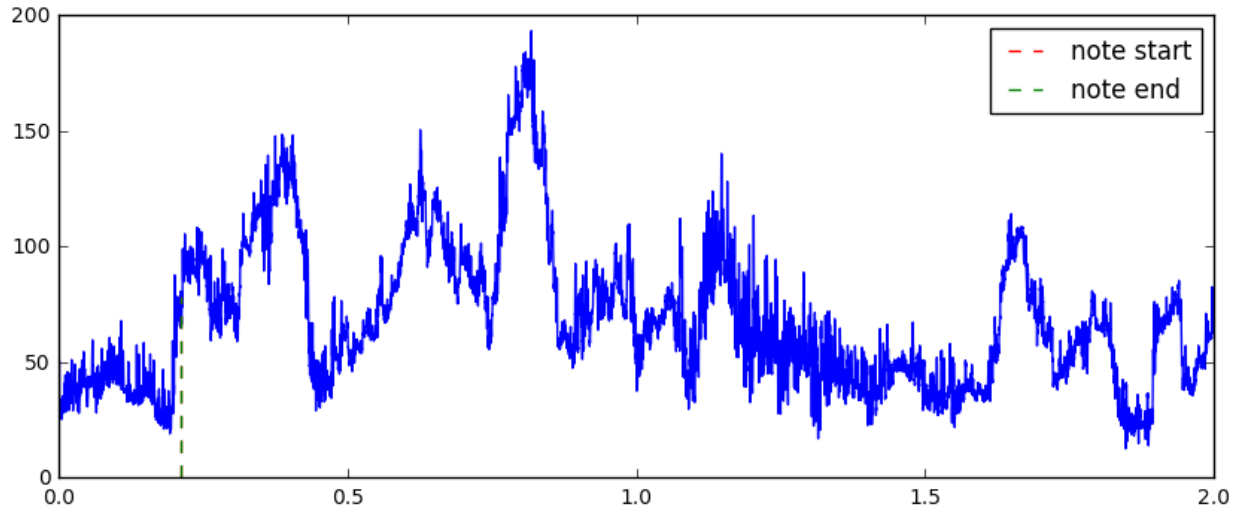
21718 Hz, 7313 Hz, 7978 Hz, 7535 Hz, 7756 Hz

We also need to find a threshold for determining when the note ends. We first examine the total power graph of just the dominant frequencies:



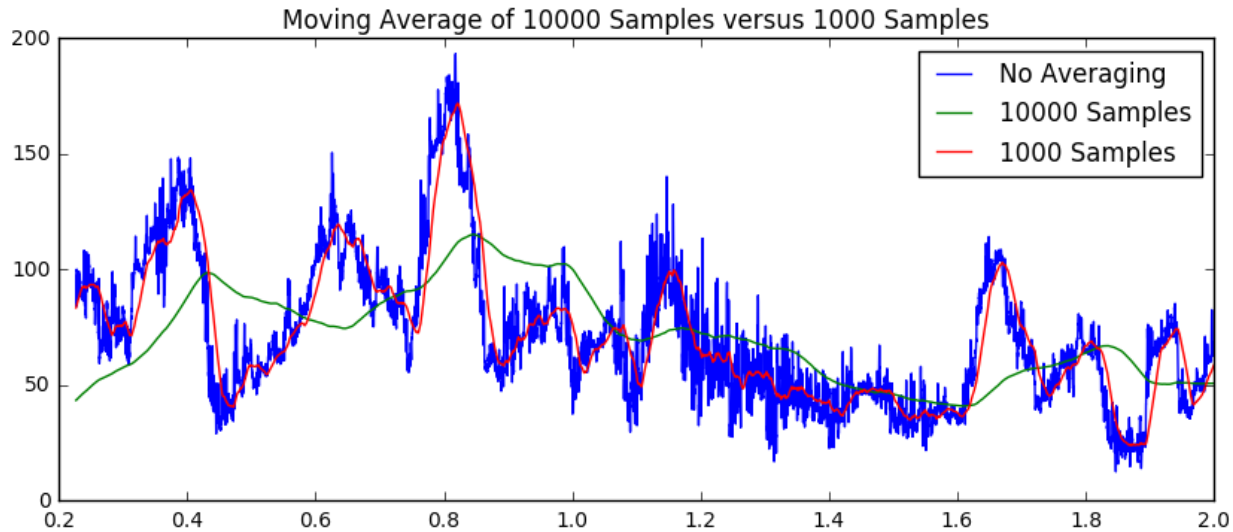
One method we can try is to start at the the point where instantaneous power rises above average power, and then check when the instantaneous power dips back below the average total power. It looks like this would give us the point between the two spikes in the beginning (from 0.2-0.5 seconds), but that should be good enough for our use cases.

However, when we actually try to employ this method, we get the graph below:



Notice that the lines delimited the note start and note end are so close together that they overlap. It seems like the noise in the instantaneous power makes it dip below average power almost immediately after rising above it.

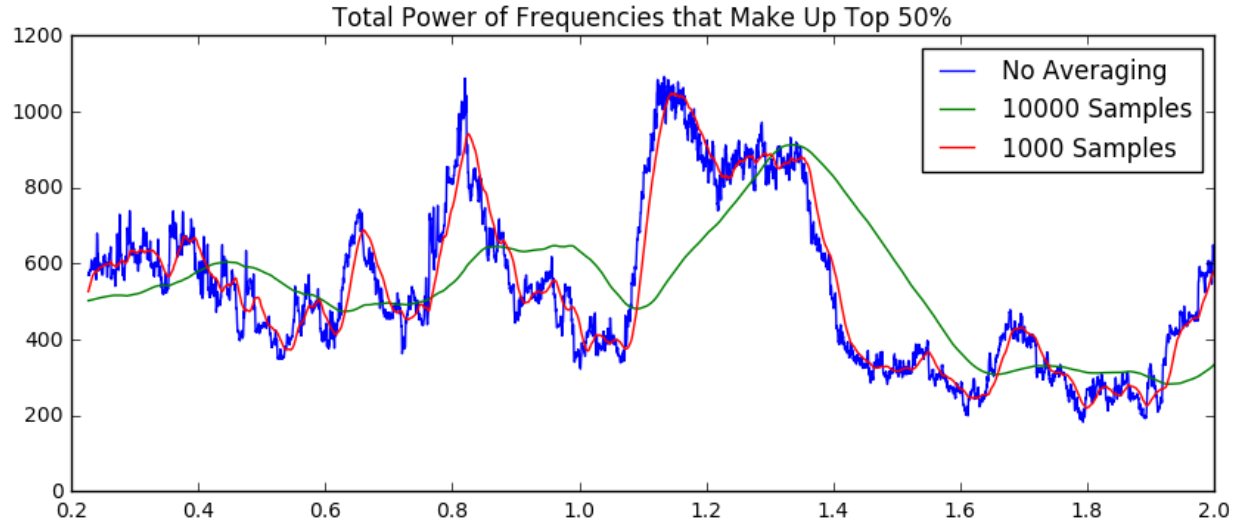
In order to solve this, we could simply take the moving average of the instantaneous power. However, a more effective and flexible method involves comparing the difference between two moving averages, one with a long window size and one with a short window size. Essentially the short moving average serves to remove noise, while the long moving average creates a threshold. Instead of comparing the short moving average to a static threshold like we were doing before (using the total average), using a long moving average makes the threshold dynamic, increasing during louder parts of the music and decreasing at smaller parts. Testing this out on the graph above, we get:



Observe that because the largest moving average is 10,000 samples, which is approximately 0.25 seconds, we can only calculate the long moving average after 0.25 seconds, so that is where our graph starts. On the other hand, the results of this method look fantastic. The two spikes between 0.2 and 0.5 seconds are now classified as a single note. A lot of the other "peaks" as perceived by the human eye are also being captured by the technique.

However, as noted earlier, the cymbals build up lasts all the way to 0.8 seconds. Instead of looking at the total power, let's focus on the cymbals. We look at the first instance at which the cymbals note is detected, and then average the next 1000 frequency distributions to get a rough "timbre" of the cymbals instrument. Then, we find which frequencies make up the top 50% of the total power of this timbre. The advantage of looking at the total power of the top frequencies, rather than just taking the top 5 frequencies, is that for instruments with large distributions like percussion instruments, the number of frequencies we "capture" gets larger in order to reach the 50% sum.

Taking the double moving average of this, we get



One of the major differences between the graph of the total power of the cymbal's timbre, and the total power of the song, is that in the cymbals graph the gap between 0.4 and 0.6 seems to be reduced. This is good, and should help us ultimately blend those two notes into one. In addition, The cymbals graph has a large spike from 1.1 to 1.4 seconds, aligning with the piano instruments. However, this shouldn't matter too much because by that time the cymbals note would have ended, and so I would not be tracking these frequencies anymore.

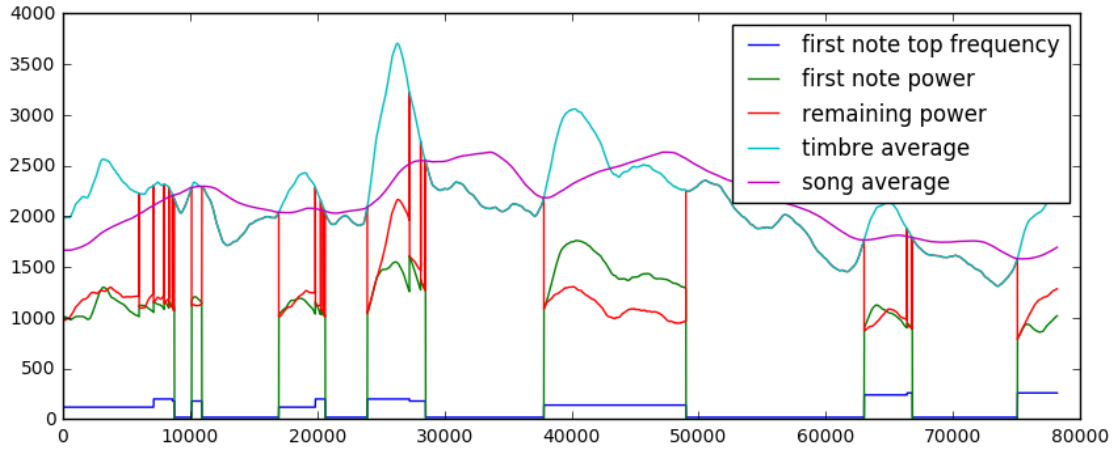
At this point, I decided it was time to put all the pieces together and look at the result. I followed the procedure at the beginning of "Finding the Difference" and factoring in fourier transforms, moving averages, and a difference mechanism all in a single Python class, `NoteExtractionSampler` . I did make a few modifications. First, to make things simpler I used the song's total power for the long window average, instead of looking at the total power of each individual note. In addition, for the short window average I simply took the average power of the last 1000 frequency distributions. Then, I could use this short window average to both

detect notes and to get the average timbre of a nope. In addition, instead of just keep track of the last registered note, I keep track of an array of registered notes that have not ended yet, and subtract all of them from the current total power to get the remaining power. Lastly, because I used the 50% threshold to get the top frequencies of each timbre, the note power will be around 50% of the total short-window power. Thus, when looking for the end of the note, I multiply the note power by 2 before comparing it against song average power. The complete code is shown below:

```
class NoteExtractorSampler(Sampler):
    def __init__(self, song, N=Fs/20, offset=0, Fs=44100., frequencyCap=4000):
        self.songAvgWindow = 10000
        self.noteAvgWindow = 1000

        Sampler.__init__(self, song, N, offset+self.songAvgWindow, Fs)
        self.frequencyRange = (self.frequencies > 0) & (self.frequencies < frequencyCap)

        tempSampler = Sampler(song, N, offset, Fs)

        # initialize the long averaging window
        startPower = np.sum(power(tempSampler.dft[self.frequencyRange]))
        initialSongAvg = [np.sum(power(tempSampler.next()[self.frequencyRange])) for i in
range(self.songAvgWindow-1)]
        initialSongAvg = np.append([startPower], initialSongAvg, axis=0)
        self.songAvg = RollingAverager(initialSongAvg)

        # initialize the short averaging window
        tempSampler.reset(offset+self.songAvgWindow-self.noteAvgWindow)
        startTimbre = power(tempSampler.dft[self.frequencyRange])
        initialTimbreAvg = [power(tempSampler.next()[self.frequencyRange]) for i in
range(self.noteAvgWindow-1)]
        initialTimbreAvg = np.append([startTimbre], initialTimbreAvg, axis=0)
        self.timbreAvg = RollingAverager(initialTimbreAvg)

        self.notes = []
    def nextNote(self):
        self.next()
        currPower = power(self.dft[self.frequencyRange])
        self.songAvg.nextSample(np.sum(currPower)) # increment the averaging windows
        self.timbreAvg.nextSample(currPower)

        allNoteFrequencies = []
        # calculate each note's power
        for note in self.notes:
            if (np.sum(self.timbreAvg.average[note]) > self.songAvg.average):
                allNoteFrequencies = np.union1d(allNoteFrequencies, note)
            else:
                self.notes.remove(note) # note below average song power, remove it

        # use a boolean mask to exclude all current notes from the frequency spectrum
        mask = np.ones(len(self.frequencies[frequencyRange]), dtype=bool)
        mask[allNoteFrequencies] = False

        remainderTopFreqs = self.top50PercentFreqsIndices(self.timbreAvg.average[mask])
        self.remainingPower = np.sum(self.timbreAvg.average[remainderTopFreqs]) # calculate
power of top frequencies of remainder
        if (self.remainingPower > self.songAvg.average):
            self.notes.append(remainderTopFreqs) # if above threshold, register as new note

        return self.notes
    def top50PercentFreqsIndices(self, timbre): # calculate the top frequencies that add up to
50% of total power
        power50perc = 0.5*np.sum(timbre)
        sortedindices = np.argsort(timbre) # note: ascending order
```

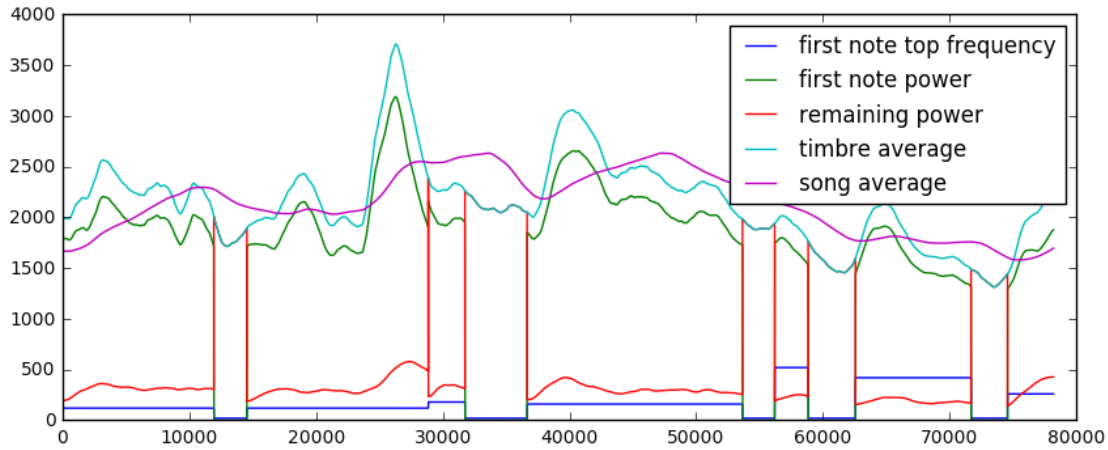
```
        sortedtimbre = timbre[sortedindices]
        cumulativeSum = np.cumsum(sortedtimbre)
        return sortedindices[cumulativeSum >= power50perc] # when cumulative sum surpasses
50%, the remaining numbers will sum to 50%
```

I ran this procedure on the drum and cymbals clip of the song, and plotted the results:



Notice the many vertical lines in the "remaining power" graph. This represents places where a note was either detected or removed. However, the cluster of vertical lines towards the beginning indicates that a note is being detected and then immediately removed, and then immediately re-detected, etc. Also, notice that the remaining power is always around 50% of the timbre average, and seems to follow the same shape as well. This means that when detecting a note, we are excluding a lot of the frequencies that are part of the instrument.

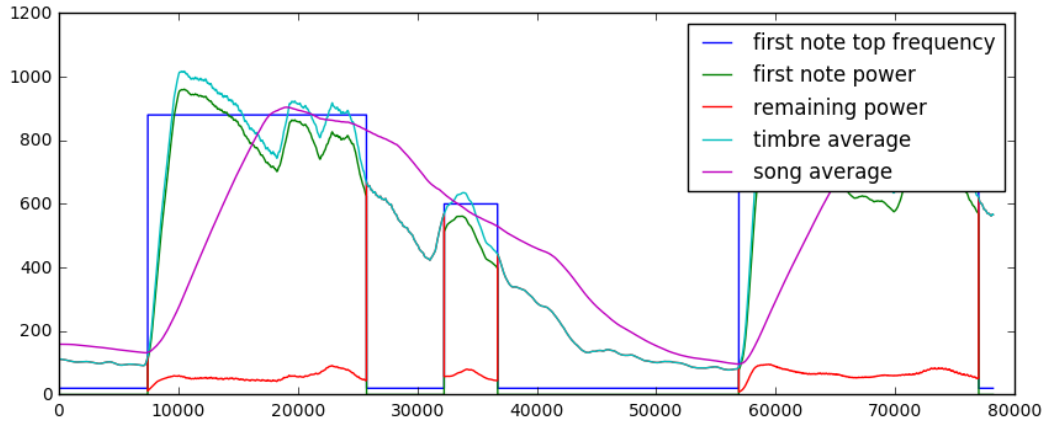
To fix both these issues, I decided to capture the frequencies that sum to 90% of the total power. Then, when looking for the end of a note, I overcompensate by multiplying the note power by 1.2 before comparing against the final note power. This overcompensation gives a bit of "wiggle room" for the note power, reducing the chance of notes being detected and then immediately removed. Lastly, I decreased the song average to 0.9 times its original value, further increasing note detection. The result of these tweaks is shown below:



This graph looks pretty good, and shows a huge step towards accurate note classification. The number of notes in the graph matches the number of notes in the clip: a cymbals build up, a drum clash, and then three piano notes. We were even able to isolate and extract the three piano notes, that visually look meshed together. Lastly, the note power shapes seem to match the timbre average shapes extremely well, indicating that we are extracting the correct frequencies.

However, there are still some glaring issues in the graph. First, the cymbals build up got split up into two, but this time the second section merged with the drum clash. In addition, note that I was never really able to detect multiple notes at the same time (which would look like the "remaining power" graph dipping down twice in a row). This seems to be because I am capturing too many frequencies using the 90% detection method, so the remaining power ends up way too low to ever exceed the song average. Note that if we capture only one note at a time, then we might as well not even use "remaining power" and just use the average timbre! On the other hand, recall that we increased the threshold to 90% because the shape of the "remaining power" graph suggested that we weren't capturing enough frequencies. While it's impossible that we simultaneously capture too much and too little frequencies, what is possible is that the two notes share many frequencies, and thus "giving" all the frequencies to the first note prevents the second note from being detected.

There is always more fine tuning and optimization that can be done. Nonetheless, for the scope of this project I decided that this was good enough. For good luck, I tested my technique on the beginning of the song, the "PVV" pattern discussed earlier (offset by 1 second)

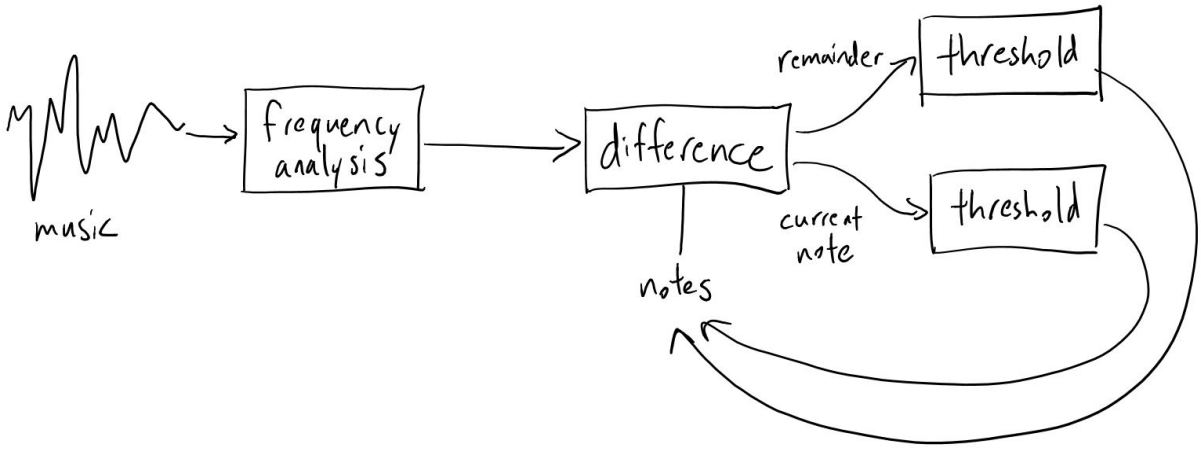


It does not seem like luck is on my side, The first two notes of the pattern got blurred together into one note. I realize that one of the main issues is that while my short average is based on the remaining frequencies, my long average is not, so in the first two notes on the graph, the song average power increases steadily while the remaining power drops drastically. Thus, it would probably be better to exclude the note frequencies from the song average as well. However, the project has to end at some point, so I leave this for a later time.

It is worth noting that my final script for analyzing songs took much longer than expected, about 5-10 minutes just to analyze 2 seconds (88200) samples. And this is while using the optimizations for shifting DFT and averaging windows, which should be linear operations at the slowest. This speed would be unacceptable for real-time data analysis purposes. Thus, in the next section I outline the major optimizations and improvement I thought of during my project.

Next Steps and Optimizations

Our final procedure for isolating notes (outlined in the beginning of the section “Finding the Difference”) essentially boils down to a feedback loop that looks like the following:



Throughout my research and experimentation, I came up with many optimizations and improvements on each of the components of this procedure. While I unfortunately did not have time to test many of these, I explain my ideas below to hopefully emphasize the overall potential of this technique.

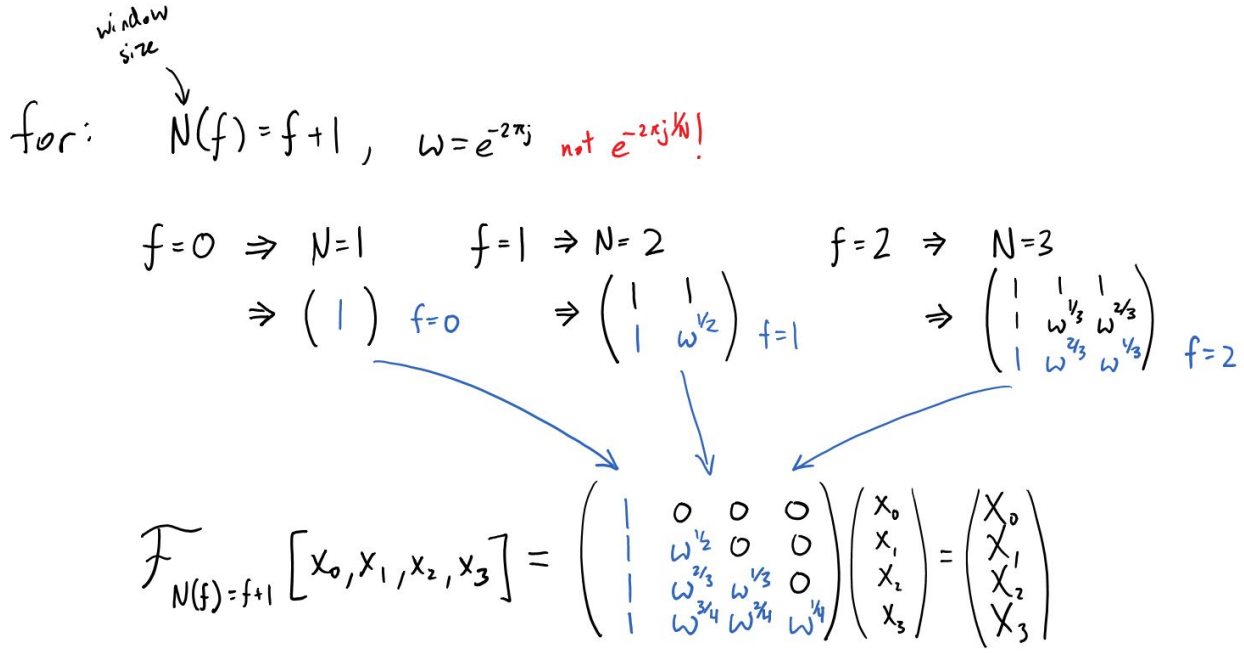
Frequency Analysis

Recall that every term in the fourier transform is calculated independently by multiplying the corresponding row of the fourier matrix by the column vector of time domain data. During our analysis, most of the time we were only interested in a subset of the total set of frequencies calculated through the fourier transform. To speed calculations up, instead of cropping the frequency domain result, we can crop the fourier matrix to contain only the rows corresponding to frequencies we are interested in.

This method can be extended to the fast fourier transform as well, with the same reasoning that even with the FFT, each term is calculated independently. Thus, if we are interested in K terms of an N-point dataset, the time complexity becomes O(KlogN). Likewise, this method can be used for the sliding DFT as well, reducing the time complexity from O(N) to O(K). In most cases we were only interested in frequencies between 0 and 4000, and with our unmodified fourier transforms giving us a range of -22050 Hz to 22050 Hz, an O(K) runtime would speed things up by a factor of 10.

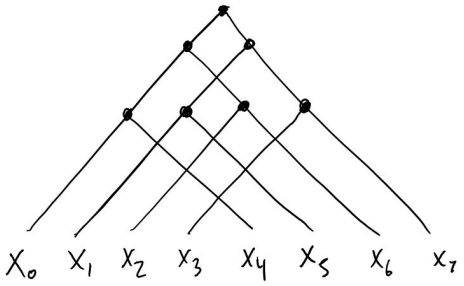
Furthermore, we can use this idea when employing the multiple-samplers technique discussed earlier. For the multiple-samplers technique, each sampler examines only one frequency, so the fourier matrix for each sampler is reduced to one row. In addition, because each sampler maps to a unique frequency and vice versa, we can combine all these single-row

fourier matrices into a single fourier matrix, padding 0's at the end of each row to make all rows the same length. For example, if the ideal window size for a given frequency *f* is given by the formula *N(f) = f+1*, we end up with a lower-triangular fourier matrix:

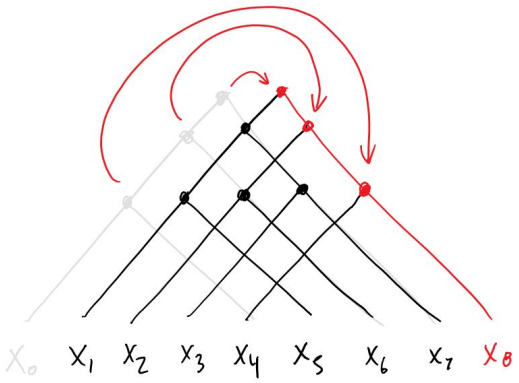


However, note that for different functions of *N(f)*, such as the one we originally came up with, N(f)=(Fs/f) where Fs is the sampling rate, then the row sizes won't always increase linearly in size and give us a nice triangular matrix.

One last optimization I explored in the context of frequency analysis is extending the sliding DFT algorithm to the fast fourier transform. The idea is to cache the intermediate steps of the FFT and take advantage of them when calculating the fourier transform of the shifted the window. Recall that the fast fourier transform works by successively splitting the data into even and odd parts, and calculating the transform at the leaves, and then recombining the leaves back up the tree:



In the diagram, the numbers at the bottom are time domain samples, and the black nodes represent recombination steps. If we shift the samples, the new FFT would involve some of the cached values:



In the diagram above, the old FFT nodes that need to be discard are shown in grey, and the new nodes that need to be added are shown in red. The nodes in black are present in both the old and new FFT and can thus be reused. The red arrows indicate how we can modify the recombination formula to quickly convert grey nodes into red nodes. For each node, we simply "negate" the old grey child and adding the new red child.

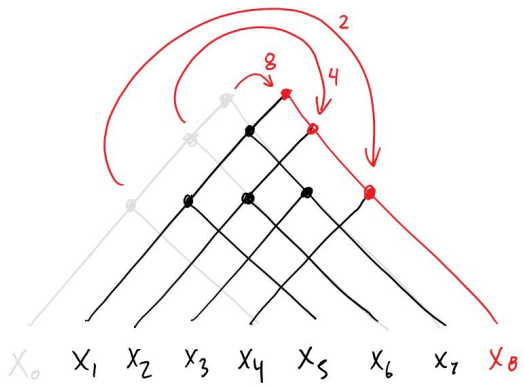
For N=4, the last modified-recombination step for N=4 looks like:

$$\text{for: } \mathcal{F}[x_0, x_2] = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix}, \quad \mathcal{F}[x_1, x_3] = \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}, \quad \mathcal{F}[x_2, x_4] = \begin{pmatrix} C_0 \\ C_1 \end{pmatrix}$$

$$\mathcal{F}[x_0, x_1, x_2, x_3] = \begin{pmatrix} A_0 + \omega^0 B_0 \\ A_1 + \omega^1 B_1 \\ A_0 + \omega^2 B_0 \\ A_1 + \omega^3 B_1 \end{pmatrix} = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} \qquad \text{note: } \omega = e^{-2\pi j \frac{1}{4}}$$

$$\mathcal{F}[x_1, x_2, x_3, x_4] = \begin{pmatrix} B_0 + \omega^0 C_0 \\ B_1 + \omega^1 C_1 \\ B_0 + \omega^2 C_0 \\ B_1 + \omega^3 C_1 \end{pmatrix} = \begin{pmatrix} (X_0 - A_0)\frac{1}{\omega^0} + \omega^0 C_0 \\ (X_1 - A_1)\frac{1}{\omega^1} + \omega^1 C_1 \\ (X_2 - A_0)\frac{1}{\omega^2} + \omega^2 C_0 \\ (X_3 - A_1)\frac{1}{\omega^3} + \omega^3 C_1 \end{pmatrix}$$

However, notice that to subtract the old values and add in the new values, it takes O(N) steps. The total time complexity thus becomes O(N+N/2+N/4+N/8...) = O(2N) = O(N), which is the same as the time complexity of the normal sliding DFT.



However, note that if we were to "jump" the window by N samples, the sliding DFT would need O(N^2) steps, and ends up calculating the DFT for every shift in between. On the other hand, the sliding FFT would not be able to reuse any of its cached values, and would end up re-calculating the FFT, an O(NlogN) operation. Thus, if we consider "jumping" windows and not sliding windows, then the sliding FFT would have better time complexity than both the sliding DFT and the normal FFT. Moreover, "jumping" windows could prove especially useful when dealing with real-time data, and we need to jump a certain amount of samples at a time to keep up with the incoming stream.

Difference

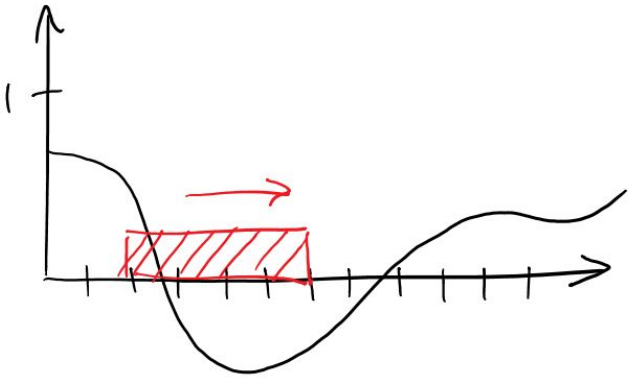
This is arguably the most crucial part in the feedback loop, because it combines all the current frequency information and an "archive" of past notes to split the frequency distribution into two parts, one representing a continuation of the past notes and one representing possible introduction of new notes. In addition, it is probably the most flexible out of all the parts. There are many different ways we could have implemented this module. Our method of excluding the last registered note's frequencies doesn't account for when the next note has some overlapping frequencies. One way to factor this in is to use Least Squares curve fitting or PCA to try and estimate how much of the current frequency distribution is part of the last registered note.

We could also look at the slope of the last note's power over time to predict what the power of the note will be on the next sample, and subtract that prediction from the next sample's frequency distribution. In fact, the decay rate of different frequencies that are part of the same note can help during instrument classification.

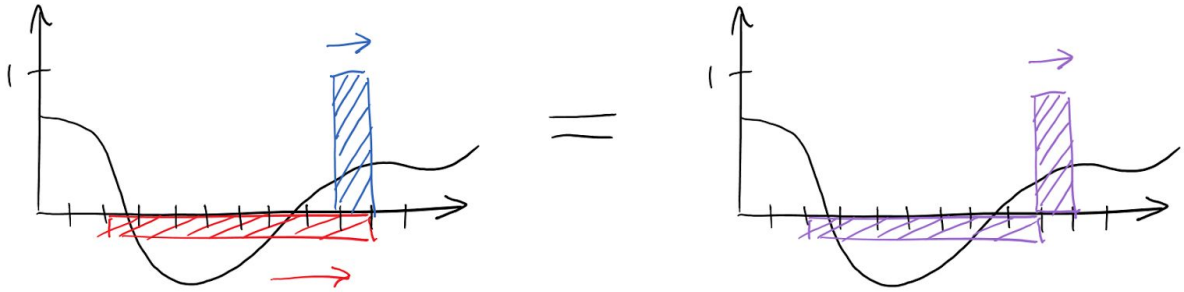
In addition, we could factor in other attributes of the music. For example, if we had a module that estimated the tempo and bpm of the song, we could feed that into the difference module to get a better sense of where notes should be and shouldn't be. A note pattern recognizer can take the notes registered by the difference module and look for patterns in the notes, which can also help predict notes. This would work especially well for our use cases because music tends to contain many repeating patterns, as demonstrated by the "PVV PVV PVV PVVV" pattern in the beginning of Sakae in Action.

Averaging and Thresholds

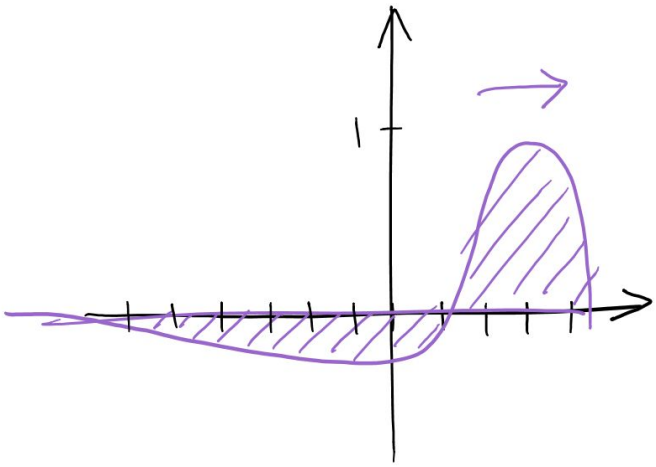
Averaging was a common theme throughout the entire project, serving as a quick and easy way to reduce noise. In fact, the double moving average was probably the single most effective technique discovered during my research. However, there are still improvements and enhancements that can be made to it. First, notice that moving averages can be visualized as a sliding rectangle function in the time domain, multiplying element-wise with the data to produce a sliding average:



We can also see that the area of such a rectangle should be 1 (because for an average across N points, the height at each point would be 1/N). Now consider two sliding averages, one with a very long window and one with a very short window. If subtract the wider window from the thinner one, we get the following:



Note how the total area of the purple shape is 0. We can generalize this shape to any sort of curve with a tall “head”, a wide “tail”, and a total area of 0:



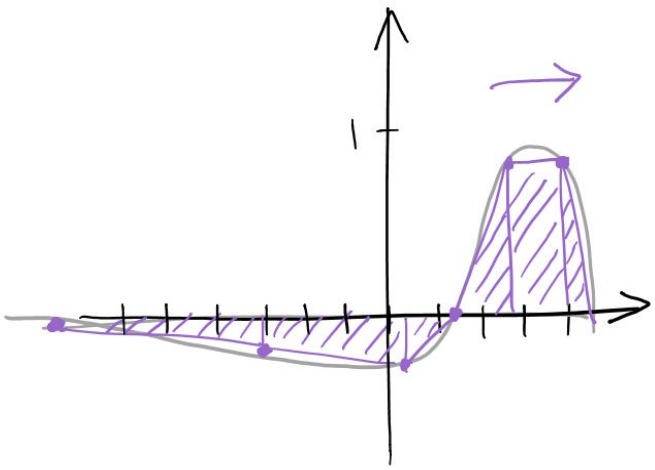
For example, a curve like the one above would achieve around the same effect, except with the tapered tail at the left, it should more accurately mimic the way our ears work. We can further fine tune the shape of this averaging curve to perfectly suit our needs.

However, note that one of the main benefits of the rectangular averaging functions is that it is quickly computable. For any rectangular averaging function, when we shift the window one sample to the right, we simply need to subtract the leftmost sample multiplied by *1/N*, and then add in the new sample multiplied by *1/N*

Average[1:N+1] = Average[0:N]-Average[0]/N+Average[N+1]/N

Each shift operation is just a O(1) calculation. On the other hand, for an arbitrary averaging curve we would need to recompute the element-wise products along the entire curve, for every shift.

We can combine the two advantages by approximating the averaging curve with a piecewise function of lines:



Each of these trapezoidal pieces can be shifted in O(1) time. For a trapezoid with leftmost height of *b*, and slope (over *N* samples) of *m*, the shifted trapezoid product can be calculated like so:

Trapezoid[1:N+1] = (Trapezoid[0:N] - b*Trapezoid[0])/m + (m*N+b)Trapezoid[N+1]

Another major component of the project that involved averaging was the timbre of the note. My final technique involved averaging the power of each frequency along 1000 samples in order to reduce noise and determine the dominant frequencies of a note. However, I realized that when I took the power of each frequency, I was restricting all values to be above zero. Thus, if I had noise in my frequency data where the magnitudes of oscillated between positive and negative, summing them up normaling would give an answer close to 0, but summing up their powers would give a large positive value. Thus, perhaps it would be better to average the timbres and then afterwards take the power, rather than the other way around.

In addition, I realized that if a certain frequency was actually coming from an instrument and not noise, than the phase of that signal over time can be predicted using the frequency and sampling rate. Thus, we can compare the actual signal's phase with the predicted signal's phase over time to determine whether or not a certain frequency is noise or if it's originating from an instrument.

Conclusion

One of the main things I learned from this project is the sheer difficulty of signal and frequency analysis. I clearly underestimated the scope of my original proposal, believing that it would just be a few hours of extracting notes, and then perhaps a few days of analysing instrument data. On the contrary, I did not even get to touch instrument classification, and spent countless days working full time on trying to extract a clear set of notes.

One of the most interesting things I found was the difference between how data is perceived and how it looks graphically. When listening to the sound clips or viewing the spectrogram data, it was almost painfully obvious where the notes were, what instruments were playing, etc. However, once it came to analyzing the data piece by piece, suddenly it was very hard to find well-defined boundaries to delineate the data. The biggest challenge I faced was the Gabor limit, and how to find a balance between frequency domain analysis and time domain analysis to achieve the best results. It is uncanny how simple a clip of music may sound, but once you try to analyze it, all the methods begin to fail.

However, the Gabor limit was only the beginning of my problems. I also realized that there seem to be a lot of built-in thresholds in our ears that need to be experimentally determined in order to be programmed into our analysis techniques. In addition, there are a ton of different factors in music that all work with each other, like note strength, pitch, attack, modulation, and speed, and our brain is able to quickly interpret those relationships in order to gain a better picture of the music. On the other hand, to simulate this, we would have to use complex feedback loops and double averaging thresholds like I ended up doing, seemingly much

more complicated and involved.

I tried my best to tie it into concepts we learned in class (besides the Fourier transform), but in the end I couldn't just normalize the data or examine the covariance like we had done in the homeworks. In fact, at one point I did try to use PCA[2] to see if I could find a good "best fit" timbre for a set of frequency distributions, but it was extremely un-insightful and the singular values did not follow a sharp curve like I had hoped. Conceivably, many of the statistical methods we learned in class could have been used during instrument classification or note pattern recognition, but I was so far away from either of these, still trying to extract some sort of useful data out of the chaotic frequency and time domains.

In fact, to be honest I think that by the end of the project, even after the countless hours I spent on it, I still think I have only scratched the surface of music analysis. My note extraction algorithm, while decently complicated from my standards, still does a primitive job at extracting the actual notes. There are tons of edge cases that I did not even get to touch, like strongly modulated notes (eg dubstep), separating chords, and accounting for different note "attacks" (like legato, pizzicato, etc). Overall, there were a lot of failed attempts, and a lot of room for improvement.

However, while the project was frustratingly hard, I really enjoyed it and definitely learned more than I would have ever expected. Because the data analysis wasn't simple statistics, I had to come up with my own techniques and methods, and the constant trial and error involved in experimenting with each method taught me a lot about their advantages and shortcomings. In addition, having the freedom to explore my own techniques for data analysis gave me a newfound appreciation for the difficulty of frequency analysis. Ultimately, I'm glad I attempted such an ambitious project, and I will probably continue it in the near future.

[2] In the Jupyter notebook "Experiments Part II"