Capstone Project: Real Time Musical Audio Detection, Conversion & Transcription

ECE-499: Capstone Design Project Part III

By Ian Krause & Raphael Sebastian II

Advisor: Aussie Schnore

March 21st 2019

Report Summary

The following report details the design process, implementation, and results of a system intended to transcribe music to MIDI and sheet music in real-time. The system is intended to use an electric guitar as its input instrument, and record and process the instrument's output using a computer or microcontroller. It is written in the Python programming language and uses several Python-specific libraries and modules. The system is intended to assist DIY musicians with limited music theory knowledge, and to be low-cost and open source. The project focused on monophonic audio transcription, or music with one note at a time (no chords), with a tempo less than or equal to 120 BPM, though research was done into polyphonic transcription, which requires a significantly more complex algorithm. The major design goals were that the system be roughly the size of a large guitar pedal currently on the market, have a low transcription error rate at 120 BPM and below, operate in real-time, and be low cost and easy to use/improve upon. The final system meets many of the initial design goals, though its error rate is much higher than we had hoped due in large part to the complexities of a real-time processing and scheduling, and harmonic transients that interfered with our peak-based pitch detection method.

Table of Contents:

Terminology

- MIDI - Musical Instrument Digital Interface - a file format and technical communication protocol that carries data for pitch, volume, rhythm, and tempo.
- MIDI Data - Processed data that contains the elements needed to generate a MIDI note or series of MIDI notes.
- DAW - Digital Audio Workstation - an application software that records, edits, and processes audio files.
- FFT - Fast Fourier Transform - an algorithm that computes the Discrete Fourier Transform of a sequence from the time domain to the frequency domain.
- MIR - Musical Information Retrieval - a small but growing interdisciplinary field focused on retrieving information from music.
- ISMIR - The International Society for Music Information Retrieval - a non-profit organization which oversees the ISMIR conference and is the world's leading research forum on processing, searching, organising and accessing music-related data.
- AMT - Automatic Music Transcription - The capability of transcribing music audio into music notation using computational algorithms.
- BPM - Beats Per Minute, a musical unit describing the tempo of a song based on how many quarter notes, or "beats," occur in a minute of music.
- PyAudio - Is a Python module that allows programmers to capture audio with Python and manipulate it easily.

- NumPy - NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- LilyPond - LilyPond is a Python module and file format for music engraving. One of LilyPond's major goals is to produce scores that are engraved with traditional layout rules, reflecting the era when scores were engraved by hand.

Introduction

Since digital musicians deal in MIDI files, and analog musicians deal in Western notated sheet music, it seems vital that a tool be developed that can easily transcribe between the two. Currently, the toolset available is expensive and bloated, a mere ad-on feature of sheet music composition software, or a subscription service early in its infancy. Free options have limited features and require payment for full features. In addition, most programs aren't designed to seamlessly generate audio recordings, MIDI files, and sheet music. The goal of this project is to produce a system that will allow users to easily generate recorded audio, MIDI, and sheet music automatically, in real-time, and with low cost and complexity. The following paper details our design for converting analog monophonic audio from a single instrument to MIDI files and sheet music, with the goal being low error transcriptions, low cost hardware, and a final system that is modular enough to be built upon and recreated by others. We also cover ideas for polyphonic transcription, though the data acquisition process varies significantly and is much more difficult due to the nature of polyphonic audio.

1 | BACKGROUND

With the advent of the digital age, it has become possible for any driven artist to become a musician. Some musicians work exclusively using digital software-based tools while others prefer to work using sheet music and physical instruments. Digital musicians use Digital Audio Workstations (DAWs) to produce their work and record using MIDI (Musical Instrument Digital Interface), but often do not have a solid background in music theory. Traditional musicians tend to work mostly with physical instruments and record their work using sheet music of some kind,

which requires knowledge of music theory. As a result, musicians with vastly different experiences and music education backgrounds end up collaborating on many projects. Therefore, a tool for transposing analog audio signals to MIDI files, sheet music, tablature, and other representations of music is necessary for modern musicians on a limited budget who want to work together on complex projects. Traditional musicians benefit from transposing their music directly to MIDI files that can be manipulated much more precisely than raw audio recordings using a DAW.

The goal of this project is to complete a compact, affordable, all-in-one system that bridges the gap between traditional and digital musicians by recording a raw analog signal, transposing from analog audio to sheet music, and creating midi files. Currently, in the market, musicians are limited to pricey subscription-based services, like Lunaverus and Scorecloud, that hide many of the more valuable benefits of basic conversion behind a paywall or simply do not include them at all. The other market options are professional musical transcription software such as Finale or Sibelius, that can cost hundreds of dollars just to get access to their mid-tier version. The other issue with these services is that the transcription error rates they report for testing don't need to be on extreme test cases, so their software could be worse than they report.

In the world of Academia the problem we are attempting to tackle is fundamental to the rapidly growing field of Music Information Retrieval (MIR), called Automatic Music Transcription (AMT). AMT is an end to end process consisting of many components, including pitch detection[1][3][5] and note onset detection[1], each of which have had multiple papers examining them. Only fairly recently, with the development of machine learning, have

researchers attempted to complete the full end-to-end process[2], and there has yet to be a conclusive result as to which method is the best.

The three different papers discussing approaches to pitch detection we focused on include: "Towards Complete Polyphonic Music Transcription: Integrating Multi-Pitch Detection and Rhythm Quantization," which outlines the standard approach to complete AMT (signal to midi to sheet music) and goes into depth about using the Hidden Markov Model to detect the pitch and rhythm of a polyphonic signal[1], "Polyphonic Pitch Tracking with Deep Layered Learning" which discusses polyphonic pitch detection and rhythm detection by way of cascading Neural Networks[3], and "Using a Pitch Detector for Onset Detection," which discusses a method to detect pitch and offset of a monophonic signal[5]. Another paper, "An End-To-End Framework For Audio-To-Score Music Transcription On Monophonic Excerpts," focused on tend-to-end AMT, which takes a monophonic signal and skips the conversion to an intermediate stage (piano-roll data) and goes straight to sheet music using a Convolutional Recurrent Neural Network[2].

It's one thing if our final product works, but it's another if it works and is unethical, so we must consider the ethical implications of the project. Since the goal of our device is to transcribe music, a task that is currently reserved for people who went to music school or at the very least studied music theory in some context, it could be argued that our device could put people out of work, or cause harm to music theory learning institutions. This is not the case. Our tool cannot replace the role of humans in transcribing music, it simply makes it easier for musicians without theory knowledge to create sheet music and work with their music in DAWs.

The ability to interpret and error check traditional music transcription remains a skill that must be learned from professionals.

The other ethical consideration is that this technology could be used for music industry gatekeeping if it were to become a large part of the music industry. Making a cost-effective system, and specifically preventing it from becoming bloated, over-complicated, and overpriced is essential to our goal. An all-in-one system is our final goal, or if this is not possible, a small system that is connected to an external computer running software. Keeping the tech small-scale and musician oriented is our utmost priority. This will prevent industry giants from limiting the distribution of our tech, should it become an industry standard.

2 | DESIGN REQUIREMENTS

We came up with the following design specifications for our final product in order for it to function well and produce results we are satisfied with. This section acts as both a reference for technical specifications, and for more broad goals and stretch objectives. The section is broken down into categories with items that are either demands, marked D, or wishes, marked W.

- Performance
    - The system must be able to take a monophonic signal as input (D)
    - The system must be able to take a polyphonic signal as input (W)
    - The system must be able to take audio files as input (D)
    - The system must be able to produce MIDI output that can be used with external audio processing software (D)

    - The system must operate with an overall percent error of less than 30% for polyphonic transcription (at ≤ 120 BPM)[1](D)
    - Working monophonic transcription error of less than 20% (at BPM ≤ 120)[1] (D)
    - The system should transcribe in real time to facilitate creativity and to be practical for the user (W)
    - The system should have a user-set tempo that can go up to at least 150 BPM[1] (W)
- Geometry
    - The size of the system will most likely be small by virtue of the fact that we plan to use a microcontroller, but we hope that it will be the size of a large guitar pedal (W).
- Economic
    - The parts for the system should cost no more than $150 (W)
    - The whole system should be cheaper than similar proprietary software to recreate (W)
- Energy
    - The system should be able to be powered by a standard 9V or 12V power source (W)
- Environmental & Safety
    - Should operate at given constraints for room temperature (approx. 25° C) (D)
    - System should be properly grounded and use low voltage so as to be safe for user contact (D)
- User Interface
    - The user interface should display information about the system BPM, which note is being read, how long the recording has been running, whether it's recording or not, and whether or not it's powered (D)
    - The user should have access to each of the steps of the full AMT process (i.e. just MIDI to sheet music, or just audio to MIDI, etc.) (D)

    - The user should be able to select when the system begins and ends recording (D)
- Materials
    - The prototype will most likely be made in a rectangular wood or aluminum case so that it is easy to customize for the project constraints (W)
- Manufacturability
    - We will document our device creation process well enough to be reproduced by other students if they want to continue and improve out work (W)
    - The system will use off the shelf components that can be utilized easily for replicating the device (W)
- Social and Cultural
    - By creating this device, we bridge an important generational & technological divide and help to preserve traditional music notation and make it more accessible to digital musicians. The design we choose should not make music notation obsolete, but should increase the ease with which it can be used. (W)
- Aesthetics
    - The system will have a minimalist design that emphasizes practicality and functionality (W)
- Engineering standards
    - Adhere to applicable IEEE standards and restrictions (D)

3 | PRELIMINARY PROPOSED DESIGN

High Level Design & Hardware

Fundamentally, our design is simple (Fig. 1). The system will take audio generated by an instrument as its input, detect when notes begin, analyze those notes frequency to determine their pitch, and detect when they end. Then it will process and package this data, combining it with other known, externally provided information (BPM, start time) to generate MIDI data that can

be turned into a MIDI file, be translated into sheet music, or both. The process is similar for polyphonic audio, but we use different methods to achieve the same MIDI data, which is then converted to MIDI files or sheet music.

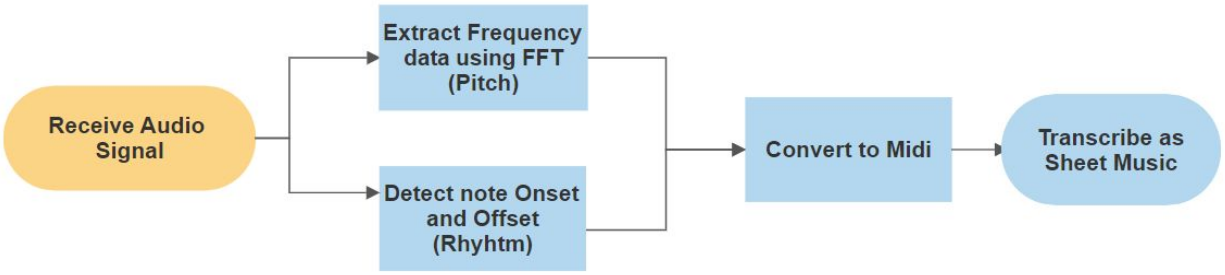


Figure 1: Functional Decomposition of Monophonic Signal Transcription

To functionally implement our specific design (Fig. 2), we have chosen to focus on the guitar as our instrument of choice for audio signal generation. We will connect it to a direct injection box (DI box) which is essentially a transformer that balances the signal coming from the guitar and boosts signal strength, and we will connect the DI box to the Raspberry Pi 4 with our Python-based audio to MIDI software via USB. The Raspberry Pi can then output whichever file is preferred via USB to an external computer. The Pi can output MIDI to a computer with a DAW installed for editing MIDI and producing music, or it can output the finished sheet music as a MusicXML file to be edited or as a PDF.

We will use a Behringer passive DI box, which is an inexpensive box that will work well for balancing our signal and converting the high output impedance signal source (on the order of 50k Ohms) to a low impedance source (100-200 Ohm range)[6]. We will connect the DI box to the Raspberry Pi using an XLR to USB cable. The Raspberry Pi will process the signal and output the chosen file type, then output via USB to a PC.

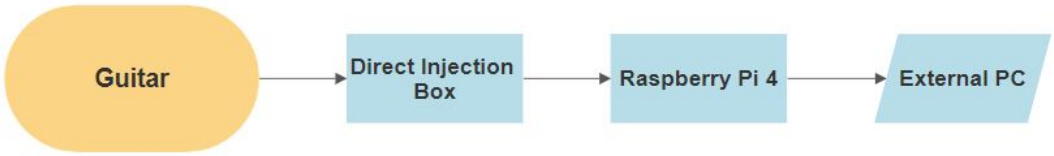


Figure 2: Hardware Configuration

Low Level Design and Software

The process that the Python-based Raspberry Pi code follows for monophonic audio consists essentially of converting the time domain audio signal into the frequency domain, searching for the highest amplitude peaks and detecting their frequencies, detecting their onset times, then associating them with the closest pitch that is recognized as a note in Western music notation, and detecting its offset time. It simultaneously records the audio in the time domain, and packages the MIDI data into MIDI files which are converted to sheet music afterwards using an outsourced Python module called LilyPond. Monophonic audio can be analyzed in real time, or from pre-recorded audio using our current planned implementation. Polyphonic audio will likely require analysis from recorded audio because of the added complexity of a machine learning algorithm, however we need to do additional testing to figure out how feasible real time polyphonic transcription may be.

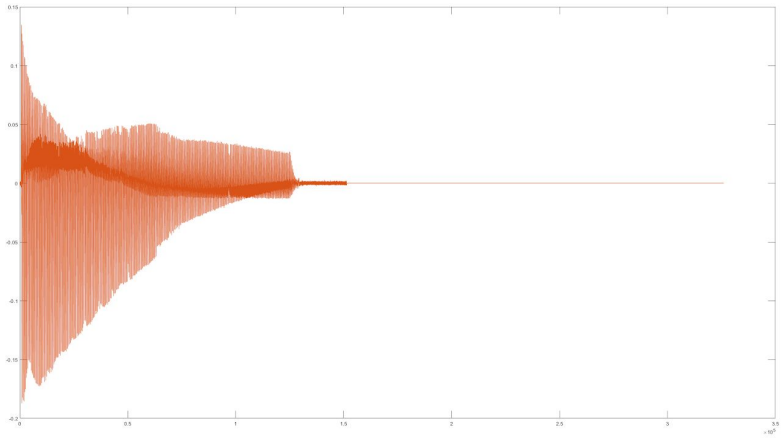


Figure 3: Raw Monophonic Audio

Fig. 3 contains an example of a monophonic signal we might get. This signal is in the time domain, thus its y-axis corresponds to the amplitude (volume) of the signal. In order to extract frequency data from this signal, we have to apply a Fourier Transform of some type in order to get it into the frequency domain.

Figure 4: FFT of Monophonic Signal

Fig. 4 shows the result of running the original signal through a Discrete Fourier Fourier transform, also known as a fast Fourier Transform (FFT) in Matlab. The x-axis is the frequency

and the y-axis is the amplitude of the spectrum frequency. The next step was learning how to obtain FFTs in real-time, and the algorithm we used along with some important functions are shown in Fig. 5, while the results of entering a pure tone (whistling) into the real time FFT are shown in Fig. 6 (code in Appendix B).

```
SPECTRUM ANALYZER(AudioFile)
1  while system is active
2      take chunk of raw data
3      convert raw data to Numpy array of integers(Numpy.Array)
4      add offset to integers to put values in the proper range
5      take Discrete Fouier Transform(Scipy.FFTPack.FFT) of chunk
6      normalize the data
7      display the data(Matplotlib.fig.canvas.draw)
```

Figure 5: Algorithm for the Spectrum Analyzer

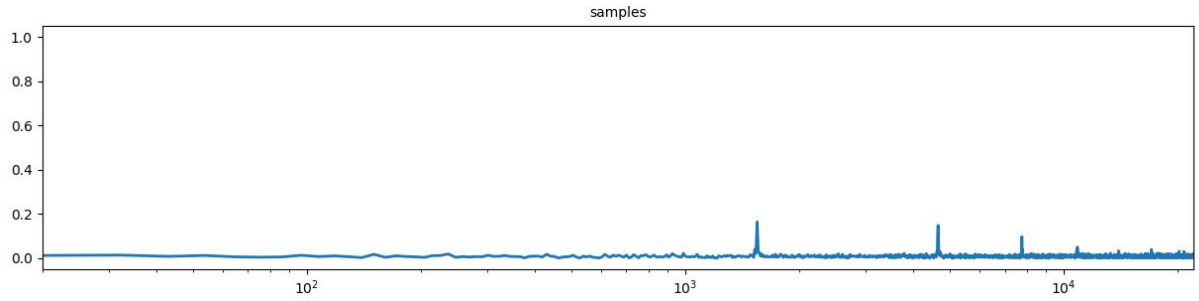


Figure 6: Spectrum Analyzer Response to Pure Tone

Fig. 6 demonstrates that when we create a pure tone, we receive the fundamental frequency as the highest peak, and then its harmonics afterwards. This means that in order to get the pitch of a monophonic signal, all we need to do is find the maximum value of the spectrum analyzer to get the fundamental frequency and convert that frequency to a pitch. The algorithm for it is roughly described in Fig. 7.

```
PITCH DETECTION(AudioFile)
1  while system is active
2      take chunk of raw data
3      convert raw data to Numpy 2D array of integers(Numpy.Array)
4      add offset to integers to put values in the proper range
5      take Discrete Fouier Transform(Scipy.FFTPack.FFT) of chunk
6      normalize the data
7      convert to 1D Numpy Array(Numpy.Flatten)
8      store the fundamental frequency(Scipy.SignalArgrelextrema)
9      convert to pitch(p = 12 * log2((Frequency + 440)/440))
```

Figure 7: Potential Algorithm for Pitch Detection

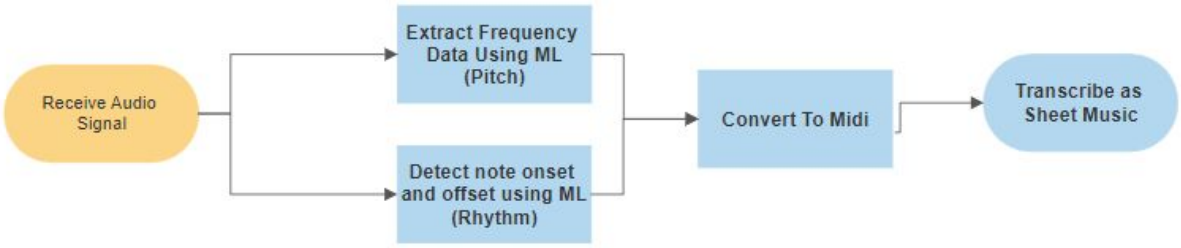


Figure 8: Alternate Functional Decomposition of Polyphonic Signal Transcription

In terms of the next steps, we need to figure out how to detect note onset and offset to get rhythm data of a monophonic signal[5] and then we can create a midi note, and we can then create sheet music from the ability to find midi notes. Once we figure out how to create sheet music for Monophonic signals, we must figure out how to do the same for polyphonic signals. The approach for polyphonic signals (Fig. 9) is to use one of the Machine Learning methods described in the Nakamura paper[1] and the Elowsson paper[3], in order to detect note pitch, offset and offset in order to create MIDI data.

4 | DESIGN ALTERNATIVES

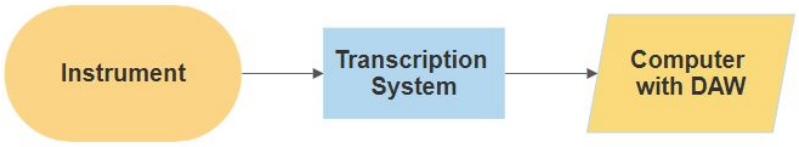


Figure 9: High-Level Functional Decomposition

The fundamental idea behind this system is to take output from a real instrument (in our testing we will use a guitar), convert that input to MIDI and sheet music and export it to a computer with a DAW (Fig. 9). There are multiple ways that we considered creating this system. One way was taking a similar approach to the paper by Román et al.[2] and creating the sheet music directly from the signal itself while we create the midi data in a separate process. We decided against this approach for multiple reasons. The biggest reason being that it only worked for monophonic signals and we wanted the ability to detect polyphonic signals. On top of this, since it uses deep learning to create sheet music data while also trying to create midi data in an entirely separate process, the processing power required would be immense and slow the system down. For these reasons, we decided to approach this problem by using the MIDI data to help build the sheet music.

The challenge once we decided to take this approach was to decide how we would create the MIDI data and how we would build the sheet music, both of which have multiple approaches. The MIDI data could be generated by Machine Learning methods/Convolutional Neural Networks being run on the signals, or, if we are just working with monophonic signals, it can be done with Fourier Transforms to get the pitch and some calculus in order to calculate the onset

and offset times[5]. Machine learning has parameters that can be trained to accurately predict complex results. The main downside is that it requires a decent amount of storage and isn't computationally efficient. Another idea we had was generating a look-up table of wave forms that we could reference in order to quickly identify frames of data. We decided fourier transforms would be best for monophonic signals, due to their computational simplicity, and Machine Learning for polyphonic signals due to their compositional complexity, though we did not get there.

Once we have the MIDI data we must convert that into sheet music and that can also be done in a variety of ways. One approach is to export the MIDI data to an external program that automatically creates sheet music from MIDI. The other way to go about this is to use a musical transcription programing module such as Lilypond. We decided to go with Lilypond in order to allow us maximum control of the system and because it has a function to automatically create sheet music from a MIDI file, though we did not integrate this feature into our final version, it is trivial using the system we've created.

5 | FINAL DESIGN & IMPLEMENTATION - Ian & Rafi

Hardware Implementation

We used the following setup to record audio from an electric guitar for MIDI conversion; the guitar was connected via a ¼" instrument cable to a passive direct injection box. The DI box was connected with an XLR to USB 2.0 cable with a built-in ADC. This was connected to the computer, which could be the Raspberry Pi, or a standard laptop or desktop capable of running depending on user preference. For the final tested system iteration, the system ran on a laptop

with Windows 10. No additional hardware was required, but if desired, the user is able to connect the parallel output of the DI box to an amplifier to better hear themselves, and make sure their note accentuation is precise enough to ensure good detection.

Figure 10: Hardware Diagram

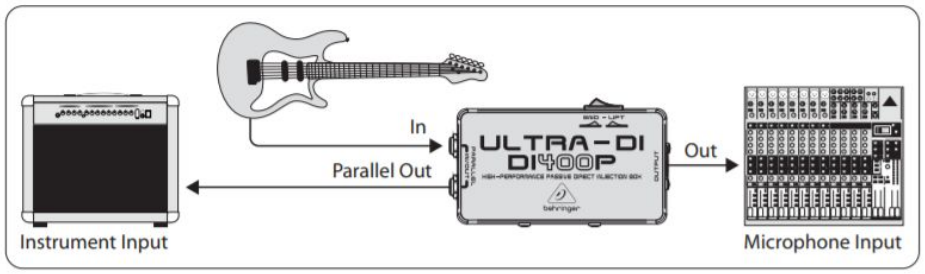


Figure 11: Optional DI Configuration for External Amplification

Pitch & Rhythm Detection Algorithm Implementation

The fundamental idea of this process is to check every frame of data read in from the guitar, calculate and store the frequency values and assign a made up number to that frame, so that it can be referenced in the future. There are some frames that can be disposed of immediately as they are just noise but others that need to be tracked no matter what, so as a result we determine MIDI data after the real-time stream is done. The overall algorithm for our design is shown below in Figure ø.

1. For each Frame of input data:
2. Convert Frame to Frequency Domain using DFT
3. Find Peak Frequency, P
4. If P is in the range of guitar frequencies:
5. convert frequency to midi, store and display
6. store time data and display
7. Use calculated data to build MIDI file
8. Use MIDI file to build PDF of sheet music

Figure ø: Real-Time Sheet Music Generation Algorithm

Step 1 of the algorithm is handled by the Pyaudio non-blocking mode stream pulling in data every TimePerTick (variable defining how long a tick lasts) seconds. Step 2 is handled using Scipy's FFT function in order to perform a discrete fourier transform to get frequency data in Numpy format, then we find the max of that array in order to find the fundamental frequency of the frame for step 3. Steps 4 through 6 are handled by lines 187 - 227 of the code.

$$f = 440*2^{((n-69)/12)}$$

Equation ¥: Conversion from Frequency to MIDI Note Value

These lines store and process the read pitch data through a series of if-statements in order to calculate MIDI data based on equation ¥ the tick number of the frame being processed. These checks are supposed to determine the MIDI value read for the frame's fundamental frequency, whether we care about it and whether it is a part of a note or just noise. If the frame is the last of a MIDI note, then the MIDI note value is displayed along with its onset and duration in ticks. Step 7 of the algorithm is completed by the buildMIDI() function, which cycles through the data stored from the real-time processing. Step 8 is discussed in the next section

File Creation Implementation

The file creation is the most crucial step, as this is the step that builds the usable end product for the target audience of the project. As the program runs, the raw audio is stored as a .WAV file while the data for the MIDI file is being calculated and stored in real-time based on the raw audio. This MIDI file is stored in the same folder the program is being run from, as is the .WAV.

6 | PERFORMANCE ESTIMATES & RESULTS

The physical components of the system were expected to be less than $150 and be about the size of a large guitar pedal. We were able to achieve both, as the total cost of the system is $95.98, including the Raspberry Pi, and the system is just a USB to XLR cord, a ¼" instrument cable and a D/I Box.

The expected runtime performance for this system was to run in real time with a total accuracy rate of 80% and was mostly based around rough estimates of how other papers did in this category. In order to test our design, we ran 6 total tests on the MIDI data, three playing the C scale at 60, 90 and 120 BPM and the others playing the E scale at 60, 90, and 120 BPM. We then used this to calculate a pitch accuracy, rhythmic accuracy and total accuracy. Pitch Accuracy was measured by if each of the notes appeared within the scale, within the time played. Rhythmic Accuracy was determined by if there was a MIDI Note with an onset on each beat. The total accuracy was determined by if the note was both Rhythmically accurate and had an accurate pitch. The MIDI files generated by the tests can be viewed in Appendix D and below are the results of the 6 tests:

Musical Scale	C*	E	C*	E	C	E	C	E
BPM	60		90		120		**Avg**	
Pitch Detection Accuracy	77.80%	44.44%	100%	55.60%	55.60%	44.44%	75.33%	48%
Rhythm Detection Accuracy	55.60%	44.44%	44.44%	22.22%	55.60%	22.22%	53.67%	29.33%
Total Accuracy	44.44%	44.44%	44.44%	22.22%	55.60%	22.22%	**46.67%**	**38.67%**

*Denotes Test generated extra note

Table 1: Performance Test Results

While these results are less than stellar, this was a test on the first iteration of the full working system and the data was neither quantized nor cleaned up after initial recording. These are both common techniques done automatically by many real time recording softwares and we ran out of time to implement it. Another reason the performance could have been so poor is that our algorithm may be fundamentally flawed (i.e. an error in our logic) or there is an error in our code (i.e. an error in our implementation). While the latter is more likely, some more iterations of this algorithm and a couple more tests could prove the former true.

7 | PRODUCTION SCHEDULE

Over Winter Break: We continued research on pitch detection and rhythm detection methods.

Weeks 1-2: We worked on transitioning from our real-time visualization code to fundamental frequency recognition, using the peak frequency in the Fourier domain as our presumed fundamental pitch. We also implemented a formula for converting raw frequency to a MIDI note number.

Weeks 3-5: We devised a simple system using common, low cost, off-the-shelf devices to connect a guitar almost directly to the computer, and purchased these parts. We modified the code to work on a Raspberry Pi to verify that it could meet our dimension requirements. We continued work on our algorithm, discovering that pitch and rhythm detection must be built together to generate MIDI files and minimize error. This led to breakthroughs in our code, namely waiting for a frequency to be detected in multiple consecutive samples before declaring it a note, and setting hard limits on the lowest and highest notes that could be detected, which eliminated transients and false detections.

Weeks 6-7: We began working through major difficulties with rhythm detection that prevented us from collecting audio at a fixed rate. We profiled our code to figure out which operations took the longest and what we could do to make each sample take less time. We changed our algorithm to use non-blocking mode and added a timer. We also reexamined what happened to the frequency domain signal in real-time (since we were able to use direct input from the guitar to the computer) and discovered harmonic transients that made notes that were played quickly (a quarter note at 120 BPM for instance) difficult to detect using peak-detection alone, and discussed alternate methods for detecting fundamental frequency.

Weeks 8-10: We developed a version of the code that allowed the user to set the BPM and that recorded the raw audio and generated MIDI files simultaneously so we could compare them. A metronome was added to determine if the music's speed in BPM was correct, and to allow users to synchronize their playing with the recording. We observed that generated MIDI files were much longer than they should be and worked on modifying formulas in the code to account for processing time delay and correct the length of the generated MIDI files.

8 | COST ANALYSIS

Table 1 contains our parts, and costs, totalling at $95.98, meeting our cost requirement. Our parts list assumes the user already has a guitar with a ¼" instrument cable. Note that the software can also be run on a computer besides a Raspberry Pi, assuming the user already has one, which lowers the cost to $33.99. This is very economical for a DIY musician, our target user demographic.

Part Name	**Part Description**	**Part Link**	**Part Price**
4GB RAM Raspberry Pi 4	Microcontroller that will hold the software	https://www.adafruit.com/product/4296	$55.00
XLR to USB Connecter	Connect interface to Microcontroller	https://www.amazon.com/Microphone-Connector-Microphones-Instruments-Recording/	$10.99
Behringer DI Box	An analog device to receive a cleaner signal	https://www.guitarcenter.com/Behringer/ULTRA-DI-DI400P-Passive-Direct-Box.gc	$23.00
32 GB MicroSD	Additional storage for the Raspberry Pi	https://www.amazon.com/SanDisk-Ultra-MicroSDHC-Memory-Adapter/dp/B073JWXGNT/ref=dp_ob_title_ce	$6.99

Table 2: Component List

9 | USER MANUAL

Hardware Set-Up

1. Connect D/I Box to Computer or RaspberryPi with USB to XLR cable
2. Connect guitar to D/I Box with ¼" cable

Software Set-Up

1. Make sure device has a python 3 interpreter
2. Install the following python Modules:
    a. pyaudio
    b. os
    c. struct
    d. numpy
    e. time
    f. math
    g. cProfile
    h. keyboard
    i. Wave
    j. Winsound
3. Download the Code

Execution

1. Navigate to code from terminal/ command prompt
2. Run code

File Location

1. Go to Folder where code is saved
2. Identify file named "recorded_audio.wav" - For raw audio
3. Identify file named "midiTest.MIDI" - For MIDI file

Notes

- Modify ticksPerBeat - line 43 -to change the amount of accuracy per tick
- Modify MINREAD - line 62 - to change the fastest note the system can read
- To change how long the program runs for, modify line 92 with the maximum amount of ticks desired

- Use an online MIDI to Sheet Music Converter in order to generate sheet music, such as https://solmire.com/miditosheetmusic/

10 | CONCLUSIONS & RECOMMENDATIONS

Our system was designed as a tool to flatten the learning curve associated with transcribing music and provide DIY musicians with an accessible, low-cost, compact tool for automatic transcription to MIDI and sheet music. We succeeded in creating a low cost system with simple modifiable parameters and straightforward implementation in an accessible and efficient coding language (Python, using NumPy) that generates audio and MIDI files, and whose progress is well documented for future interested parties to improve upon and use. The system uses Python, a couple libraries and several modules to detect and transcribe audio in real-time, however we were only able to complete this for monophonic audio detection due to the complexity of polyphonic transcription and time constraints.

The overall performance of the system from our first test yielded an average accuracy rate of 75.3% for a C major scale pitch detection, and 53.7% accuracy for rhythm detection. For the E major scale test, it yielded an average accuracy result of 48% for pitch detection and 29.3% accuracy for rhythm detection. These fell short of our desired accuracy measures significantly, but we were sure that given more time we could improve the algorithm, and even attempt to tackle polyphonic transcription. In particular, two main problems prevented the system from working as well as it could have.

The first was scheduling, or forcing the computer to take snapshots of the incoming signal at regular intervals. Because of the musical nature of the data we were trying to record, it

was particularly important to get this right, and while we came close, even in our current version of the code scheduling has not worked as well as we had hoped, yielding inconsistent data. This is an area that would benefit from more time spent on it, and could be improved in the future.

The other problem was harmonic transients that occur naturally in instruments. Certain notes have higher harmonics (integer multiples of the fundamental frequency of the note) that arise at the beginning of the string being struck that are higher in amplitude than the fundamental frequency, and certain notes have lower harmonics. We observed that over time these transient harmonics die out, but it can take a few seconds which is much too long if the user is playing many notes in quick succession. We considered solutions using lookup tables that stored key information about each note's frequency domain characteristics to help the algorithm match played notes to the correct MIDI note, but we did not complete this due to time constraints. This would be another key area for future work.

The biggest lessons that we learned were that frequency domain signal analysis is very tricky and that there are many approaches to note detection, and that real-time programs require significantly more precision and speed than programs that work by processing saved data retroactively. We also learned that the problem we originally sought out to tackle is very possible to overcome, and nearly all the tools we need already exist, it was just a matter of putting all of them together.

References

[1] Nakamura, E., Benetos, E., Yoshii, K., & Dixon, S. (2018). "Towards Complete Polyphonic Music Transcription: Integrating Multi-Pitch Detection and Rhythm Quantization". *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. doi:10.1109/icassp.2018.8461914

[2] M. A. Román, A. A. Pertusa, and J. A. Calvo-Zargoza (2018, September). "AN END-TO-END FRAMEWORK FOR AUDIO-TO-SCORE MUSIC TRANSCRIPTION ON MONOPHONIC EXCERPTS," in *19th International Society for Music Information Retrieval Conference, Paris, France*.

[3] Elowsson, A. (2018). "Polyphonic Pitch Tracking with Deep Layered Learning." arXiv preprint arXiv:1804.02918.

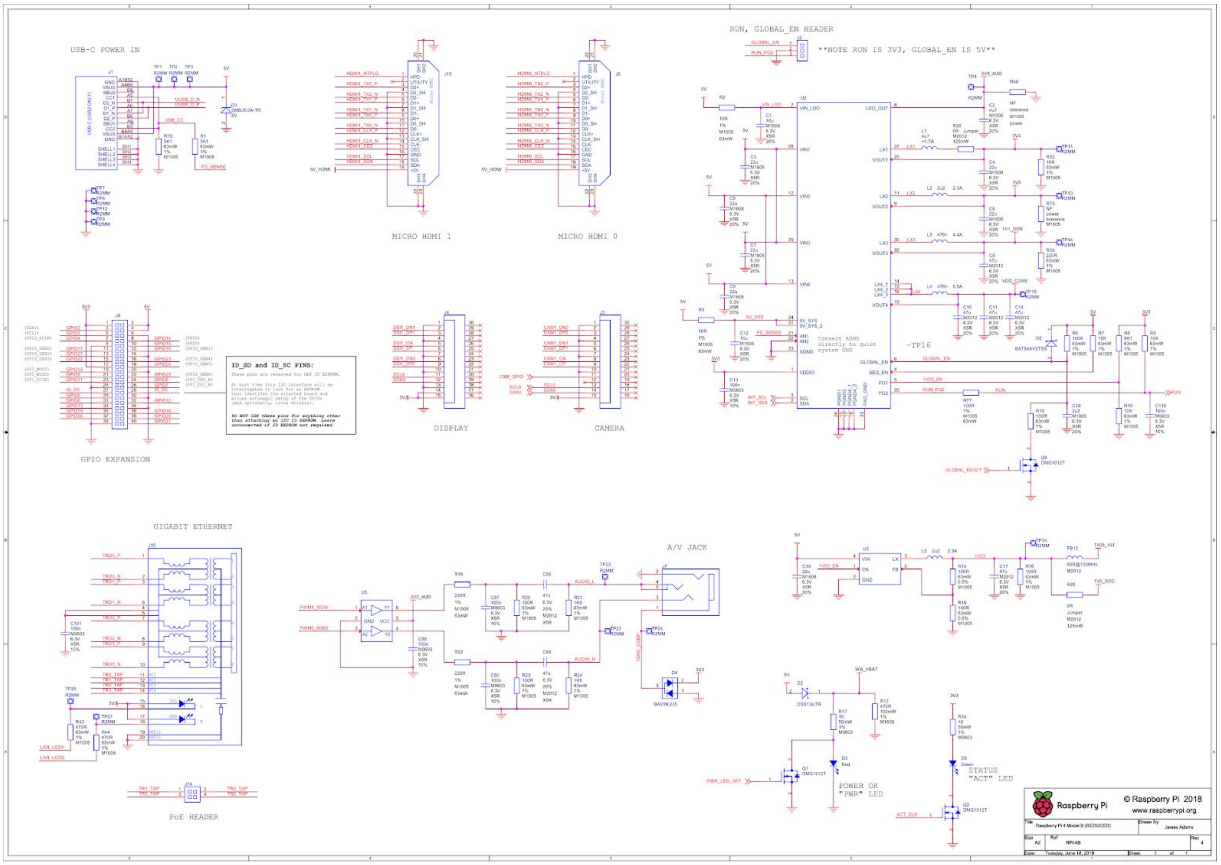
[4] Glover, J. C., Lazzarini, V., & Timoney, J. (2011). "Python for Audio Signal Processing."

[5] Collins, N. (2005, September). "Using a Pitch Detector for Onset Detection," In ISMIR (pp. 100-106).

[6] Finnern, T. (1985). AES E-Library: "Interfacing Electronics and Transformers." http://www.aes.org/e-lib/browse.cfm?elib=11561

Appendices

Appendix A: Raspberry Pi Pinout:



https://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/rpi_SCH_4b_4p0_reduced.pdf

Appendix B: Visualization Code

```
%matplotlib notebook

import pyaudio
import os
import struct
import numpy as np
import matplotlib.pyplot as plt
import time
from scipy.fftpack import fft
from tkinter import TclError

CHUNK = 1024 * 4 #samples per frame
FORMAT = pyaudio.paInt16 #pulls audio info as 16bit
```

```
CHANNELS = 1 # number of channels
RATE = 44100 #sample rate

fig, (ax, ax2) = plt.subplots(2, figsize=(15,7))
p = pyaudio.PyAudio() #create pyaudio object

stream = p.open(
    format = FORMAT,
    channels = CHANNELS,
    rate = RATE,
    input = True,
    output = True,
    frames_per_buffer = CHUNK
)

# variable for plotting
x = np.arange(0, 2*CHUNK, 2)
x_fft = np.linspace(0, RATE, CHUNK)

# create random line w/ random data. Can use either semilogx or plot
line, = ax.plot(x, np.random.rand(CHUNK), '-', lw=2)#you only need one
chunk because we slice the data in half in the while loop
line_fft, = ax2.semilogx(x_fft, np.random.rand(CHUNK), '-', lw=2)

# basic formatting for axes
ax.set_title('AUDIO WAVEFORM')
ax.set_xlabel('samples')
ax.set_ylabel('volume')
ax.set_ylim(0,255)
ax.set_xlim(0,2*CHUNK)
plt.setp(ax, xticks=[0, CHUNK, 2*CHUNK], yticks=[0,128,255])

ax2.set_xlim(20, RATE / 2)

# show the plot
plt.show(block=False)

print('stream started')

# for measuring frame rate
frame_count = 0
start_time = time.time()

while True:

    # binary data
    data = stream.read(CHUNK)
    data_int = struct.unpack(str(2*CHUNK) + 'B', data)

    # convert data to integers, make np array, then offset it by 127
    data_np = np.array(data_int, dtype='b')[::2] +128
```

```
        line.set_ydata(data_np)

        # get fft, slice, and rescale to get magnitudes
        y_fft = fft(data_int)

         # multiply by 2 and divide by the number of frequencies in spectrum
    times the amplitude of the waveform
        line_fft.set_ydata(np.abs(y_fft[0:CHUNK]) * 2 / (256 * CHUNK))

        # update figure canvas
        try:
            fig.canvas.draw()
            fig.canvas.flush_events()
            frame_count += 1

        except TclError:

            #calculate average frame rate
            frame_rate = frame_count / (time.time() - start_time)

            print('stream stopped')
            print('average frame rate = {:.0f} FPS'.format(frame_rate))
            break
```

Appendix C: End-To-End Real-Time Audio & MIDI File Generation Code

```
#Import statements
import pyaudio #The module used for pulling in audio information
import os
import struct
import numpy as np #Module used to convert Audio info into easily processed
arrays
import time #used to continuously call the stream by sleeping for the proper
amount of time
import math #used to perform certatin advanced mathematical calculations
import cProfile #used to get timing info on the performancce of the program
import keyboard
import wave #how we make wave recording
import winsound #Sound for metronome

from midiutil import MIDIFile #how we build MIDI
from scipy.fftpack import fft   #Fast Fourier transform is a computaionaly
effective fouier transform
from tkinter import TclError    #acces TclError funtion of Gui pack

MINMIDI = 28 #min midi value considered by Algo
MAXMIDI = 108 #max midi value considered by Algo
data = []

#create list of midi and frequency values
n = MINMIDI
```

```
noteNfreq = []
while n <= MAXMIDI:
    f = 440*(2**((n-69)/12))
    noteNfreq.append(f)
    n = n + 1
noteArray = np.asarray(noteNfreq)

#defines stream information
CHUNK = 1024 * 4 #samples per frame
FORMAT = pyaudio.paInt16 #pulls audio info as 16bit
CHANNELS = 1 # number of channels
RATE = 44100 #sample rate
FREQ = 2500  # Set Frequency To 2500 Hertz
DUR = 2 #set Duration of metronome to To 2 ms == .002 second

# for measuring frame rate
frame_count = 0

totalTicks = 0
ticksPerBeat = 4

#keep track of notes
noteValue = [] #stores midi Values of read data
noteTime = [] #stores time(ticks) midi Values were read at

#keep track of track
midiNote = []  #Defines note values of midi Files
mididuration = [] #defines length of note values within a Midi File
midionset = [] #defines starting point of midi values within a Midi File
tracker = 0 #tracks num of values have been tracked by the note
track    = 0
channel  = 0
midiTime = 0   # In beats
tempo    = int(input("Set Tempo in bpm: "))  # In BPM
volume   = 100 # 0-127, as per the MIDI standard

#tempo data
secPerQnote = 1/(tempo/60)
MINREAD = 16 #1/how long the smallest note is
secPerMin = secPerQnote/(MINREAD/4)
TICKTIME = secPerMin/ticksPerBeat
delay = 0

#stores audio data to provide recording
framez = []

# get device information of system
# d = pyaudio.PyAudio()
#
# for i in range(d.get_device_count()):
#     dev = d.get_device_info_by_index(i)
#     print((i,dev['name'],dev['maxInputChannels']))
```

```
# d.terminate()

#Calbackfunction that puts pyaudio in non-blocking mode
def callback(in_data, frame_count, time_info, status):
    global totalTicks
    global data
    global delay
    global TICKTIME

    data = in_data
    #if the program took longer to run than the required tick time
    if(TICKTIME-delay > 0):
        time.sleep(TICKTIME-delay)
    else:
        print("wrong")
    #Only allows program to run for 300 ticks, for the purpose of testing
    if(totalTicks > 300):
        return (data, pyaudio.paAbort)

    return (data, pyaudio.paContinue)

#uses midi info stored from the real time processing to build midi file
def buildMidi():
    global track
    global channel
    global midiTime
    #global duration = noteDuration/(total time/(beat per second))  # In beats
    global tempo
    global volume
    global totalTicks
    global ticksPerBeat
    global midiNote
    global mididuration
    global midionset

    MyMIDI = MIDIFile(1) # One track, defaults to format 1 (tempo track
                     # automatically created)
    MyMIDI.addTempo(track, midiTime,tempo)
    delay = midionset[0]


    #add logic to convert ticks to time value

    for num in range(0, len(midiNote)):
        MyMIDI.addNote(track, channel, midiNote[num],
math.floor((midionset[num]-delay)/2), mididuration[num]/2, volume)

    print(midiNote)
    print(midionset)
    with open("midiTest.MIDI", "wb") as output_file:
        MyMIDI.writeFile(output_file)
```

```
def main():
    global TICKTIME
    global data
    global noteValue
    global noteTime
    global totalTicks
    global ticksPerBeat
    global midiNote
    global mididuration
    global tracker
    global midionset
    global framez
    global data
    global delay
    global secPerQnote
    #instantiate PyAudio
    p = pyaudio.PyAudio()

    #open Stream Using Callback
    stream = p.open(
            format = FORMAT,
            channels = CHANNELS,
            rate = RATE,
            input = True,
            output = False,
            #input_device_index=1,
            frames_per_buffer = CHUNK,
            stream_callback = callback
        )

    streamin = True
    #Count in
    print("Stream start in 4")
    for x in range(0,3):
        winsound.Beep(FREQ, DUR)
        time.sleep(secPerQnote)

    #start Stream
    stream.start_stream()
    while stream.is_active():
        startTime = stream.get_time()
        if(totalTicks % 16 == 0):
            winsound.Beep(FREQ, DUR)
        if(data != []):
            framez.append(data)
            data_int = struct.unpack(str(2*CHUNK) + 'B', data)
            # convert data to integers, make np array, then offset it by 127
            #this is the magnitude data?
            data_np = np.array(data_int, dtype='b')[::2] +128

            # get fft, slice, and rescale to get magnitudes
```

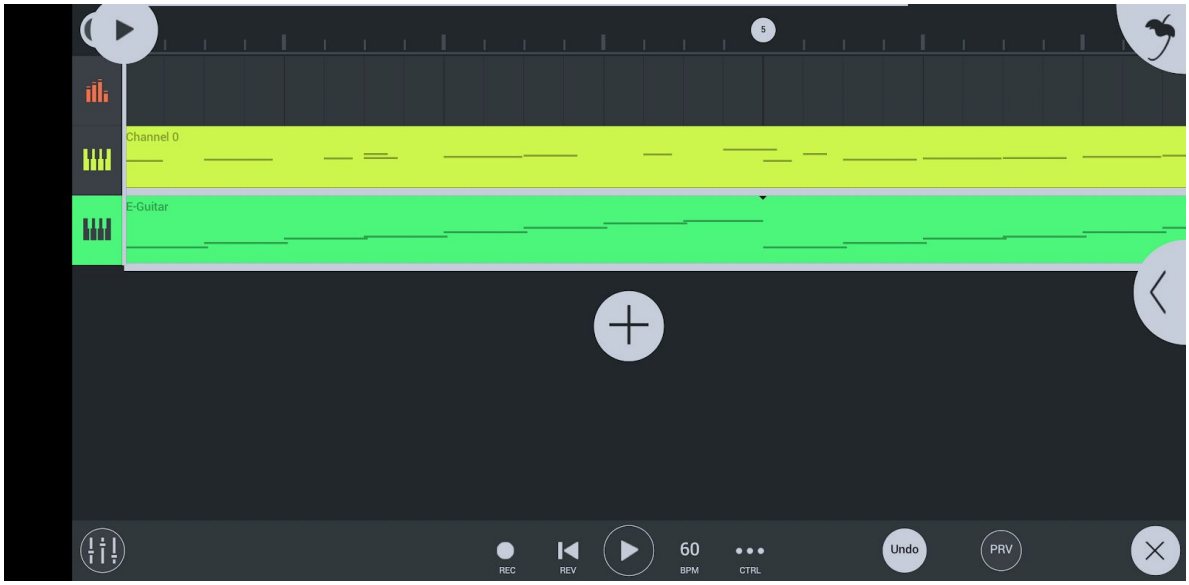
```
            # multiply by 2 and divide by the number of frequencies in spectrum
times the amplitude of the waveform
            y_fft = fft(data_int)
            frequencies = np.abs(y_fft[0:CHUNK]) * 2 / (256 * CHUNK)
            maxf = max(frequencies[1:])
            flist = frequencies.tolist()
            findex = flist.index(maxf)


            if findex < 1000:
               #find midinums
                frequency_val = findex * (RATE/CHUNK)
                arrindex = (np.abs(noteArray - frequency_val)).argmin()
                midiNum = arrindex + MINMIDI


                #if we find a midiValue we wanna keep
                if midiNum <= MAXMIDI:
                   #if there have already been a notes worth of midiValues
stored up
                    if len(noteValue) >= ticksPerBeat:
                       #if the midi we see is the same as the other notes, add
it to the list
                            if midiNum == noteValue[tracker-3] and midiNum ==
noteValue[0]:
                                noteValue.append(midiNum)
                                noteTime.append(totalTicks)
                                tracker = len(noteValue)
                            #otherwise add note to track, display and clear
variables
                            else:
                                midiNote.append(noteValue[0])
                                midionset.append(noteTime[0]/ticksPerBeat)
                                duration = (totalTicks -
noteTime[0])/ticksPerBeat
                                mididuration.append(duration)
                                print ("{} note, at time {}, for
{}".format(noteValue[0], noteTime[0], duration), end="\r")
                                tracker = 1
                                noteValue = [midiNum]
                                noteTime = [totalTicks]
                    #if the note is empty
                    elif len(noteValue) == 0:
                            noteValue.append(midiNum)
                            noteTime.append(totalTicks)
                            tracker = 1
                    else:
                            #if note is the same as last, store value else,
restart
                            if noteValue[tracker-1] == midiNum:
                                noteValue.append(midiNum)
                                noteTime.append(totalTicks)
```

```
                            tracker = len(noteValue)
                        else:
                            tracker = 1
                            noteValue = [midiNum]
                            noteTime = [totalTicks]
        delay = stream.get_time()-startTime
        #if the program ran slower than the required ticktime.
        if(TICKTIME-delay > 0):
            time.sleep(TICKTIME-delay)
        else:
            print("wrong")
        totalTicks = totalTicks+1


    #Stop Stream
    stream.stop_stream()
    stream.close()

    #close pyAudio
    p.terminate()
    wf = wave.open('recorded_audio.wav', 'wb')
    wf.setnchannels(CHANNELS)
    wf.setsampwidth(p.get_sample_size(FORMAT))
    wf.setframerate(RATE)
    wf.writeframes(b''.join(framez))
    wf.close()

    #build Midi
    buildMidi()
    return

#cProfile.run('main()')
```

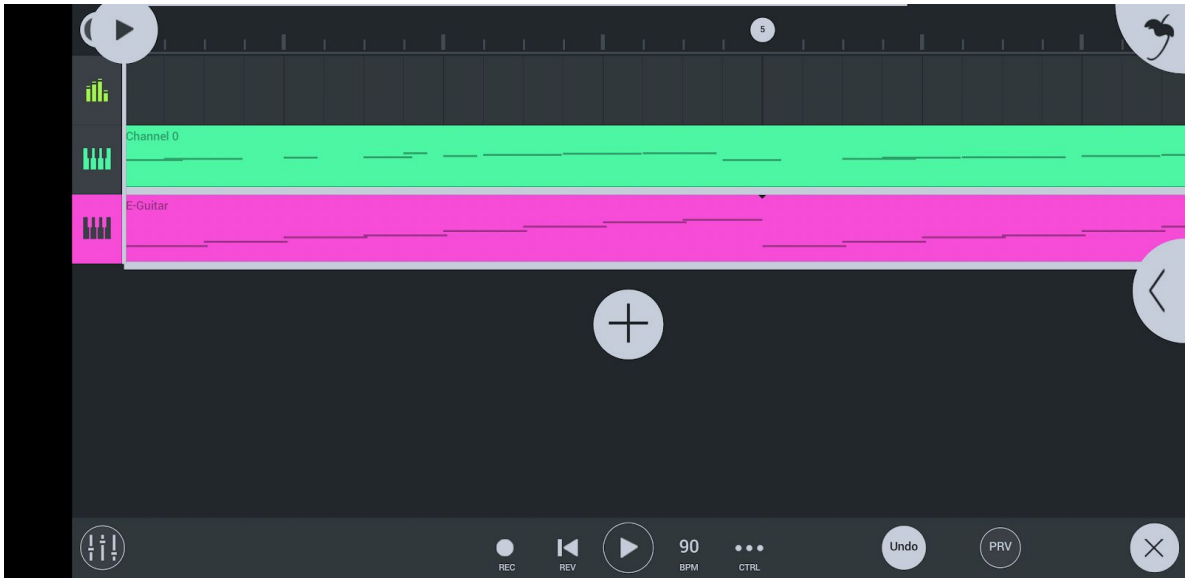
Appendix D: MIDI Test Screenshots

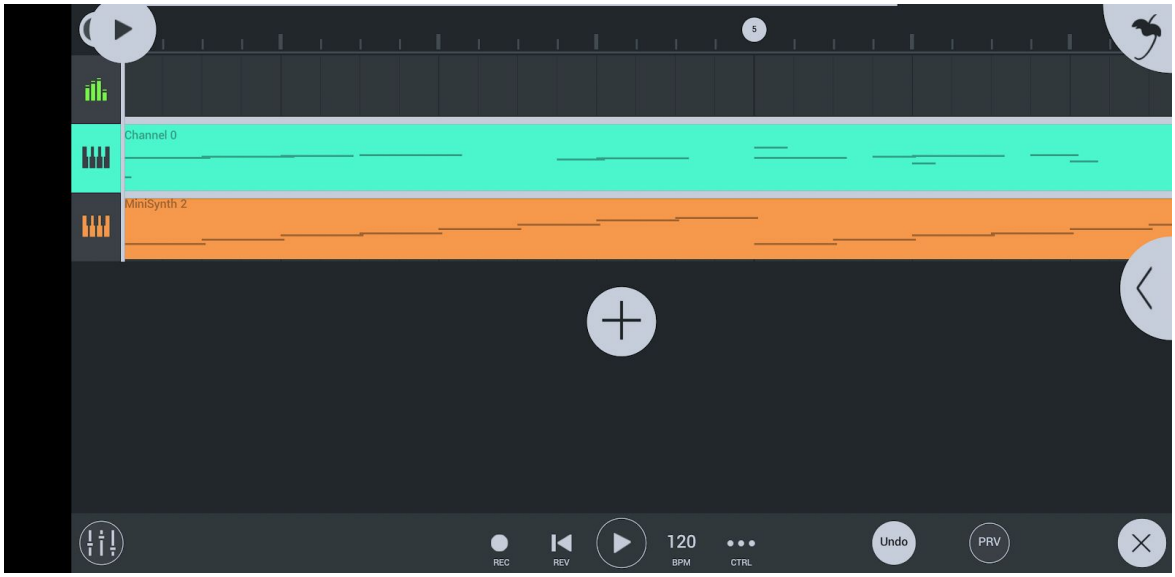
1) **E Scale - 60 BPM**
2) **E Scale - 90 BPM**
3) **E Scale - 120 BPM**
4) **C Scale - 60 BPM**
5) **C Scale - 90 BPM**
6) **C Scale - 120 BPM**

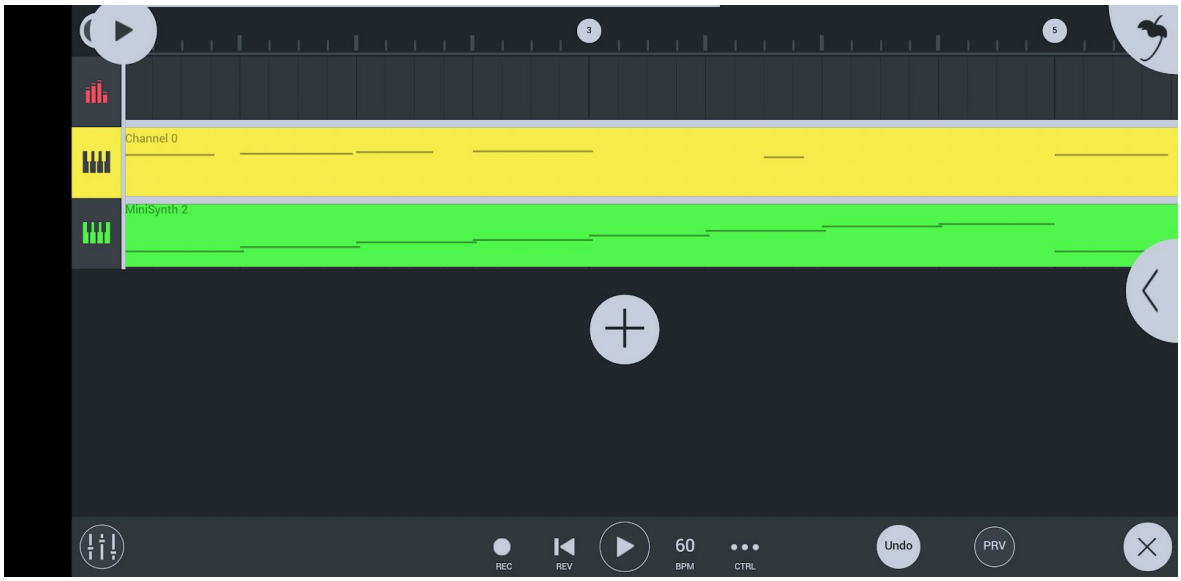
1)


5
Channel 0
E-Guitar
REC
REV
60
BPM
CTRL
Undo
PRV

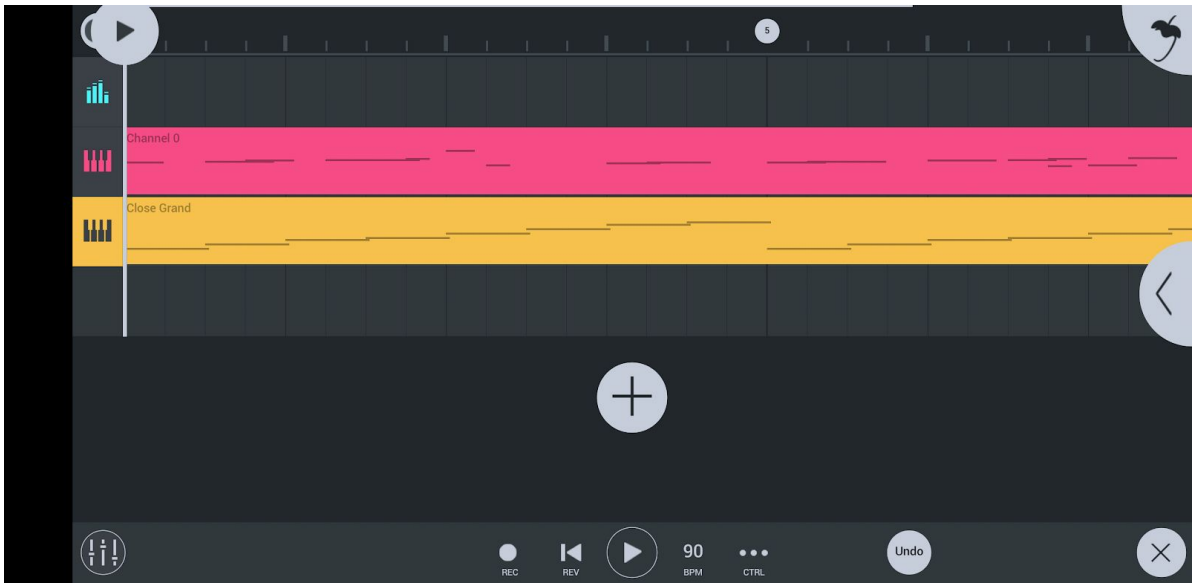
2)


5
Channel 0
E-Guitar
REC
REV
90
BPM
CTRL
Undo
PRV

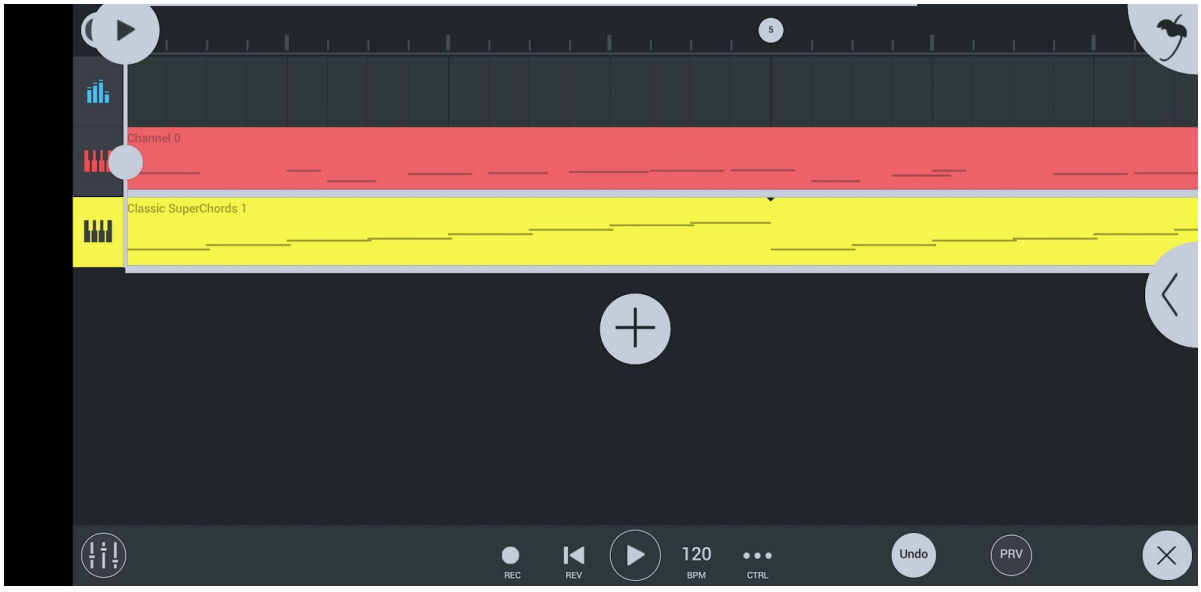
3) 
5
Channel 0
MiniSynth 2
REC
REV
120
BPM
CTRL
Undo
PRV
4)

4) 
3
5
Channel 0
MiniSynth 2
REC
REV
60
BPM
CTRL
Undo
PRV

5)


5
Channel 0
Close Grand
REC
REV
90
BPM
CTRL
Undo

6)


5
Channel 0
Classic SuperChords 1
REC
REV
120
BPM
CTRL
Undo
PRV