IDENTIFICATION OF NOTES AND STRUCTURES IN MUSIC THROUGH FOURIER TRANSFORMS

by

Andy Romero

A Thesis Submitted to the Faculty of

The Wilkes Honors College

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Biological and Physical Sciences

with a Concentration in Physics

Wilkes Honors College of

Florida Atlantic University

Jupiter, Florida

May 2021

IDENTIFICATION OF NOTES AND STRUCTURES IN MUSIC THROUGH FOURIER TRANSFORMS

by

Andy Romero

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Yaouen Fily, and has been approved by the members of their supervisory committee. It was submitted to the faculty of the Harriet L. Wilkes Honors College and was accepted in partial fulfillment of the requirements for the degree of Bachelor of Science in Biological and Physical Sciences.

SUPERVISORY COMMITTEE:

Dr. Yaouen Fily

Dr. Warren McGovern

Dean Justin Perry, Harriet L. Wilkes Honors College

Date

Abstract

Author:	Andy Romero
Title:	Identification of Notes and Structures in Music Through Fourier Transforms
Institution:	Harriet L. Wilkes Honors College, Florida Atlantic University
Thesis Advisor:	Dr. Yaouen Fily
Degree:	Bachelor of Science in Biological and Physical Sciences
Concentration:	Physics
Year:	2021

Can we use the mathematical rules that govern waves to characterize notes and chords from a sound sample? For this project I designed a python program that uses Fourier transforms, a mathematical technique to extract the frequencies that make up a sound wave, to do just that. I will describe the method and explain how this can be used to identify musical structures in a piece of music.

Contents

List of Figures

1 Introduction

One of the strongest ideas in the arsenal of music is the concept of harmony. The concept expresses that different, disparate sounds can come together to create structure, meaning, and feeling. To a musician, understanding these structures is paramount to understanding the process of musical writing, storytelling, and replayability. Luckily, we know that music, as sound, acts as a wave, and thus exhibits the mathematical properties of waves. This tells us that music can be broken down quantitatively and analyzed at a rigorous level. What we want from an analysis of music are the specific structures that musicians require and that audiences are moved by, such as chords and scales. Thus, we can ask ourselves: Can these structures be identified analytically? Through employment of the mathematical laws which dictate how sound works, can we characterize notes and chords from audio samples?

Coming to understand these questions, and ultimately resolving them, requires a quantitative analysis of the structures of music, primarily chords and the notes that make them up. Audio is first heard or interpreted as a signal over time. This signal contains information of the loudness of the audio over time, forming the time domain. One of the main methods of analysis performed on signals in the time domain is the Fourier transform. The transform changes the signal from amplitude over time to amplitude over frequencies, the frequency domain. Basically, the Fourier transform is the idea that all continuous signals are the sum of specific sine waves, and that the graph of a Fourier transform shows the frequencies that make up a signal and their intensity in contributing to the sound. Figure 1 shows an audio signal in the time domain, as in showing the loudness of the final signal over time, and the two peaks of the frequency domain that makes it up. Not only can the Fourier transform show us information regarding the music that is being dealt with, it shows us a complicated sweep of harmonics unique to the respective instruments playing the tones.

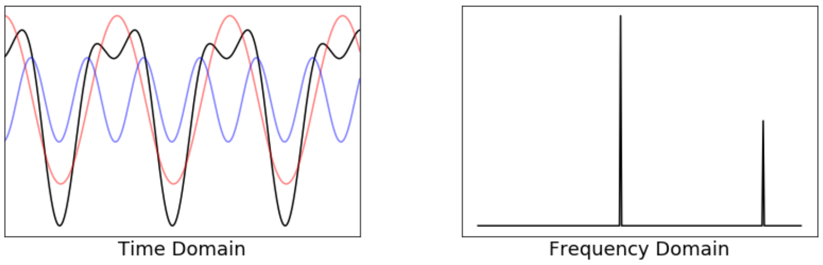

Figure 1: The Fourier transform decomposes a signal that is given as amplitude over time, the Time Domain, to a signal that shows amplitude over frequencies, the Frequency Domain. The two peaks in the frequency domain are what constitute the signal in the time domain. The left peak corresponds with the red sine, the right peak corresponds with the blue sine.

This thesis sets out to write code that successfully can take a signal, either a note or a chord, and deconstruct it into its respective frequencies and harmonics using the Fourier transform, where then the frequencies can be identified and categorized. Those frequencies can then be further categorized as chords and parts of scales. Future work in real-time applications will also be discussed. This program is written in python and uses several libraries, such as libROSA and SciPy, for audio and signal processing.

Other similar projects and libraries that process music and signals to detect pitches and higher structures already exist, each with their own angle of specialization. For example, the dedicated library aubio was created for a similar task to this project, using Fourier transforms to identify single notes and rhythm from a signal [2]. This library, along with several other independent projects that achieve the same ends through various methods, share the spirit of this project. Our angle differs from these because of what we chose to reimplement and what we chose to create ourselves. Our project specializes in the extraction of notes, chords, and scales from the Fourier transform of the signal.

The Fourier transform itself is taken from LibROSA. The note identification and categorization, chord identification, and scale identification is written from scratch.

2 Methods

2.1 Sound

Before coming to understand how to work with and analyze music, it would be helpful to have an understanding of what sound in general is. Physically, sound is an acoustic wave, a wave of compression propagating through a medium such as, but not limited to, air. Sound is created by the vibration of objects. As an object vibrates, the air around it compresses. Local pockets of compressed air, when returning to normal volume, compress a neighboring pocket of air. This motion of compression propagates through the air until it reaches the ear, where it is interpreted in the brain as sound.

These waves carry information and hold defining characteristics which are needed for analysis. The two most important and relevant defining characteristics of a sound wave for the purposes of this project, are its amplitude and frequency. The amplitude of the sound wave tells us its intensity, the strength of the signal itself. To the human ear, amplitude is loudness or volume; quieter sounds are therefore signals of lower amplitude. Then the frequency defines the period of the oscillation of the wave. It identifies how often the air oscillates back and forth with pressure as the wave propagates. This is measured in cycles per second, which is the definition of the Hz unit. Importantly, note that time is an important factor in determining the frequency of a sound wave. To the human ear, the frequency is perceived as pitch. A high frequency sound is heard as a high pitch tone. Now we need a tool for which sound can be analyzed quantitatively.

2.2 Fourier Transform

In order to work with signals of sound in a meaningful way, one must be able to break down those signals into their component parts. This is the main function of the Fourier transform, showing the composition of a signal. The Fourier transform was first employed by Jean-Baptiste Joseph Fourier in his work The Analytical Theory of Heat, where he claims that any function can be expressed as an infinite series of certain sinusoids, modulated by constants [3]. Later work by Joseph Louis Lagrange, in calculating coefficients of a trigonometric series [6], and by Carl Fredriech Gauss, in calculating the orbits of asteroids, also used early Fourier transform algorithms [5]. The application of Fourier transforms most relevant to this project, however, is harmonic analysis.

Harmonic analysis refers to the study of Fourier analysis of waves in particular. This particular discipline has applications in analyzing tides [4], tomography (imaging cross-sections of the body through x-rays or ultrasound) [1], and signal processing.

2.2.1 Noise vs. Notes

The Fourier transform gives us the range of frequencies which comprise any sound, but what is relevant to this thesis is the transform of a sound that is recognizable as a note in music. For example, the transform of the audio of a car crash would be so full of noise and frequencies that identifying a note or even a group of notes from such a structure would be a fruitless effort.

Figure 2 shows the Fourier transform of the sound of a car crashing into a stationary obstacle as part of a crash test demonstration from the Insurance Institute of Highway Safety [8], compared with the spectrum of a single note played on a piano. As we can see from the top spectrum, the loudest frequencies of the signal, around or below 1000 Hz, show behavior that is very jagged, noisy,

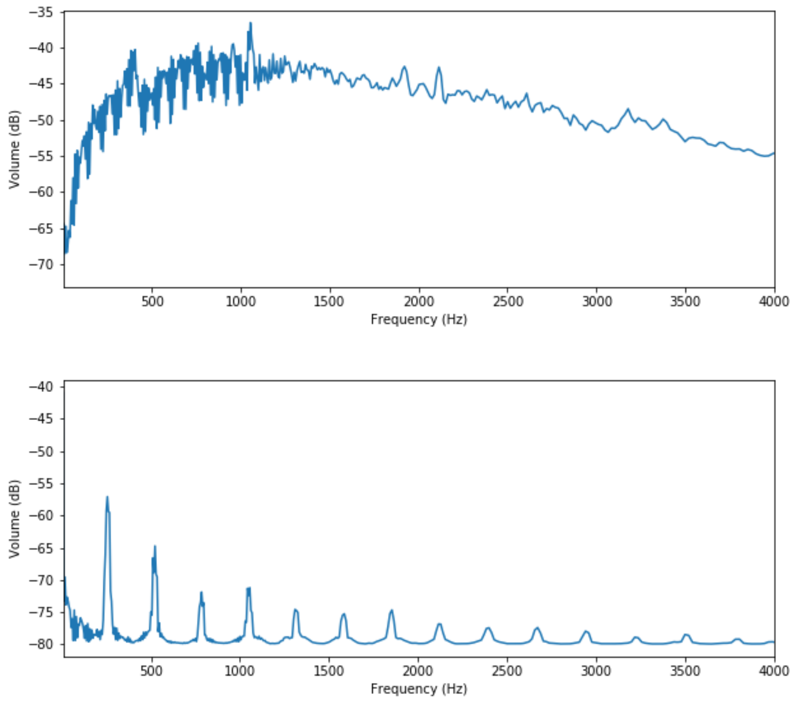

Figure 2: Two spectra. Vertical axes are volume in dB. Horizontal axes are frequencies in Hz. Top spectrum shows the Fourier transform of a car crash. Bottom spectrum shows the Fourier transform of a single note played on a piano.

and difficult to work with. Contrasting that harsh noise is the relatively minimal noise present in the bottom spectrum, which also presents clearly defined peaks.

We also see from the bottom spectrum of Figure 2 the sort of structure which we expect music to take, and that this program aims to isolate and analyze. From the defined peaks of the bottom spectrum, we can identify the first and loudest peak of the signal as the fundamental. Notice then evenly spaced smaller peaks afterward, which we define as the harmonics of that fundamental. This shows that instruments do not play perfect sinusoidal frequencies; rather they are marked by the presence and expression of harmonics. The harmonics define the signature of the instrument in each note it produces. What makes a grand piano sound like a grand piano are the harmonics that it produces. Harmonics are subtle instances of notes, quieter and higher pitched, which give the nuanced feeling of a piano. This is true for any instrument. Another factor that may define or differentiate an instrument is the quality the sound takes as it is played. For example, the loudness at the beginning of a note may be high and immediate, like a guitar, or it may be softer and develop slowly, like a flute.

2.3 Music Theory

Music theory provides the framework needed to understand sounds in a musical context and identify structures. For example, The note in music is simply a pitch with a duration. From that note, intervals may be defined which form the core of the 12-tone equal temperament tuning system. This is a tuning system which is most ubiquitous in western classical music tradition, which defines that notes can be expressed in groups of 12 equally spaced intervals called semitones. The collection of these intervals forms the octave. This pattern repeats, marking a note 12 semitones higher than the base note as one octave higher. These pitches, defined by their intervals between each other, are named by letter from A to G, including some named between letters expressed as sharp(♯). This means that all of the notes we work with are: A, A♯, B, C, C♯, D, D♯, E, F, F♯, G, and G♯, as seen in Figure 3.

Further information is given by assigning a value of which octave the note appears in, named so based on the location of that octave on the keyboard of a piano. This means that octave intervals are numbered. For example, Middle C, C4, is an important note in many musical compositions, since it is the C closest to the center of the piano, which has the 4th octave. All of these notes are named and tuned to specific pitches and frequencies, defined by a standard pitch: A4 = 440 Hz. This, though being called standard pitch, is arbitrarily

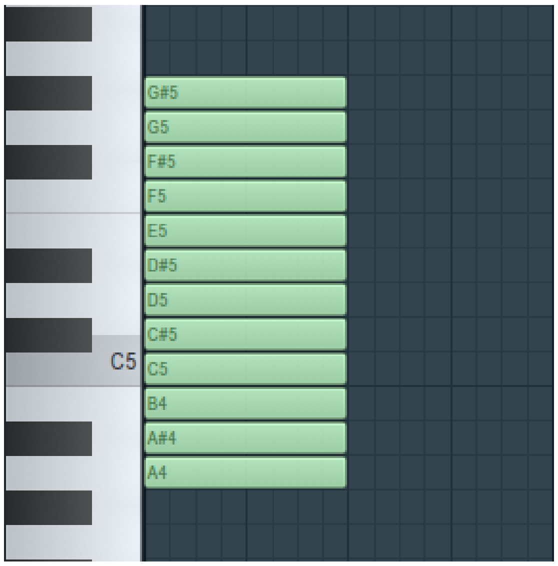

Figure 3: Piano roll showing the 12 notes in 12-tone Equal Temperament tuning. Created in FL Studio by Image Line.

chosen both by music theorists and by us for the purposes of this thesis.

Now to introduce the concept of playing several of those notes at the same time. In music, playing three or more notes at the same time is called a chord. These are some of the most basic and fundamental structures in music, and are critical to analysis of music production. The most common chords are triads, three notes of particular intervals from each other. These particular intervals give the chord a special structure, feeling, and most relevant to analysis, name. The most common triad chords are Major, Minor, Augmented, and Diminished. An example of this naming convention, along with a demonstration of the intervals that define them, can be seen with a major chord starting with C, seen in Figure 4.

A major chord has the root note followed by a major third and a perfect fifth. A major third is four semitones above the root note, in this case, E. The perfect fifth is seven semitones above the root note, in this case G. Therefore, a

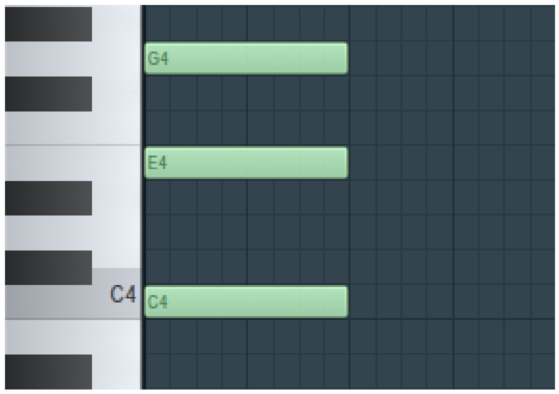

Figure 4: Piano roll showing C4, E4, G4, comprising a C Major chord. Created in FL Studio by Image Line.

C major chord consists of C, E, and G. If any of those intervals are different, it produces a different chord with a different name and structure.

From our 12 tone equal temperament system, there are pools of notes which generate chords and melodies readily. This is the scale. There are various kinds of scales, which include any number of the 12 tones in our tuning system. The most common in western music, however, are the heptatonic (seven tone) scales. Among the heptatonic scales are the Major and Minor scales. Like chords, scales are defined by the first note and the intervals above that note since. For example, a major scale is defined by the intervallic pattern of W–W–H–W–W–W–H, where W is a whole step, two semitones, and H is a half step, or one semitone.

Thus, a C Major scale has the notes: C–D–E–F–G–A–B–C, seen in Figure 5. We are only concerned with the Major and Minor scales, since they are the most ubiquitous in modern western music.

2.4 Microphones

In order to work with audio in a digital, computational sense, we must record it on microphones. The choice of microphone to be used presents the question

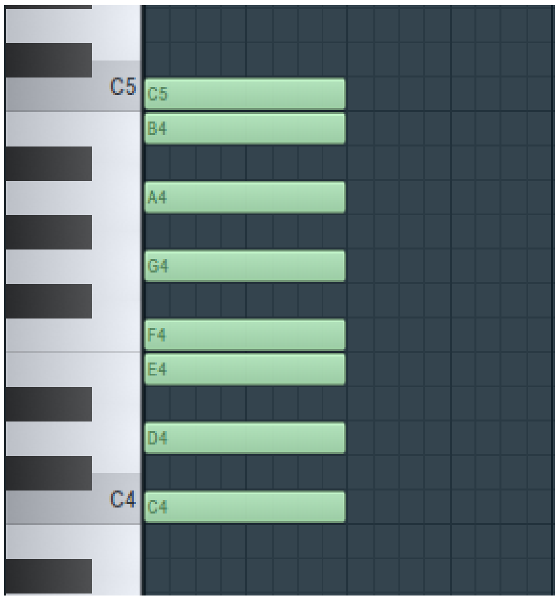

Figure 5: Piano roll showing the seven notes (eight if including the octave) that comprise the C Major scale. The notes shown are C4, D4, E4, F4, G4, A4, B4, and C5. Created in FL Studio by Image Line.

of how sensitive the microphones must be in order to still record a workable signal. To that end, several microphones were tested and used throughout the project. The clearest, best quality microphone used is the Yeti microphone by Blue. A mid-tier microphone used for comparison is the built-in microphone of the Cloud Stinger by Hyper X. The lowest quality microphone used is the built-in microphone of the Dell Inspiron 5559 laptop. The higher fidelity of the audio, the clearer the signal, the less noise takes a toll on the quality of the audio.

2.5 LibROSA

Armed with a basic explanation of music theory, we require some computational method by which we can process a sample and introduce our Fourier transform. In python, a commonly used module for audio processing is LibROSA. It can readily retrieve information from a recorded piece. From LibROSA, we used a few functions for the display and processing of signals [7].

2.5.1 librosa.load

Processing cannot begin if there is no method by which to load up an audio file. Thus, for a signal, librosa.load must be called. This function only requires a string of a file path for which to read and interpret as a signal. It returns an array of amplitudes over time, which is the signal we will perform the Fourier transform over. Another important parameter to consider for the load function is 'sr', or sampling rate. The sampling rate is a value that determines the amount of samples taken from the signal per second. If none is specified, it uses the native sampling rate of the signal.

2.5.2 librosa.feature.melspectrogram

After loading, the signal must be processed with a Fourier transform. For that, the program uses the feature.melspectrogram function. A spectrogram can be thought of as dividing an entire signal into discrete windows of time, and then performing a Fourier transform on each of those windows. These can then be represented together in two dimensions, using brightness to display the axis of loudness or volume, with the function librosa.display.specshow. Figure 6 shows such a spectrogram for a single note, middle C, played on a piano for 30 seconds. Each column along the spectrogram represents its own Fourier transform around that point of time in the signal.

From a programming perspective, "librosa.feature.melspectrogram" outputs a 2D array of amplitude values in which each column represents one of the time windows and each row represents a frequency.

The rows of the output, though they are frequencies, are measured in mels, rather than Hz. The mel scale is a log-scaled measure of pitches which are

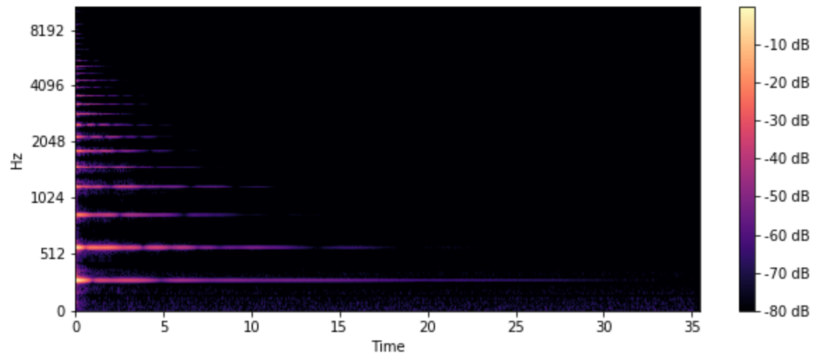

Figure 6: Spectrogram of a single note, middle C, played on a piano. Brightness represents intensity of the signal in decibels at that frequency at that time. Vertical axis is frequency in Hz, horizontal axis is time in seconds.

determined by observers to be equal in distance from each other. Mels were developed as a subjective scale in 1937, where 5 observers were selected that fractionated (categorized and divided) several tones [9]. From this data, a model was constructed which defines the conversion between Hz and mels, as seen in Figure 7. Note that according to Figure 6, display.specshow shows the vertical axis in Hz, while the data it is interpreting from melspectrogram is in mels. Specshow internally converts the mels into Hz. LibROSA also has a function which converts from mels to Hz, core.mel-frequencies. This conversion was taken from LibROSA and used throughout the project. core.mel-frequencies takes an input of the number of mels used, and a range of frequencies in Hz on which to map the mels.

The mel spectrogram takes the loaded signal as input, also needing the sampling rate, and the length of the window used to perform the Fourier transform. The latter-most of which is critical for the clarity of the processed signal. For a true Fourier transform, the entire signal must be taken across all time. In order to show the frequencies present per unit of time, windows of time must be chosen to perform the Fourier transform on. The longer the window, the

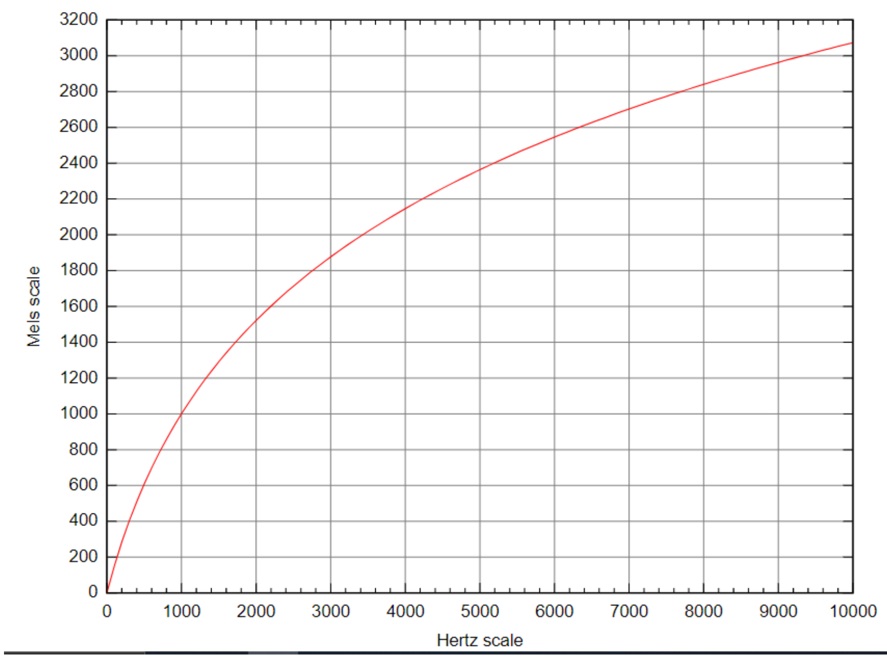

Figure 7: Shows the conversion between the Mel scale and the Hz scale. Vertical axis is mels, horizontal axis is hz. Note the logarithmic line that defines the conversion.

clearer the Fourier transform is in terms of showing the frequency, but the more information is lost about notes over time, such as when notes end or change. If we played two distinct notes, first one, then the other after some time, the complete Fourier transform of the signal would show both of the notes, but not when they occurred. With the same example of the two notes, with a short window we would be clearly seeing when the notes changed, but would be much less certain as to what their frequencies are. This is a form of the uncertainty principle. It is why keeping a consistent time window in melspectrogram is needed.

2.6 Peak Detection

In order to identify the notes being played in a sound sample, the peaks of the spectrum must be located and their frequencies compared. We do this with scipy's "find peaks" function [10]. It takes a list of data and finds local maxima by comparing neighboring values. The function with default parameters is very sensitive, and can collect many peaks that are not necessary, see Figure 8. For this, the function allows for various parameters that can refine the conditions for detecting peaks. We tweaked the following parameters arbitrarily to values that produced the seemingly most "clean" peak detections: height, which requires the peaks to meet a minimum value, threshold, which requires peaks to have some minimum vertical distance from its neighbors, and width, which requires the values around a potential peak to have a certain spread.

3 Results

The signals required for processing were recorded on microphones or produced with MIDI, or Musical Instrument Digital Interface. MIDI is a process of synthesizing audio and music with computers. MIDI data carries information of timing, loudness, and the data that defines the instrument. Instruments that produce MIDI such as MIDI keyboards or controllers use synthesized sounds created by oscillators, the same process which is used in an electronic synthesizer. The majority of the signals used in this thesis were synthesized in MIDI. The rest were recorded by microphone, as discussed in the methods. With recorded and produced signals, the first step in processing is being able to project signal into a spectrogram, with values showing the frequencies present over time and how strongly those frequencies are represented, as discussed in the methods.

3.1 Line of Spectrogram

To begin work processing the spectrogram, it was represented in a line by plotting the average value of each mel. First, the mels were converted to Hz using lr.core.mel-frequencies.

Figure 8 shows the line of the same signal from Figure 6 with peaks detected with orange dots. This line would make it easier to find the peaks of the signal, and thus to identify the notes. As we can see from Figure 8, many of the peaks are not needed, since that far into the frequencies, the peak detection algorithm struggles to detect the notes, and it is unclear what even constitutes those peaks. This presented the first major hurdle of the process, where the noisiness of the signal translated to noisiness in the line for peak detection.

The problem resulted in many false peaks detected, such as the two last dots on the right in Figure 8, or the slight peaks which did not have peaks detected to the left of those two. These problems were foreseen in the methods section.

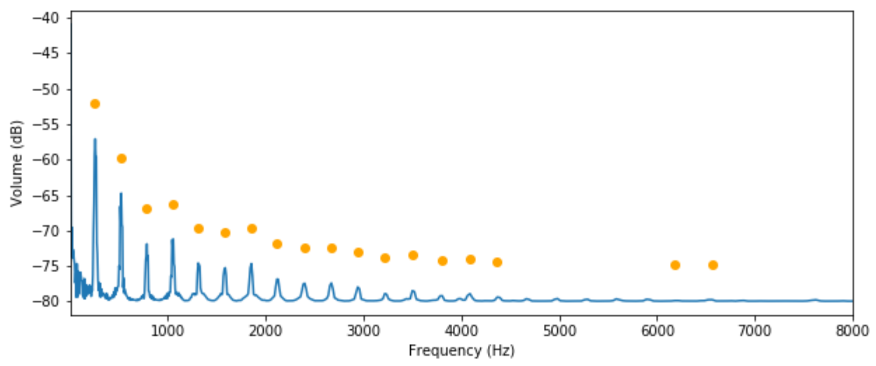

Figure 8: Average spectrum of a signal, with the peak detection algorithm plotting the perceived peaks with orange dots. The vertical axis is the amplitude in decibels. The horizontal axis are the frequencies in Hz.

They are mostly attributable either to the noisy signals themselves, the Fourier transform window of the spectrogram itself, the way that the average value of the mel bins was taken, or the way that the peaks were interpreted. The resolution to it was to adjust the peak detection parameters, as mentioned in the methods section, as well as adjusting the number of mel bins used in the spectrogram for maximal clarity. We arrived at an arbitrary value of 520 mels, as this was the least amount of mels found which preserved information and harmonic structure in the signal. The amount of mel bins taken in the mel spectrogram gives us the resolution by which frequencies are assigned. Consider Figure 9. If we were to choose an arbitrarily low number of mels, 50 as in the figure example, it would return that the frequencies of the spectrogram were in one of 50 possible bins. An arbitrarily high amount of mels, 1000, as in the bottom spectrum of Figure 9, conversely, produces jagged spectra and/or empty bins, which are difficult to negotiate with the peak detection algorithm. Either too few or too many both affect the peak detection, and thus the frequency detection. 520 mels was arrived at by tweaking to maintain accuracy while minimizing noise.

Since the output of the melspectrogram is in mels, we needed to write or re-

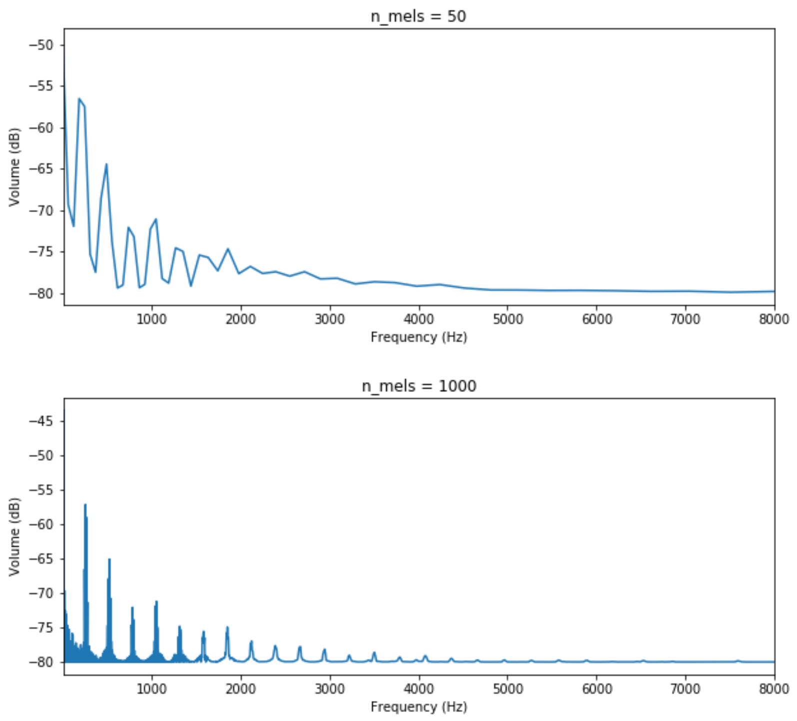

Figure 9: Two spectra, showing the same signal plotted with different number of mel bins. Top spectrum is 50 mels, bottom specturm is 1000 mels.The vertical axes is the amplitude in decibels. The horizontal axes are the frequencies in Hz.

implement a function that can convert from mel bins to Hz. We used a function from LibRosa, core.mel-frequencies. This function was able to convert the mel bins used into their respective Hz frequencies. After the frequencies are found, naming them according to their note and octave forms the core of the algorithm we implemented. The first step was to set the frequency of C0 = 16.35 Hz (in the scheme of A4 = 440 Hz) as the low baseline, and to identify the number of semitones away from C0 as the number which defines the name of the note. The number of semitones is arrived at by taking the ratio of the identified frequency and the base frequency, C0, and taking the logarithm of that ratio. Multiplying that value by 12 gives us the amount of semitones a given frequency is above C0.

The floor division of the number of semitones by 12 would return the octave that note is in. Then, the number of semitones modulo 12 would correspond to the name of a note, from C, C♯, . . . , to B. This method delivered effectively and was able to identify several test signals correctly, even in different tuning schemes such as A4 = 432 Hz and others. Different tuning schemes can have varying definitions for tones that may sound distinct to the ear, but are mathematically rather similar. For example, C4, middle C, can be anything between C4 = 256.87 and C4 = 265.19. These are audibly different notes, but the rounding in the method of note identification considers them both C4 accurately.

3.2 Categorization of Notes

Now that the core identification has been laid out, the code must be able to identify several notes in the same signal that each have their own harmonic profiles. This accounts for signals which have several instruments playing at once. To that effect, notes and their harmonics would need to be categorized together. We went about this by going through the average spectrum of the signal, peak by peak, and grouping the peak with its respective multiples in log scale. If one note is C3, you may expect its harmonics to be C4, C5, etc. An example of the process is given in Figure 10. There, the first fundamental, C4, is seen as the first peak on the left, with some of it harmonics, octaves higher, labeled in red. The other fundamental is the second peak, F4, with its harmonics in green.

We first identify the fundamental that we want to work with by finding the highest decibel, or loudest, frequency from the elements of the list that have not been assigned to a fundamental already. From here, we calculate the multiples of the frequency of that fundamental, since those are supposed to be its harmonics. The program then sweeps through all the remaining frequencies and

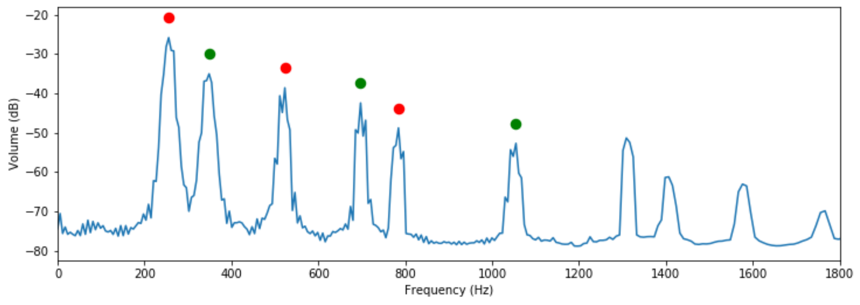

Figure 10: Spectrum of two notes being played together, C4 and F4. The vertical axis is the amplitude in decibels. The horizontal axis are the frequencies in Hz.

best fits them to the multiple of the fundamental. If the frequency is within a certain threshold, it is identified as a harmonic of that frequency and labeled accordingly. It repeats this process, identifying the next fundamental from the pool of remaining frequencies.

One issue that may come up from this approach is that the loop may mis-ascribe a frequency or label a frequency multiple times over. If a harmonic is sufficiently close to theoretical multiples of two distinct fundamentals, it might be registered as a harmonic of the first fundamental when it actually is more accurately a harmonic of the second. The program could also label a frequency twice. This is why the method of picking harmonics by meeting a threshold of octave multiples of a note is what we chose to implement in this thesis. This prevents it from mis-ascribing harmonics because before, the program would just search for the nearest peak to that multiple, without considering how near it actually is to that peak.

3.3 Naming of Chords and Scales

The next logical step, after assuring that notes are not only identified, but categorized as to correct for the harmonic profiles that define them, is to identify multiple notes as part of a chord, and to name that chord. There is no theoretical limit to how many notes may make up a chord, so the program is limited to identifying a select few chords. Within that, we should be able to differentiate between major, major 7th, augmented, minor, minor major 7th, and diminished chords. With these limitations, it is reasonable to construct a list of all possible chords of those forms and to check if the notes identified from the signal are in that list.

Rather than a list, this method can be implemented as a dictionary of the six chord types being used, with the keys of the dictionary being the types and the respective entries are the structure of that chord type in integer notation. For example, a major chord would be written as [0,4,7], where 0 is the root, 4 semitones above which is the major third, and 7 semitones above is the perfect fifth. Then, a loop can be written which cycles through all 12 tones in an octave and populates a second dictionary with all the select chords of that tone. For example, if C is the first note in our list of tones, the first pass will populate a dictionary of C major , C major 7th, C augmented and so on, and then repeat the process for C♯, D, D♯, etc.

The program, with a constructed list of 72 chords (being 6 chord types * 12 semitones), can quickly check and see if the interpreted notes form a valid chord. Simply form a string of the interpreted notes, omitting the octave, and checking if that string exists in the dictionary of chords. An extra piece of information that may be useful at the print statement would be the octave the chord is in, since just knowing that a chord is C major is insufficient for a musician trying to replicate the sound from the data. Knowing that a chord is a 4th octave C major is sufficient information to know exactly what a chord is and may sound like. There is also the consideration of inverted chords, which are chords for which one of the notes besides the root note is played down an octave. This however would increase the complexity of the search and require a much more complex chord naming system. Instead, we choose to ignore both the octave each note is in and the order of the notes in the chord. This is done by storing each chord as an unordered set of octave-less notes. An example of which is shown in Figure 11. If the program identifies [E,C,G] as the notes in the signal, in ascending order of frequency, then the chord is [C, E, G] = C Major.

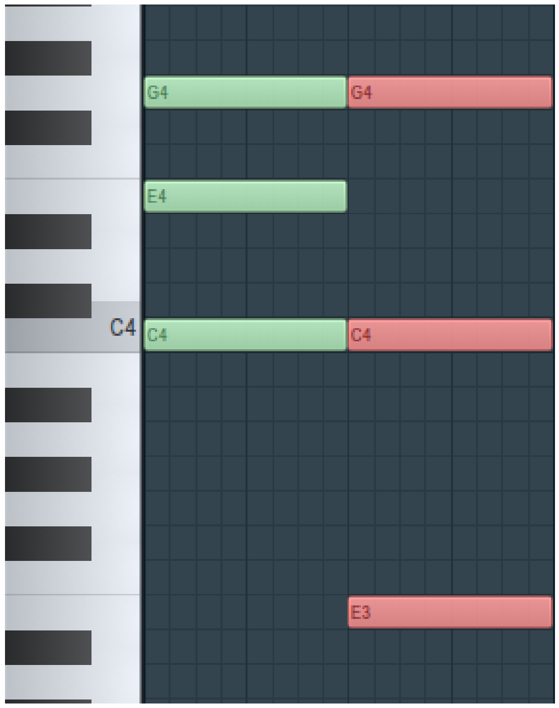

Figure 11: Piano roll showing a C Major Chord, in green, on left, and an inverted C/E chord, in red, on right. Note how in C/E, the major third, E, is played an octave down. Both are C chords. Created in FL Studio by Image Line.

The program is then successful in identifying notes from a signal and interpreting the notes as a chord. The next step is to then interpret several chords distinctly, and understand them together as the scale of the song or piece. Addressing the naming of scales raises an important question in the chord naming process, as a scale is made up of several chords. The program, however, is designed to loop over all available fundamentals of a signal across the entire time and interpret the result as one chord. This raises complications if one tries to process a signal with multiple chords playing at different points of time. One way to address this is to change the way that the signal is interpreted into slices of time, to be able to interpret chords when they are played.

The process of naming scales is relatively similar to the chord technique, with an added caveat. The types of scales are listed as a dictionary, with the semitone numbering of the notes present in the scale in integer notation. Earlier, the example of a major chord being [0,4,7] was used. In this case, a major scale would be [0,2,4,5,7,9,11]. This is done for major and natural minor, the two common and popular scales. Then, similar to the chords, a new dictionary is populated with major and minor scales for all 12 semitones.

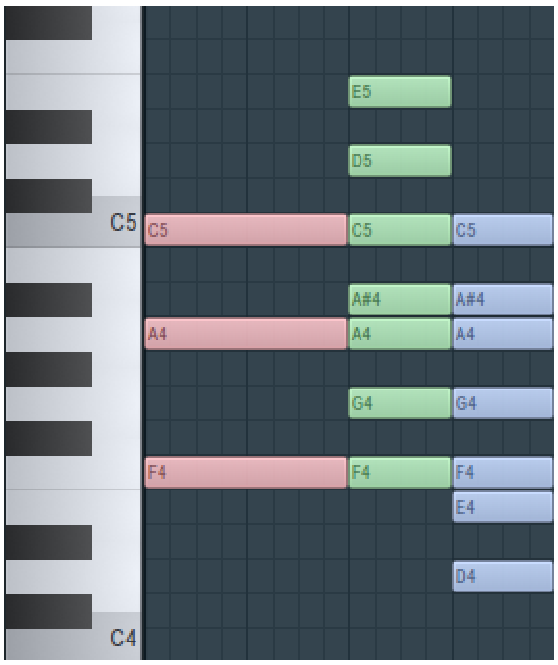

Figure 12: Piano roll of an F Major chord, in red, an F Major scale, in green, and a D Minor scale, in blue. Note how F, A, and C are notes present in both scales, thus F Major is part of both scales. Created in FL Studio by Image Line.

The caveat being that chords overlap between several scales. An F major chord can be part of an F Major scale as well as it is a part of a D Minor scale, see Figure 12. Thus the task is not as simple as checking to see if the chord is somewhere in the list. For that reason, it is sufficient for the program to provide a list of potential matched scales associated with the notes interpreted from the piece. This is done by seeing if the list of interpreted notes is a subset of any scale. If the notes are a subset, then that scale is a potential match. It is clear to see that the list of matches is only reduced with added information. As discussed, a single chord will return many scales, but a progression of 3-4 chords will return only a few matches.

It is important to note that the scale naming process is independent of the chord process. Indeed, all the program needs to identify a scale is a list of notes. Thus, both techniques can deliver information in tandem, without being conflated.

4 Outlook

4.1 The State of the Project

At the time of ending work on this thesis, it can successfully take a signal containing music and return the notes present, if a chord is present, and a list of scales the signal may be a part of. It does have substantial limitations, which may or may not be addressable with more time:

4.1.1 Chord and Scale Limits

Due to the large amount of possible chords one could construct from 3 or more notes, we had to limit our selection of chords and scales to some that are relatively few out of the respective lists of chords and scales available. For example, this project only connected notes to Major or Minor scales, only two of the heptatonic (seven note) scales. There are myriad more scales even just among the heptatonics. Then there are myriad more among chromatic (12 note) scales, hexatonic (six note), etc. Then, even if granting an expansion of the list of available chords and scales, we run into the issue of how the program should account for inverted chords.

At present state, the program can handle inverted chords, where one note in the chord is played an octave down, by not caring for the order in which the chord is played. If the notes interpreted are, in this order: E, C, G, then the program tells us it is a C Major. A musician would rather call this chord C/E, indicating that the third of the chord has been played an octave down, or has been inverted. This is a subtle point, but carries contextual information that can be lost if one only cares about the absolute notes in a chord, and not their placement. If a musician in learning relied on this program to identify the chords in a song, and those chords are to be inverted, then the musician would not know. However, this presents a problem. To account for all chord inversions, voicing, and styles, the program will require a more robust method of interpreting chords after the notes have been identified.

4.1.2 Sensitivity to Noise

The program is exceptionally sensitive to noise. In live recorded samples, even the distant rumbling of traffic or the noise of an A/C unit are enough to trip up the peak detection algorithm and send false peaks through. These false peaks can be manually thrown out by inspection, or systematically removed by continuing to tweak the parameters of the peak detection function, but no universal solution exists for the problem as it is right now. Potentially, a reconsideration of the peak detection algorithm altogether, perhaps with less arbitrary parameters, or with a different function entirely, may be what the program needs to maintain clarity in its signal. Ultimately, there is a reason recording booths are sound-proofed.

4.2 Future Direction

Based on the current limitations that exist within the program, there are some interesting directions that come to mind in terms of future applications. Aspects of this program seem fit enough to form the core of more complicated projects. The simplest of the expansion projects would be an application that listens to live audio and can identify and store information on the structures of the music in real time. At present state, the audio must be saved as a sound file and stored in a specific folder, where it is then loaded into a python interface for processing. Such a slow process would be of little use to a musician wanting to identify notes and chords in a timely fashion. Thus, the next steps for this program are to find ways of optimizing the code and getting it to a state that can input audio, process it, and tell us its musical structures in real time. This would be an ambitious extension of the project, but it keeps to the spirit of identifying the structures of music.

A Appendix

Listing 1: Identifying Single Note, Middle C

```python
import librosa as lr
import librosa.display
import numpy as np
import matplotlib.pyplot as plt
import scipy as sci
import math

# Loads the data.
data_dir = 'C:/WinPython/WPy64-3740/notebooks/docs/thesis/samples/c3
    .ogg'
y, sr = lr.load(data_dir)

# Defines the number of mels to be used throughout the project when
    calculating the mel spectrogram.
n_mels = 520

# Defines the maximum frequency that the program cares about. 8,000
    Hz is very high pitched. Frequencies beyond this are not useful.
fmax= 8000

# Defines a list of names of the 12 notes in 12 Tone Equal
    Temperament Tuning. Used in note naming process.
names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#",
     "B"]

# Creates a list of frequencies based on the amount of mels used and
     the boundaries of frequencies used.
freqs = lr.core.mel_frequencies(n_mels=n_mels, fmin=0, fmax=fmax)

# Function which takes a frequency in hertz and returns how many
    semi-tones it is over C0 = 16.35 Hz.
def semitones(hz):
    log = math.log2((hz / 16.35))
    semitones = 12 * log
    return round(semitones)

```

```
# Function which takes an amount of semi-tones and returns a note
    with octave. Calculates the semi-tones modulo 12 to get the
    letter. Floor divides by 12 to get the octave.
def notes(x):
    r = round(x)
    note = r % 12
    octave = r // 12
    return names[note]+str(octave)

# Computes the mel spectrogram of the loaded signal.
S = lr.feature.melspectrogram(y, sr=sr, n_mels=n_mels, fmax=fmax)
# Converts the power spectrogram from amplitude squared to Decibels
    dB.
log_S = lr.power_to_db(S, ref=np.max)
plt.figure(figsize=(10,4))
lr.display.specshow(log_S, sr=sr, x_axis='time', y_axis='mel')
plt.colorbar(format='%+02.0f dB')
plt.show()

# Averages the value of each mel over the length of the signal.
    Creates the line of the Fourier transform spectrum.
freq_bins = np.array([np.sum(log_S[x][0:len(log_S[x])-1])/len(log_S[
    x]) for x in range(len(log_S))])

# SciPy function which detects peaks from an input array. Note the
    parameters. These are set arbitrarily.
peaks, _ = sci.signal.find_peaks(freq_bins, width=2, distance=7,
    threshold=0.6)

plt.figure(figsize=(10,4))

plt.plot(freqs, freq_bins)
plt.xlim(5,8000)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Volume (dB)')
plt.plot(freqs[peaks], freq_bins[peaks]+5, '.', mfc='orange', mec='
    orange', mew=2, ms=10)
plt.show()

# Creates an array of the interpreted peaks containing four entries:
     Frequency, then Amplitude in dB, then which fundamental that
    peak belongs to, then which harmonic it is of the fundamental it
     is a part of.
peak_list = np.array([ [freqs[i],freq_bins[i], np.nan, np.nan] for i
     in peaks])

fund_label = 0
# Considers the loudest peak to be the first fundamental.
fund       = peak_list[np.argmax(peak_list[:,1]),0]
fundamentals = []

# While loop which loops over the list of peaks so long as there are
     peaks which are unassigned to any fundamental. Considers
    multiples of the fundamental, then checks if any peak is within
    a range of those multiples. If they are, they are considered a
    harmonic of that fundamental. Continues until all peaks are
    identified.
while np.any(np.isnan(peak_list[:,2])):
    I        = np.nonzero(np.isnan(peak_list[:,2]))[0]
    fund     = peak_list[I[np.argmax(peak_list[I,1])],0]
    mult_max = int(np.ceil(np.max(peak_list[:,0])/fund))
    for j in range(1,mult_max+1):
        d  = np.abs(peak_list[I,0]-fund*j)
        i  = np.argmin(d)
        f1 = max(fund*(j-1/2),fund*j/2**(1/24))
        f2 = min(fund*(j+1/2),fund*j*2**(1/24))
        if f1<peak_list[I[i],0]<f2:
            peak_list[I[i],2] = fund_label
            peak_list[I[i],3] = j
    fundamentals.append(fund)
    fund_label += 1

fundamentals = np.array(fundamentals)
# Sorts the list of fundamental frequencies by ascending order of Hz
    .
fundamentals.sort()
# Converts the frequencies into a name letter and octave.
fund_names = [notes(semitones(f)) for f in fundamentals]

print('fundamentals:', ','.join(fund_names))
```

Listing 2: Identifying Two Notes Played at Once, Middle C and Middle F

```
import librosa as lr
import librosa.display
import numpy as np
import matplotlib.pyplot as plt
import scipy as sci
import math

# Loads the data.
data_dir = 'C:/WinPython/WPy64-3740/notebooks/docs/thesis/samples/
    c4_and_f4.ogg'
y, sr = lr.load(data_dir)

# Defines the number of mels to be used throughout the project when
    calculating the mel spectrogram.
n_mels = 520

# Defines the maximum frequency that the program cares about. 8,000
    Hz is very high pitched. Frequencies beyond this are not useful.
fmax= 8000

# Defines a list of names of the 12 notes in 12 Tone Equal
    Temperament Tuning. Used in note naming process.
names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#",
     "B"]

# Defines a dictionary of types of chords. Used to populate the
    dictionary of all chords used later. Numbers are intervals
    between root note and parts of the chord. For example, 0 is the
    root, 4 is the major third, 7 is the perfect fifth. That makes a
     Major chord.
chord_types = { 'Major':[0,4,7], 'Major 7':[0,4,7,11], 'Augmented'
    :[0,4,8], 'Minor':[0,3,7], 'Minor Major 7':[0,3,7,11], '
    Diminished':[0,3,6] }

# Defines a dictionary of two types of scales, Major and Minor. Used
     to populate the dictionary of all scales used later. Numbers
    are intervals between root note and parts of the scale.
scale_types = { 'Major':[0,2,4,5,7,9,11], 'Minor':[0,2,3,5,7,8,10]}

```

```
# Creates a list of frequencies based on the amount of mels used and
    the boundaries of frequencies used.
freqs = lr.core.mel_frequencies(n_mels=n_mels, fmin=0, fmax=fmax)

# Function which takes a frequency in hertz and returns how many
    semi-tones it is over C0 = 16.35 Hz.
def semitones(hz):
    log = math.log2((hz / 16.35))
    semitones = 12 *log
    return round(semitones)

# Function which takes an amount of semi-tones and returns a note
    with octave. Calculates the semi-tones modulo 12 to get the
    letter. Floor divides by 12 to get the octave.
def notes(x):
    r = round(x)
    note = r % 12
    octave = r // 12
    return names[note]+str(octave)

# Computes the mel spectrogram of the loaded signal.
S = lr.feature.melspectrogram(y, sr=sr, n_mels=n_mels, fmax=fmax)
# Converts the power spectrogram from amplitude squared to Decibels
    dB.
log_S = lr.power_to_db(S, ref=np.max)

plt.figure(figsize=(18,6))
lr.display.specshow(log_S, sr=sr, x_axis='time', y_axis='mel')
plt.colorbar(format='%+03.0f dB')
plt.show()

# Averages the value of each mel over the length of the signal.
    Creates the line of the Fourier transform spectrum.
freq_bins = np.array([np.sum(log_S[x][0:len(log_S[x])-1])/len(log_S[
    x]) for x in range(len(log_S))])

# SciPy function which detects peaks from an input array. Note the
    parameters. These are set arbitrarily.
peaks, _ = sci.signal.find_peaks(freq_bins, width=5, distance=7,
    threshold=0.1)

```

```python
plt.figure(figsize=(20,4))
plt.plot(freq_bins)
plt.plot(peaks, freq_bins[peaks], 'o')
plt.show()

# Creates an array of the interpreted peaks containing four entries: Frequency, then Amplitude in dB, then which fundamental that peak belongs to, then which harmonic it is of the fundamental it is a part of.
peak_list = np.array([ [freqs[i],freq_bins[i], np.nan, np.nan] for i in peaks])

fund_label = 0
# Considers the loudest peak to be the first fundamental.
fund       = peak_list[np.argmax(peak_list[:,1]) ,0]
fundamentals = []

# While loop which loops over the list of peaks so long as there are peaks which are unassigned to any fundamental. Considers multiples of the fundamental, then checks if any peak is within a range of those multiples. If they are, they are considered a harmonic of that fundamental. Continues until all peaks are identified.
while np.any(np.isnan(peak_list[:,2])):
    I        = np.nonzero(np.isnan(peak_list[:,2]))[0]
    fund     = peak_list[I[np.argmax(peak_list[I,1])],0]
    mult_max = int(np.ceil(np.max(peak_list[:,0])/fund))
    for j in range(1,mult_max+1):
        d  = np.abs(peak_list[I,0]-fund*j)
        i  = np.argmin(d)
        f1 = max(fund*(j-1/2),fund*j/2**(1/24))
        f2 = min(fund*(j+1/2),fund*j*2**(1/24))
        if f1<peak_list[I[i],0]<f2:
            peak_list[I[i],2] = fund_label
            peak_list[I[i],3] = j
    fundamentals.append(fund)
    fund_label += 1

fundamentals = np.array(fundamentals)
# Sorts the list of fundamental frequencies by ascending order of Hz.
```

```python
fundamentals.sort()
# Converts the frequencies into a name letter and octave.
fund_names = [notes(semitones(f)) for f in fundamentals]
# Removes the octave number. Needed for chord and scale
    identification.
no_octave = set([fund_names[i][:1] for i in range(len(fund_names))])

print('fundamentals:', ','.join(fund_names))

# Populates a dictionary of chords based on chord_types. Creates the
     chords by indexing the list of names with the intervals in
    chord_types. Then repeats the process for all 12 notes with all
    6 chord types.
chords = {}
for n in range(12):
    for k in chord_types.keys():
        chords[names[n]+' '+ k] = set([names[(i+n)%12] for i in
    chord_types[k]])

# Checks if the list of fundamental letters matches an entry in the
    dictionary of chords.
for k in chords.keys():
    if no_octave == chords[k]:
        print(k, chords[k])

# Populates a dictionary of scales based on scale_types. Creates the
     scales by indexing the list of names with the intervals in
    scale_types. Then repeats the process for all 12 notes with both
     scale types.
scales = {}
for n in range(12):
    for k in scale_types.keys():
        scales[names[n]+' '+ k] = set([names[(i+n)%12] for i in
    scale_types[k]])

# Checks if the list of fundamental letters is a subset of an entry
    in the dictionary of scales.
for k in scales.keys():
    if no_octave.issubset(scales[k]):
        print(k + ' is a match')
```

Listing 3: Identifying Notes, Chords, and Scales of an F Major Chord

```
import librosa as lr
import librosa.display
import numpy as np
import matplotlib.pyplot as plt
import scipy as sci
import math
from matplotlib.patches import Rectangle

# Loads the data.
data_dir = 'C:/WinPython/WPy64-3740/notebooks/docs/thesis/samples/
    f4maj.ogg'
y, sr = lr.load(data_dir)

# Defines the number of mels to be used throughout the project when
    calculating the mel spectrogram.
n_mels = 520

# Defines the maximum frequency that the program cares about. 8,000
    Hz is very high pitched. Frequencies beyond this are not useful.
fmax= 8000

# Defines a list of names of the 12 notes in 12 Tone Equal
    Temperament Tuning. Used in note naming process.
names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#",
     "B"]

# Defines a dictionary of types of chords. Used to populate the
    dictionary of all chords used later. Numbers are intervals
    between root note and parts of the chord. For example, 0 is the
    root, 4 is the major third, 7 is the perfect fifth. That makes a
     Major chord.
chord_types = { 'Major':[0,4,7], 'Major 7':[0,4,7,11], 'Augmented'
    :[0,4,8], 'Minor':[0,3,7], 'Minor Major 7':[0,3,7,11], '
    Diminished':[0,3,6] }

# Defines a dictionary of two types of scales, Major and Minor. Used
     to populate the dictionary of all scales used later. Numbers
    are intervals between root note and parts of the scale.
scale_types = { 'Major':[0,2,4,5,7,9,11], 'Minor':[0,2,3,5,7,8,10]}
```

```

# Creates a list of frequencies based on the amount of mels used and
    the boundaries of frequencies used.
freqs = lr.core.mel_frequencies(n_mels=n_mels, fmin=0, fmax=fmax)

# Function which takes a frequency in hertz and returns how many
    semi-tones it is over C0 = 16.35 Hz.
def semitones(hz):
    log = math.log2((hz / 16.35))
    semitones = 12 *log
    return round(semitones)

# Function which takes an amount of semi-tones and returns a note
    with octave. Calculates the semi-tones modulo 12 to get the
    letter. Floor divides by 12 to get the octave.
def notes(x):
    r = round(x)
    note = r % 12
    octave = r // 12
    return names[note]+str(octave)

# Computes the mel spectrogram of the loaded signal.
S = lr.feature.melspectrogram(y, sr=sr, n_mels=n_mels, fmax=fmax)
# Converts the power spectrogram from amplitude squared to Decibels
    dB.
log_S = lr.power_to_db(S, ref=np.max)
plt.figure(figsize=(18,6))
lr.display.specshow(log_S, sr=sr, x_axis='time', y_axis='mel')
plt.colorbar(format='%+03.0f dB')
plt.show()

# Averages the value of each mel over the length of the signal.
    Creates the line of the Fourier transform spectrum.
freq_bins = np.array([np.sum(log_S[x][0:len(log_S[x])-1])/len(log_S[
    x]) for x in range(len(log_S))])

# SciPy function which detects peaks from an input array. Note the
    parameters. These are set arbitrarily.
peaks, _ = sci.signal.find_peaks(freq_bins, width=5, distance=7,
    threshold=0.1)

```

```
plt.figure(figsize=(30,4))
plt.plot(freq_bins)
plt.plot(peaks, freq_bins[peaks], 'o')
plt.show()

# Creates an array of the interpreted peaks containing four entries:
     Frequency, then Amplitude in dB, then which fundamental that
    peak belongs to, then which harmonic it is of the fundamental it
     is a part of.
peak_list = np.array([ [freqs[i],freq_bins[i], np.nan, np.nan] for i
     in peaks])

fund_label = 0
# Considers the loudest peak to be the first fundamental.
fund       = peak_list[np.argmax(peak_list[:,1]) ,0]
fundamentals = []

# While loop which loops over the list of peaks so long as there are
     peaks which are unassigned to any fundamental. Considers
    multiples of the fundamental, then checks if any peak is within
    a range of those multiples. If they are, they are considered a
    harmonic of that fundamental. Continues until all peaks are
    identified.
while np.any(np.isnan(peak_list[:,2])):
    I        = np.nonzero(np.isnan(peak_list[:,2]))[0]
    fund     = peak_list[I[np.argmax(peak_list[I,1])],0]
    mult_max = int(np.ceil(np.max(peak_list[:,0])/fund))
    for j in range(1,mult_max+1):
        d  = np.abs(peak_list[I,0]-fund*j)
        i  = np.argmin(d)
        f1 = max(fund*(j-1/2),fund*j/2**(1/24))
        f2 = min(fund*(j+1/2),fund*j*2**(1/24))
        if f1<peak_list[I[i],0]<f2:
            peak_list[I[i],2] = fund_label
            peak_list[I[i],3] = j
    fundamentals.append(fund)
    fund_label += 1

fundamentals = np.array(fundamentals)
# Sorts the list of fundamental frequencies by ascending order of Hz
    .
```

```
fundamentals.sort()
# Converts the frequencies into a name letter and octave.
fund_names = [notes(semitones(f)) for f in fundamentals]
# Removes the octave number. Needed for chord and scale
    identification.
no_octave = set([fund_names[i][:1] for i in range(len(fund_names))])

print('fundamentals:', ','.join(fund_names))

# Populates a dictionary of chords based on chord_types. Creates the
     chords by indexing the list of names with the intervals in
    chord_types. Then repeats the process for all 12 notes with all
    6 chord types.
chords = {}
for n in range(12):
    for k in chord_types.keys():
        chords[names[n]+' '+ k] = set([names[(i+n)%12] for i in
    chord_types[k]])

# Checks if the list of fundamental letters matches an entry in the
    dictionary of chords.
for k in chords.keys():
    if no_octave == chords[k]:
        print(k, chords[k])

# Populates a dictionary of scales based on scale_types. Creates the
     scales by indexing the list of names with the intervals in
    scale_types. Then repeats the process for all 12 notes with both
     scale types.
scales = {}
for n in range(12):
    for k in scale_types.keys():
        scales[names[n]+' '+ k] = set([names[(i+n)%12] for i in
    scale_types[k]])

# Checks if the list of fundamental letters is a subset of an entry
    in the dictionary of scales.
for k in scales.keys():
    if no_octave.issubset(scales[k]):
        print(k + ' is a match')

```

```

# Redefines peaks so that it finds peaks in each time window of the
    spectrogram.
peaks = np.array([ sci.signal.find_peaks(s, width=5, distance=7,
    threshold=0.1)[0] for s in log_S.T ])

# Defines a crude piano roll which is to visualize the peaks
    detected over time. Takes the respective peak as a number of
    semitones above C0 and plots it as a rectangular unit on a
    labeled grid.
def piano_roll(peaks):
    sample = []
    for i in range(len(peaks)):
        sample.append([semitones(freqs[peaks[i][j]]) for j in range(
    len(peaks[i]))])


    plt.figure(figsize=(12,8))
    ax = plt.gca()

    for i,notes in enumerate(sample):
        for note in notes:
            r = Rectangle((i,note),1,1)
            ax.add_patch(r)

    ax.set_yticks([12*i+0.5 for i in range(13)])
    ax.set_yticklabels([f'C{i}' for i in range(13)])
    ax.set_xlim(0,len(sample))
    ax.set_ylim(0,96+1)

    for y in np.arange(*ax.get_ybound(),1):
        ax.axhline(y,color=(0.5,)*3,lw=0.5)

    plt.show()

piano_roll(peaks)
```

Listing 4: Comparison of Peaks Over Time in A Piano Roll of the Same Signal Recorded With Three Microphones

```
import librosa as lr
import librosa.display
import numpy as np
import matplotlib.pyplot as plt
import scipy as sci
import math
from matplotlib.patches import Rectangle


# Loads the data.
data_dir_high = 'C:/WinPython/WPy64-3740/notebooks/docs/thesis/
    samples/high_noise.ogg'
y_high, sr_high = lr.load(data_dir_high)

data_dir_mid = 'C:/WinPython/WPy64-3740/notebooks/docs/thesis/
    samples/mid_noise.ogg'
y_mid, sr_mid = lr.load(data_dir_mid)

data_dir_low = 'C:/WinPython/WPy64-3740/notebooks/docs/thesis/
    samples/low_noise.ogg'
y_low, sr_low = lr.load(data_dir_low)

# Defines the number of mels to be used throughout the project when
    calculating the mel spectrogram.
n_mels = 520

# Defines the maximum frequency that the program cares about. 8,000
    Hz is very high pitched. Frequencies beyond this are not useful.
fmax= 8000

# Defines a list of names of the 12 notes in 12 Tone Equal
    Temperament Tuning. Used in note naming process.
names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#",
     "B"]

# Creates a list of frequencies based on the amount of mels used and
     the boundaries of frequencies used.
freqs = lr.core.mel_frequencies(n_mels=n_mels, fmin=0, fmax=fmax)

# Function which takes a frequency in hertz and returns how many
    semi-tones it is over C0 = 16.35 Hz.
def semitones(hz):
    log = math.log2((hz / 16.35))
    semitones = 12 *log
    return round(semitones)

# Function which takes an amount of semi−tones and returns a note
    with octave. Calculates the semi−tones modulo 12 to get the
    letter. Floor divides by 12 to get the octave.
def notes(x):
    r = round(x)
    note = r % 12
    octave = r // 12
    return names[note]+str(octave)

# Computes the mel spectrogram of the loaded signal.
S_high = lr.feature.melspectrogram(y_high, sr=sr_high, n_mels=n_mels
    , fmax=fmax)
# Converts the power spectrogram from amplitude squared to Decibels
    dB.
log_S_high = lr.power_to_db(S_high, ref=np.max)
# Averages the value of each mel over the length of the signal.
    Creates the line of the Fourier transform spectrum.
freq_bins_high = np.mean(log_S_high[:,:100],axis=1)

S_mid = lr.feature.melspectrogram(y_mid, sr=sr_mid, n_mels=n_mels,
    fmax=fmax)
log_S_mid = lr.power_to_db(S_mid, ref=np.max)
freq_bins_mid = np.mean(log_S_mid[:,:100],axis=1)

S_low = lr.feature.melspectrogram(y_low, sr=sr_low, n_mels=n_mels,
    fmax=fmax)
log_S_low = lr.power_to_db(S_low, ref=np.max)
freq_bins_low = np.mean(log_S_low[:,:100],axis=1)

# SciPy function which detects peaks from an input array. Note the
    parameters. These are set arbitrarily. Takes peaks from each
    time window of the spectrogram.
peaks_high = np.array([ sci.signal.find_peaks(s, width=6, distance
    =8,threshold=5)[0] for s in log_S_high.T ])

peaks_mid = np.array([ sci.signal.find_peaks(s, width=6, distance=8,
    threshold=5)[0] for s in log_S_mid.T ])
```

```

peaks_low = np.array([ sci.signal.find_peaks(s, width=6, distance=8,
     threshold=5)[0] for s in log_S_low.T ])

# Defines a crude piano roll which is to visualize the peaks
     detected over time. Takes the respective peak as a number of
     semitones above C0 and plots it as a rectangular unit on a
     labeled grid.
def piano_roll(peaks):
    sample = []
    for i in range(len(peaks)):
        sample.append([semitones(freqs[peaks[i][j]]) for j in range(
     len(peaks[i]))])


    plt.figure(figsize=(12,8))
    ax = plt.gca()

    for i,notes in enumerate(sample):
        for note in notes:
            r = Rectangle((i,note),1,1)
            ax.add_patch(r)

    ax.set_yticks([12*i+0.5 for i in range(13)])
    ax.set_yticklabels([f'C{i}' for i in range(13)])
    ax.set_xlim(0,len(sample))
    ax.set_ylim(0,96+1)

    for y in np.arange(*ax.get_ybound(),1):
        ax.axhline(y,color=(0.5,)*3,lw=0.5)

    plt.show()


print('High Noise')
piano_roll(peaks_high)
print()
print('Mid Noise')
piano_roll(peaks_mid)
print()
print('Low Noise')
```

```
piano_roll(peaks_low)
```

References

[1] Nasser Abbasi. The application of fourier analysis in solving the computed tomography (ct) inverse problem. Nov 2008.

[2] Paul Brossier, Tintamar, Eduard Müller, Nils Philippsen, Tres Seaver, Hannes Fritz, cyclopsian, Sam Alexander, Jon Williams, James Cowgill, and Ancor Cruz. aubio/aubio: 0.4.9, February 2019.

[3] Jean Baptiste Joseph Fourier. *The Analytical Theory of Heat.* Cambridge Library Collection - Mathematics. Cambridge University Press, 1878.

[4] A. Dos Santos Franco and Norman J Rock. The fast fourier transform and its application to tidal oscillations. *Boletim do Instituto Oceanográfico*, 20(1):145–199, 1971.

[5] Carl Friedrich Gauss. Theoria attractionis corporum sphaeroidicorum ellipticorum homogeneorum, methodo nova tractata. *Werke*, page 279–286, 1877.

[6] Joseph-Louis Lagrange. Solution de différents problèmes de calcul intégral. *Miscellanea Taurinensia*, III, 1762.

[7] Brian McFee, Colin Raffel, Dawen Liang, Daniel P W Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and Music Signal Analysis in Python. page 7, 2015.

[8] Motor1. IIHS Muscle Car Crash Tests, May 2016.

[9] S. S. Stevens, J. Volkmann, and E. B. Newman. A scale for the measurement of the psychological magnitude pitch. *The Journal of the Acoustical Society of America*, 8(3):185–190, 1937.

[10] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.