

Group 17

Department of Electrical and Computer Engineering
ECE 4600

Android Guitar Trainer With Raspberry Pi Spectrum Analyzer

March 6th 2020

Group Members

Ahmed Kashef AL Ghetaa

Marc Roy

Conrad Rycyk

Greq Sinclair

Yifan Zhang

Academic Supervisor

Douglas Buchanan, Ph.D., P.Eng., FCAE

Abstract

The purpose of this project was to design and implement an Android phone application which teaches the user how to play the guitar. The app prompts the user to play a chord, and then gives live feedback on each note in real-time.

The notes are detected in real-time by a spectrum analyzer running on a Raspberry Pi. The Raspberry Pi uses a USB preamplifier to measure the guitar signal, then performs a series of calculations to find the centre frequency and the harmonics in the signal. This information is then relayed back to the Android device using a Bluetooth communication protocol. The Raspberry Pi is capable of both single-note and multi-note detection so it can work with simple melodies and songs containing simple chords.

The accuracy for the note detection algorithm is nearly 100% for single in-tune notes. Single notes are relatively easy to parse, however when chords are being detected we often receive false-positives due to the similar harmonic properties of the root and fifth of the chord.

The application uses two modes to instruct the user on how to play the guitar. The first is a training mode which displays chords of a song one at a time, along with the finger positions for each chord. In this mode the song progresses when the user plays the chord correctly. This is helpful for developing the sense of what a chord looks like for a beginner and allowing them to focus on hand shapes as opposed to rhythm. The second mode prompts the user to play chords on a side-scrolling fretboard that shows the finger positions for each chord.

The project mainly focuses on signal processing, Bluetooth, and Android development. The details and design decisions of each aspect are discussed below.

Contributions

		AG	MR	CR	GS	YZ
Raspberry Pi	Bluetooth Permissions					
	USB Access Permissions					
	Preliminary Signal Processing Research					
	Design Architecture					
	Fundamental Frequency Collection					
	Fretboard Frequency Entry					
	USB Setup					
	USB Read Algorithm					
	FFT Algorithm					
	Power Spectrum Density Algorithm					
	Plotting Algorithm					
	Spectrum Analyzer Multithreading					
	Bluetooth					
	Note Evaluation Method					
Android	App Structure					
	Bluetooth					
	Bitmap Sprites					
	Scrolling Fretboard Mode					
	Training Mode					
	Alternate Modes					
	Settings					
	File I/O					
	Song/Chord/Scale Library					

Acknowledgements

The completion of ECE 4600 Group Design Project was due in part by the efforts of Aidan Topping, Mount-First Ng and Dr. Douglas Buchanan. The Technical Communication Instructor Aidan Topping helped us with formalizing the design reviews and preparing for oral presentations throughout the course of this project. From the electrical shop Mount-First Ng provided technical support for the team. We finally would like to thank Dr. Douglas Buchanan for helping us to keep on track with the progression and design of the project.

Table of Contents

Abstract	2
Contributions	3
Acknowledgements	4
Table of Contents	5
List Of Figures	8
List Of Tables	9
Chapter 1: Introduction	1
1.1 Motivation	1
1.2 Problem Definition	2
1.3 Performance Metrics	2
1.3.1 Guitar Audio Feedback	2
1.3.2 Note Dissociation	3
1.3.3 Strum Detection	3
1.3.4 Visual Guitar Aid	3
1.3.5 Teaching Style	3
1.3.6 ADC	3
1.3.7 Communication	3
1.3.8 Audio Spectrum Analysis	4
1.3.9 Audio Input	4
1.4 Approach	4
Chapter 2: Background	5
2.1 Musical Notes	5
2.2 Scale	5
2.3 Chord Structure	6
2.4 Tablatures	7
2.5 Guitar Components	8
2.5.1 How Sound Is Generated	8
2.5.2 Guitar Strings	10
2.6 Acoustics	11
2.6.1 Acoustic Guitar	11
2.6.2 Electric Guitar	11
2.7 Fretboard	11
2.7.1 Fret	11
2.7.2 Spacing	12
2.7.3 Fingering	13
2.8 Fundamental Frequency	13
	5

2.9 Summary	14
Chapter 3: System Overview	15
3.1 Microphone	15
3.2 Pre-Amp	16
3.3 Microcontroller	16
3.4 Android Device	17
3.5 Test Guitar	17
3.6 Summary	18
Chapter 4: Signal Analysis	19
4.1. FFT	19
4.2 Referencing Guitar Notes	24
4.3 Harmonics	25
4.4 Tolerance	27
4.5 FFT Buffer	29
4.6 Power Spectrum	29
4.6 FFT vs PSD	30
Table 4.3: Random computational times for Fourier-Transform and PSD	31
4.7 Overall Algorithm	32
Chapter 5: Raspberry Pi to Android Bluetooth Protocol	33
5.1 What Is Being Communicated	33
5.1.1 Android	33
5.1.2 Raspberry Pi	33
5.2 Why was Bluetooth Chosen	33
5.2.1 Client / Server Setup	33
5.2.2 Error Protection	34
5.2.3 Availability	34
5.3 Requirements For Connecting	35
5.3.1 Port	35
5.3.2 Text Protocol	35
5.4 Message Structure	35
5.4.1 JSON	35
5.4.2 Packet	36
5.5 Raspberry Pi	38
5.5.1 Threading	38
5.5.2 Receiving Thread	38
5.5.3 Stream Thread	39
5.5.4 Timed Thread	40
5.6 Summary	41
Chapter 6: Android Application	42

6.1 Why Android	42
6.2 Overview	42
6.3 Android Bluetooth	43
6.4 Low-Level Implementation	45
6.4.1 Beats	47
6.4.2 Fretboard Activity	47
6.4.3 Training Activity	49
6.4.4 Settings	50
6.5 Sprites	51
6.6 Data structures and Memory I/O	54
6.7 Portability	55
6.8 Education	55
6.9 Summary	57
Chapter 7: Testing and Validation	59
7.1 Single Note Comparison	59
7.2 Chords	59
7.3 Acoustic Vs Electric	61
Chapter 8: Future Work And Conclusion	62
8.1 Future Work	62
8.2 Summary	64
Reference	65
Appendix	67
Python Code For Raspberry Pi	67
Android Github Link	77

List Of Figures

Figure Number	Page Number	Figure Number	Page Number
Figure 2.1	5	Figure 4.6	26
Figure 2.2	6	Figure 4.7	28
Figure 2.3	7	Figure 4.8	29-30
Figure 2.4	7	Figure 4.9	32
Figure 2.5	8	Figure 5.1	37
Figure 2.6	9	Figure 5.2	37
Figure 2.7	10	Figure 5.3	39
Figure 2.8	12	Figure 5.4	40
Figure 2.9	12	Figure 5.5	41
Figure 2.10	13	Figure 6.1	44
Figure 2.11	13	Figure 6.2	45
Figure 2.12	14	Figure 6.3	46
Figure 3.1	15	Figure 6.4	48
Figure 3.2	15	Figure 6.5	49
Figure 3.3	16	Figure 6.6	51
Figure 3.4	17	Figure 6.7	52
Figure 3.5	18	Figure 6.8	52
Figure 3.6	18	Figure 6.9	54
Figure 4.1	20	Figure 6.10	57
Figure 4.2	23	Figure 6.11	57
Figure 4.3	23	Figure 7.1	60
Figure 4.4	24	Figure 7.2	61
Figure 4.5	25	Figure 7.3	61

List Of Tables

Table Number	Page Number
Table 1.1	2
Table 2.1	10
Table 4.1	21
Table 4.2	22
Table 4.3	31
Table 5.1	36
Table 6.1	50
Table 7.1	60

Chapter 1: Introduction

This system was designed to improve the user's skills with the guitar. The system has two major parts; the spectrum analyzer that runs on a Raspberry Pi, and an Android app. The Android app prompts users to play a guitar chord and then provides feedback. The Raspberry Pi runs a Python spectrum analyzer program that is controlled by the Android app via Bluetooth. The implementation of this system is described below.

1.1 Motivation

Video games such as "Guitar Hero" and "Rockband" have provided gamers with a unique playing experience that mimics the playing of an instrument. For some, these games have led gamers to pick up a guitar and try to learn how to play the songs they have had fun playing in "Guitar Hero" or "Rock Band". However, learning how to play a real guitar can be difficult, and it can result in frustration and even quitting. This system aims to bridge the gap between guitar video games and practical skills with an intuitive and gamified practice tool.

Virtual lessons have become a popular venue for people to practice the fundamentals of their instrument of choice as live instructors are very expensive. This application serves as an interface that can be used alongside a real guitar to provide a student with a structured method of learning their favorite songs. It is expressed in game format to engage the student in a more satisfying manner. These structured methods of learning songs presented in a gaming environment, along with a little perseverance, should be sufficient to put the user on track to becoming a rockstar.

With this system the user is given feedback on their guitar technique through an Android app pre-downloaded onto their phone. As their skills develop, the user can attempt new songs to further advance their understanding of the guitar. Through this progression, the user will gain an overall understanding of the first five frets of the guitar.. The twelve major and minor open chords, where some strings would not have a finger placed on the fretboard, will be taught. Some barre chords, where an index finger presses all of the strings along a single fret, are included in these lessons. Acquiring these skills will equip a user to learn many songs and transition into more advanced guitar techniques.

1.2 Problem Definition

The interface can be broken down into five major systems:

- An Android app to instruct the user
- A data acquisition method that will receive the acoustic signal coming from an acoustic guitar
- A signal processing algorithm to convert the time series signal into the frequency domain
- A classification algorithm that will identify the chords and notes that are being played from the frequency spectrum of the audio signal and
- A Bluetooth connection between the Raspberry Pi and the Android app

1.3 Performance Metrics

Table 1.1 : Summary of Performance Metrics

Feature	Value or Range	Objective Met
Guitar Audio Feedback	Real Time	Yes
Note Dissociation	Polyphonic	Yes
Strum Detection	Recognizing notes within a recorded time	Yes
Visual Guitar Aid	Fretboard with finger positioning	Yes
Teaching Style	Skill Trees with level ups	No
ADC	44.1kHz	Yes
Communication	Bluetooth	Yes
Audio Spectrum Analysis	Fast Fourier Transform (FFT)	Yes
Audio Input	Microphone or Audio Jack	Yes

1.3.1 Guitar Audio Feedback

Guitar audio feedback refers to how long it takes to relay guitar feedback to the user. As of now there is no delay that can be noticed by humans, as it takes less than 200ms to get a response from the Raspberry Pi.

1.3.2 Note Dissociation

Polyphonic refers to being able to look for more than one note at a time. As the guitar is a six stringed instrument and we were able to distinguish the notes of all six strings from each other we can say this is a polyphonic algorithm.

1.3.3 Strum Detection

From the audio processing algorithm described in chapter four we were able to successfully detect the strumming of the guitar. This was done by finding out when the power level of the signal passes a certain threshold. Once it passes that threshold the note dissociation algorithm takes place.

1.3.4 Visual Guitar Aid

The app's training mode is described in chapter six. It teaches the user individual chords by visually depicting their finger placements. This is done beat by beat through a song to allow the user to learn the material at any pace.

1.3.5 Teaching Style

The larger educational systems originally intended had to be cut from the project due to time constraints. Instead, a progression system was added to the app's training tools, such that each song is initially locked until the user plays its individual components in the training mode.

1.3.6 ADC

The ADC on the Raspberry Pi is able to sample audio at the musical standard of 44.1kHz. This is therefore expandable to virtually any standard USB microphone or preprocessor following industry standards. The sampled audio is later downsampled to 11.025kHz to help improve the speed of the audio processing algorithm as additional sampling rate is unnecessary in terms of accuracy.

1.3.7 Communication

Communication between the Android device and Raspberry Pi was successfully implemented with Bluetooth protocol. No errors have been observed in the finished version of this interface. Whenever the Raspberry Pi encounters a strange packet of information it is caught and ignored. Also if one of the devices disconnects abruptly the other device is able to safely exit and reset connections.

1.3.8 Audio Spectrum Analysis

The spectrum analysis is done using a fast Fourier transform on the USB data to convert it to the frequency spectrum. From the frequency spectrum it is converted to the power spectrum to clear out the noise from the signal in order to make it easier to find harmonics of each of the notes.

1.3.9 Audio Input

The Raspberry Pi is able to read input from any USB sound card device. This requires adjustments to setup variables in the Python code, such as how many channels to read and whether to read the right or left channel. This means that for acoustic guitars audio can be collected using a microphone plugged into the USB port of the Raspberry Pi. For electric guitars a preprocessor must be used to convert the analog signal into a digital one before reading from a USB.

1.4 Approach

The system teaches the user how to play the guitar with an application that provides instruction on what to play and real time feedback based on how the user played. The feedback is derived from guitar audio guided by instructions from the app. This way the system can be used with any guitar. The application is installed onto an Android phone and connected to the Raspberry Pi via Bluetooth. The user selects a song or track from the user interface that contains unique instructions on how that piece of music is to be played. These instructions indicate where to place fingers on the neck and when to strum on the fretboard. The audio that comes from the guitar is processed by the Raspberry Pi to evaluate whether or not the indicated notes have been played. This feedback is displayed on the Android device in the form of properly played notes fading from the screen.

Chapter 2: Background

In order to understand this project, some essential music theory and guitar fundamentals are explained to illustrate how the guitar is played and what causes different sounds. In addition, this chapter shows the frequency components of a single string vibration, which is essential to understand the processing of the sound analysis.

2.1 Musical Notes

The pitch of a musical note may be described as how “high” or “low” a note is perceived to be. The pitch and the frequency of the sound or notes are not synonymous, but are similar concepts that are often used interchangeably. An increase in frequency results in an increase in the pitch perceived. The frequency of a musical note is a measurement of the number of cycles per second of the sound, in units of Hertz, and can be measured with a microphone.

2.2 Scale

A scale contains a number of ascending or descending pitches and ends an octave above where it started. Most “western” or diatonic scales consist of seven different whole notes which are contained in one octave. The names of musical notes are based on the letters of the alphabet from A to G and then repeats on the next octave. The interval between two notes is the “separation” between two pitches, and the distance between two notes of the same letter is called an octave. An interval can be described using half-steps and whole-steps.

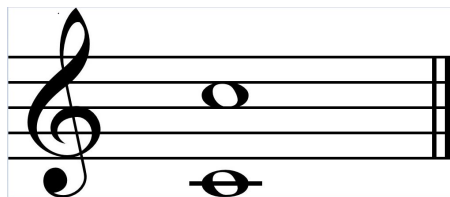


Figure 2.1 : Two C notes spaced an octave apart [1]

Between every note is a whole-step except for the transitions from B to C and E to F which are a half-step. This results in an octave having twelve half-steps also called a semitones. Scientific pitch notation is used to combine the musical note name with a number identifying the pitch’s octave.

A scale ends on the same note that it started, but an octave higher. A frequency ratio of 2 describes the difference in pitch between two notes that are an octave apart. This means that the C at the end of a C major scale with a fundamental frequency of 261.63Hz has double the frequency of the C note at the beginning of a C major scale at 130.81Hz but is typically referenced by concert A of 440Hz. Furthermore a half-step has a frequency ratio of $2^{\frac{1}{12}}$, or approximately 1.0595, of the previous note. In the example above, going up a half step from the C note at 130.81Hz means the next note a half-step away, the C#, is 138.59Hz.

The lowest note that can be played on the guitar is the open 6th E string, where the word “open” refers to the strumming of that string without any fingering.

2.3 Chord Structure

A chord is when multiple notes are played on a guitar at the same time. Chords are formed using an interval formula such that the separation in frequency between multiple notes sounds pleasing to the ear. The simplest chords consisting of three notes is called a triad. A maximum of six notes can exist in a chord when played on a guitar. The most common used triads form major and minor chords.

A triad chord is made up of a root, a third, and a fifth. These names are based on the alphabetical distance from the root. In the example shown below in Figure 2.2, the major C chord has a C note as its root the third is an E which is four half-steps above the root. The fifth of the C chord would be a G, which is six half-steps or five whole notes above the root note.

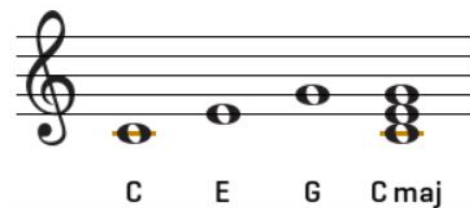


Figure 2.2 : C Major Triad [2]

A minor C chord shown in Figure 2.3 follows a similar pattern except that the middle note (or minor third) is three half-steps above the root, called a minor third. Notes can be added to the chord if they are an octave apart from any of the notes already present.

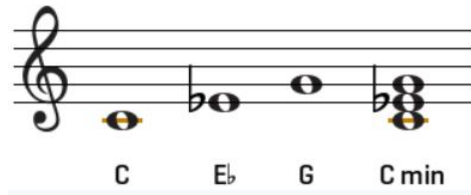


Figure 2.3 : C Minor Triad

2.4 Tablatures

Tablature is a common way of presenting instructions on how to play musical notes on a guitar while on other instruments traditional sheet music as shown in Figure 2.4 is the preferred way to read music. The upper portion of the figure refers to the musical notes and the lower portion to the “tabs” being played on a guitar.

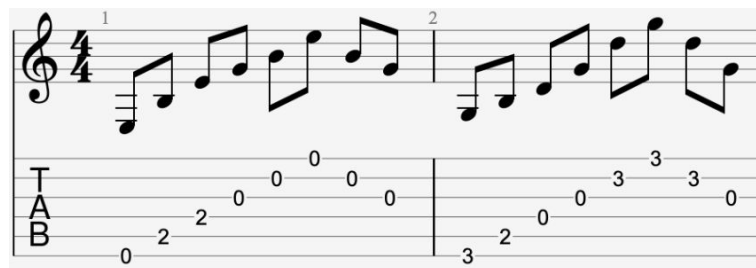


Figure 2.4 : Tablature [3]

The standard music notation, shown in Figure 2.4, displays notes played according to their pitch and duration defined by the symbol and the vertical position on a staff. Learning standard notation for guitar requires the player to locate where all of the notes are present on the fretboard. A guitar tablature, shown in figure 2.4 the standard notation, instead displays a set of fretboard fingering instructions that allow a novice player to locate and produce the same notes of a piece of music. The six lines represent the six strings of a guitar where the line at the bottom represents the low E string and the top the high E. The numbers on the strings are which fret on the neck of the guitar that need to be pressed in order to produce the same note.

The tablature method of displaying instruction can be simply converted into a text based format. Figure 2.5 shows an example of a C chord tablature.

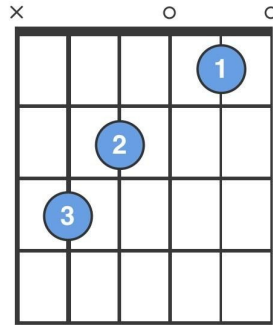


Figure 2.5 : Text based tablature

The notes present in the above chord are C, E, G, C, and E. “o” symbol represents the string is “open” while “x” usually means that string is not “strummed”. The chord above can be represented as [X,3,2,0,1,0]. By using this notation, pieces of music can be written in text form as an array for every instance of time in a song.

2.5 Guitar Components

2.5.1 How Sound Is Generated

The guitar is a stringed instrument with usually six strings each fixed at the two ends. Strings vary in thickness, tension, and the materials used in order for them to produce a specific sound when the string is plucked or strummed. Figure 2.6 shows the main parts of an acoustic guitar. The portion of the string that vibrates when plucked is the length between the saddle and the nut. Whenever a finger is placed on a fret along the length of the guitar, the effective length of the vibrating string is reduced and a different sound is produced.

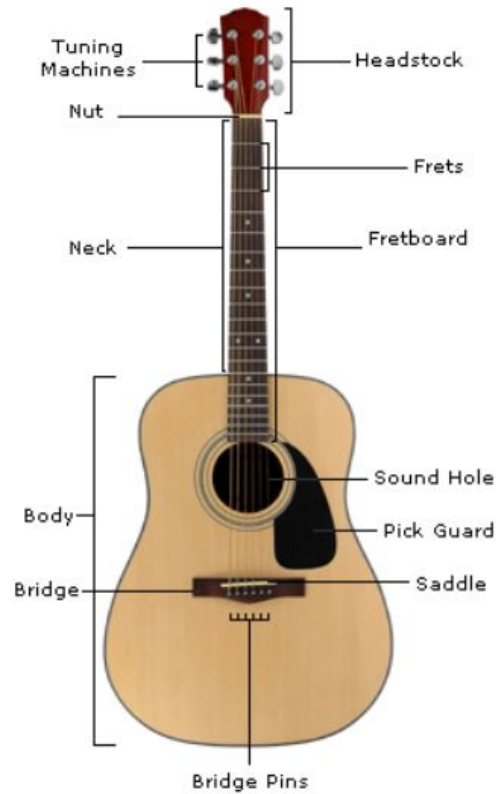


Figure 2.6 : Acoustic guitar parts [4]

Sound is created when a wave motion is produced in the air by the vibration of material bodies. When a string fixed at both ends vibrates, a wave travels along the string travelling in both directions, yielding a standing wave and produces the pitch. The speed of the wave propagating along the string must satisfy the Equation 2.7.

$$v = \lambda f \quad \text{Equation 2.7}$$

where velocity of the wave (v) is equal to the wavelength (λ) multiplied by the frequency (f) of the wave. The waves travel in both directions along the string and are reflected back at each end. However, the waves do not cancel each other out when reflected back upon themselves but form a standing wave. The speed of a wave is also affected by the string tension and the linear mass density and the relation satisfies Equation 2.8

$$v = \sqrt{\frac{Ft}{\mu}} \quad \text{where } \mu = \frac{\text{mass of string}}{\text{length of string}} \quad \text{Equation 2.8}$$

Every string on the guitar has the same length. In order to produce a different pitch for each string, the wave speed must vary across all strings. This is done by varying the tension and linear mass density of each string.

2.5.2 Guitar Strings

Each string has a different linear mass density in order to achieve a different pitch for while maintaining the same length. The most common guitar strings consist of a bronze wrapping around a hex or a round steel core.

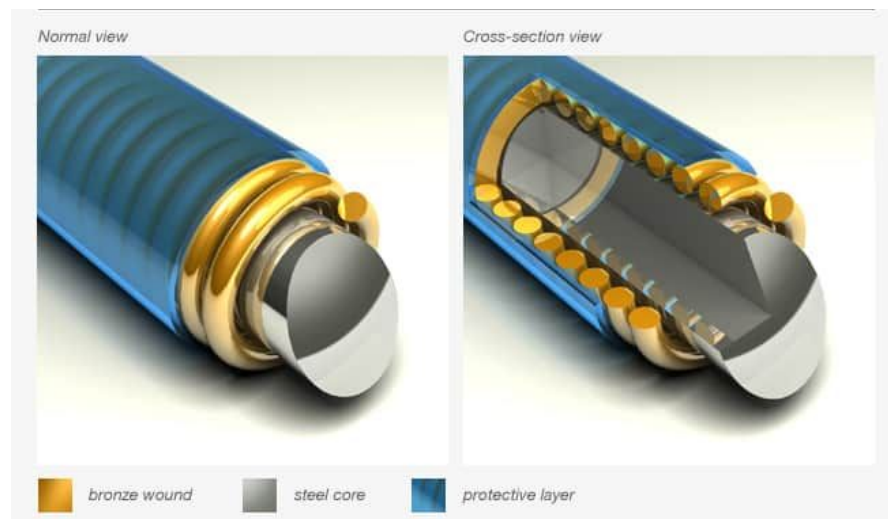


Figure 2.7: normal and cross-section view of a guitar string [5]

Steel strings for an acoustic guitar come in a variety of thicknesses (or gauge) measured in thousandths of an inch each resulting in a different pitch. Typically acoustic guitars have 12 (0.012 inch diameter, 12s) or 13 gauge (13s) strings with the 12s being the most common. This diameter refers to the top E string. Thinner strings require less tension to achieve a given pitch, and are typically easier to play. The thickness of each of the strings are given in Table 2.1.

Table 2.1 : Standard acoustic guitar string sizes [6]

	(E)	(B)	(G)	(D)	(A)	(E)
10s (10-47)	.010	.014	.023	.030	.039	.047
11s (11-52)	.011	.015	.023	.032	.042	.052
12s (12-53)	.012	.016	.025	.032	.042	.053
13s (13-56)	.013	.017	.026	.035	.045	.056

2.6 Acoustics

2.6.1 Acoustic Guitar

A guitar has several sound coupling modes: string to soundboard (front board), soundboard to cavity air through the hole on the front board, and both the soundboard and cavity air to outside air. The back board of the guitar also vibrates, driven by air in the cavity. With the combination of the vibration, the guitar generates and emphasizes harmonics, and it couples this energy to the surrounding air. Since string energy is transmitted efficiently because of the soundboard, compared to a solid electric guitar, the amplitude of sound for an acoustic guitar degrades much faster. [7]

2.6.2 Electric Guitar

Unlike acoustic guitars, solid-body electric guitars have no vibrating soundboard to amplify string vibration. Instead, electric guitars depend on electric pickups (or microphones), an amplifier (or amp) and speaker. Through the solid body, the vibration of the string can be fully collected by the pickups and transferred to amplifier.

2.7 Fretboard

A Fretboard, also called the fingerboard, is a long and thin piece of wood laminated to the neck of a guitar. [8] A guitar player can alter the pitch produced by a vibrating string by pressing down on a fret of the guitar. This shortens the length of the string which in turn increases the frequency of the sound produced. The space between each fret raises the frequency of the resulting sound of a half-step.

2.7.1 Fret

Each fret is separated from the next by a raised metal strip perpendicular to strings against which the player presses the strings as shown in Figure 2.10. Frets let players press the string consistently in the same place, which ensures the notes have the correct pitch.



Figure 2.8 : Fretboard

2.7.2 Spacing

Although all the frets are parallel to each other, the distance between two subsequent frets are different. To enable the guitar to play all notes, two frets side by side have half step differences in pitch. Since a note an octave higher (12 half steps) is produced with twice the frequency, the distance between each fret and the scale length has a certain ratio. The Equation 2.11 and Figure 2.12 shows the relation among the fret number, distance from nut and the scale length for a 24-fret guitar.

$$\text{Distance from the nut} = \text{scale length} \times \left[1 - \frac{1}{2^{\left(\frac{\text{fret \#}}{12}\right)}} \right] \quad [9] \quad \text{Equation 2.11}$$

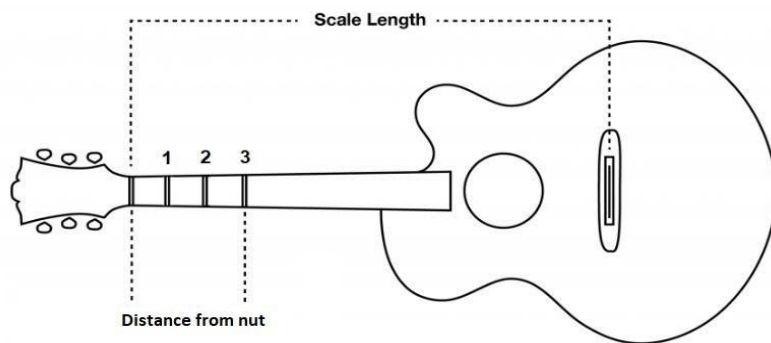


Figure 2.9 : Fret spacing equation explanation

2.7.3 Fingering

To learn the fingering (or finger position) during the guitar play, a chord diagram with finger positions and finger number is shown to let players understand chords clearly so that beginners can easily learn the chords. The finger numbers are defined by:

1 - index finger

2 - middle finger

3 - ring finger

4 - pinky finger

An example of C chord fingering and diagram is shown in Figure 2.13 and Figure 2.14.



Figure 2.10: Finger position [10]

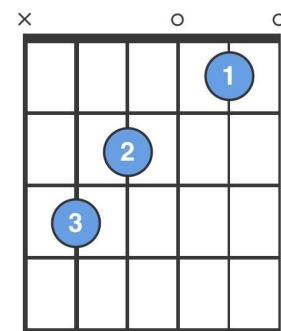


Figure 2.11: Chord diagram [11]

2.8 Fundamental Frequency

The main pitch that is heard from the vibration of any given string is referred to as the fundamental frequency. However, the fundamental frequency is not the only frequency that can come from one string of the guitar. There are also other frequencies called harmonics that are integer multiples of the fundamental frequency, as shown in figure 2.15. When a string is plucked, all of these frequencies happen at the same time. All instruments have a characteristic sound which is due to the different harmonics present in each note played. Harmonics present in a vibrating string are also referred to as an overtone.

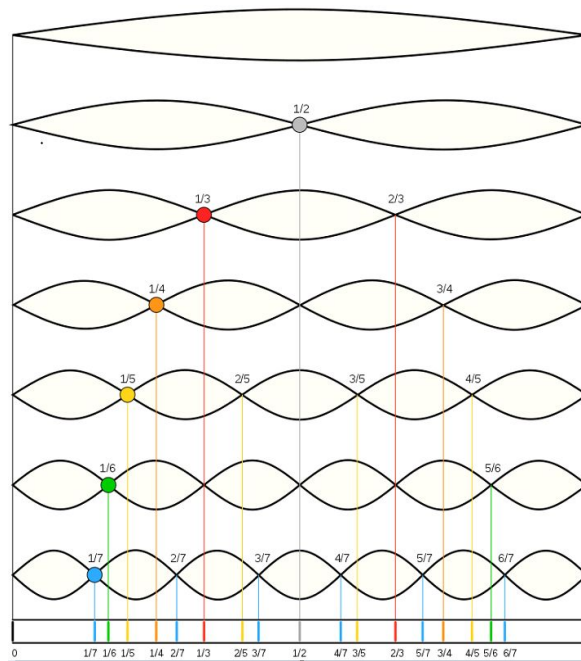


Figure 2.12 : Visual representation of different harmonics [12]

Overtone present in a vibrating string can make pitch detection difficult especially when dealing with multiple notes being played at the same time, such as a chord. If two C notes an octave apart were played simultaneously, the harmonics from the lower C note would be present in the fundamental frequency and harmonics of the higher C note. However these harmonics are detected and are used to determine which note has been played and on which string. This is described in detail later in Chapter 4: Signal Analysis.

2.9 Summary

From this chapter, music theory and guitar components are explained as an overview of the project information from the music aspect. Information about the electric hardwares and devices will be introduced in the next chapter.

Chapter 3: System Overview

The image below depicts the full block diagram of how our project is constructed, four main components can be seen below. A microphone to collect guitar audio, a pre-Amp to convert analog audio to digital data, a Raspberry Pi to analyze the data and an android app in order to give feedback to the user. The Raspberry Pi and the Android device are connected through Bluetooth in order to send commands and data between each other.



Figure 3.1: Block Diagram of Final Design

3.1 Microphone

The audio signal is measured by collecting the sound from a microphone. The microphone used for this project is the Audio-Technica AT2020 cardioid condenser microphone as shown in figure 3.2. This device has a frequency response over 20 to 20,000 Hz [13]. A three-pin XLR connector is necessary to transmit the audio from the microphone to the pre-amplifier, which is capable of sampling at a rate of 44.1 kHz [13]. An XLR connector is a 3 prong connector used for professional audio equipment such as microphones. The preamplifier needs to power the microphone through the XLR connection in order to properly read the audio signal which is known as phantom power. The phantom power necessary to run the microphone is 48V DC, drawing a current of 2mA. [13]



Figure 3.2: AT2020 Cardioid Condenser Microphone

3.2 Pre-Amp

The pre-amp is a necessary intermediary to convert analog audio into digital audio, in order to transmit data from the microphone to the microcontroller. The pre-amplifier is required to supply power to the internal circuitry of the microphone and transmit data to the microcontroller. The pre-amplifier selected for the project is the U-PHORIA UM2. The maximum power consumption of the device is 2.5 W [14] and was powered by the USB connection from the microcontroller. This device is used because it can provide 48V [14] of DC phantom power to the microphone used. When setting up the preamp it is important to pay attention to the amplitude of the guitar and make sure that the audio is not too loud. If the audio is too loud then the peaks of the signal are clipped which adds extra harmonics to the signal which could interfere with the note detection algorithm. Typically for these types of devices when transferring data it is sent as stereo audio. Stereo audio means there are two channels the left is the mike and the right is the electric guitar audio.



Figure 3.3 Behringer UPHORIA UM2 Pre-Amp

3.3 Microcontroller

A microcontroller was necessary to process the audio signals from the microphone as well as send and receive data to and from the Android device. The Raspberry Pi 3B+ with an 8GB SD card is used because the Raspberry Pi is a Linux operating system and capable of using Python. Python is a computer language which has many available open source libraries which

makes app development faster. The Raspberry Pi also has Bluetooth functionality which is necessary for wireless transmission of data to and from the Android device. This device has four USB 2.0 ports which provide power for the pre-amplifier.

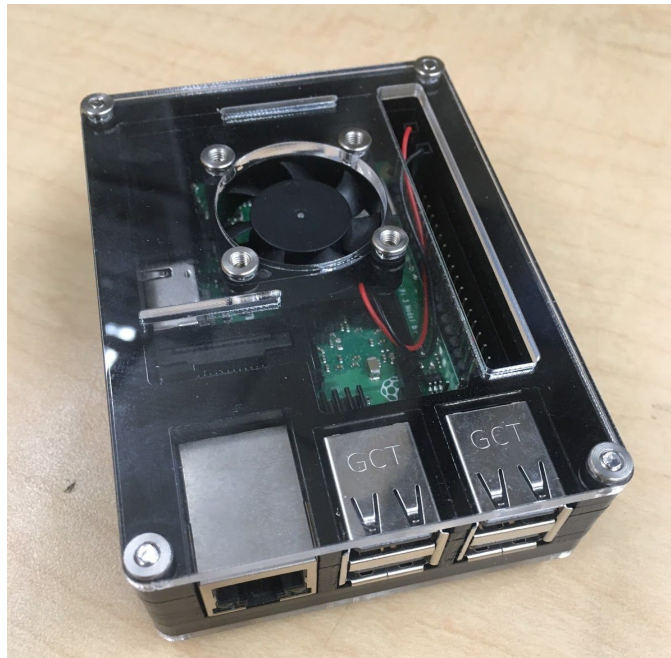


Figure 3.4 : Raspberry Pi

3.4 Android Device

The app that has been developed to work with Android devices. Android Studio is the software used to develop applications to work on Android operating systems. The benefit of using an Android powered phone is that Android Studio is the easiest and most widely used development environment to create apps. Android phones also come with Bluetooth that can be used to communicate wirelessly with the Raspberry Pi.

3.5 Test Guitar

The project is capable of using both an acoustic and an electric guitar. If the user decides to use the project with an acoustic guitar then the microphone will have to be used along with the pre-amplifier. One of the advantages of using an electric guitar is that it can be plugged directly into the pre-amplifier. The guitars shown below are the guitars we used to test our application. The Vox electric guitar has been used to test the note detection capabilities of the Raspberry Pi.

The Vox guitar has an interesting problem where the intonation is off by a quarter to half a semitone along the neck. Do to the heating and cooling of summer and winter guitars neck's can often warp slightly to give off sounding notes. In the case of the Vox guitar the note detection algorithm was still able to detect the notes because the algorithm senses a possible frequency band where each harmonic might be located which gives leeway for slightly out of tune guitars.



Figure 3.5: Acoustic Guitar



Figure 3.6: Electric Guitar

3.6 Summary

From this chapter, combining with the previous one, all the hardwares and devices we use are shown and explained. After establishing the hardware system, we can move the focus to the theory used to analyse the guitar sound. In the next chapter, The methods and algorithm used in the microprocessor will be introduced.

Chapter 4: Signal Analysis

The Fourier Transform is needed in this project because it makes it easier to analyze the signal in frequency-domain rather than in time-domain. It is easier to analyze because the Fourier Transform gives information about the fundamental frequency f_0 . The Fourier Transform on the x-axis represents all the observable frequencies in the spectrum and the y-axis represents the amplitude of each of these frequencies. Therefore the x-axis is used to find a notes harmonic amplitudes and if they are greater than a certain threshold then they are present in the signal and being played by the guitar user. The x-axis measured in Hertz going from 0Hz to half the sampling rate ($\frac{f_s}{2}$) where f_s is the sampling frequency in Hz. This is known as the observable spectrum of frequencies that defines the audio signal and will be used in the sections below.

4.1. FFT

The following section will describe what a Fast Fourier Transform (FFT) is and why it is being used in the note detection algorithm. A guitar audio signal is observably non-periodic or in other words does not repeat itself. A non-periodic signal $x(t)$ is a signal that and can be described by:

$$x(t) \neq x(t + T) \quad [18] \quad \text{Equation 4.1}$$

Where T is the period of the fundamental frequency:

$$T = \frac{1}{f_0} \quad [18] \quad \text{Equation 4.2}$$

The non-periodic signal $x(t)$ can be expressed as sum of infinite exponentials of single frequencies $e^{j\omega}$:

$$e^{j\omega} = \cos(\omega) + j\sin(\omega) \quad [18] \quad \text{Equation 4.3}$$

Euler's identity is equation 4.3 which is the result of Fourier Transform that takes a periodic signal and decomposes it into a sum of infinite complex pairs of sines and cosines. The magnitude $|X(\omega)|$ is taken from each pair of sine and cosine for all frequencies from 0 Hz up to $\frac{f_s}{2}$ Hz, where f_s is the sampling frequency [18].

Thus, the Fourier transform of $x(t)$ can be written as:

$$X(\omega) = F[x(t)] \text{ or } x(t) \Leftrightarrow X(\omega) \quad [18] \quad \text{Equation 4.4}$$

where

$$X(\omega) = \int_{-\infty}^{+\infty} x(t)e^{-j\omega t} dt \quad [18] \quad \text{Equation 4.5}$$

In order for the Fourier Transform for a signal $x(t)$ to exist there has to be three conditions to exist. [15]

1. $x(t)$ should be integratable such that

$$\int_{-\infty}^{+\infty} |x(t)| dt < \infty \quad [18] \quad \text{Equation 4.6}$$

2. $x(t)$ must have a finite number of discontinuities within any finite interval.[18]
3. $x(t)$ must contain only a finite number of maxima and minima within any finite interval. [18]

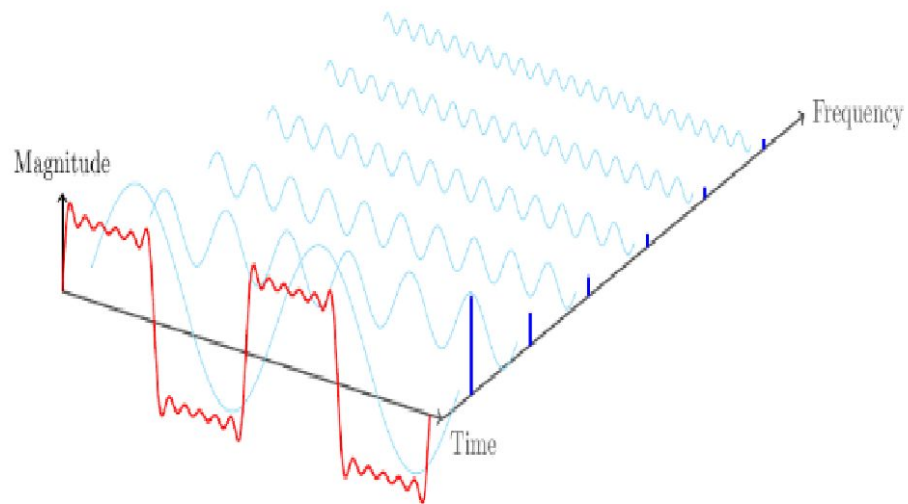


Figure 4.1: Visual illustration of Fourier Transform. The input signal $x(t)$ in red, after applying a Fourier transform resulting in $X(\omega)$ as shown in blue. [15]

An experienced guitar player might recognize a played note or play a chord simply by listening to it. However, to discern a note or a chord, it is easier and more convenient to convert the guitar audio signal to the frequency domain using a Fourier transform to the audio signal. Doing so has two major advantages.

1. The fundamental frequency f_0 is unique for every musical note and can be derived by referencing concert A or 440Hz. These root frequencies all have unique harmonics which are an integer multiple of the fundamental frequency and can be used to identify the note.

Table 4.1: Examples of guitar notes and their corresponding fundamental frequencies f_0 .

Guitar Notes	Fundamental frequency f_0 [Hz]	Harmonic 1 $2f_0$ [Hz]	Harmonic 2 $3f_0$ [Hz]
A_4	440.0	880.0	1320.0
E_2	82.41	164.82	247.23
G_3	196.0	392.0	588.0

2. The harmonics which are multiple integers of the fundamental frequency ηf_0 are used to distinguish between two or more chords. As such several harmonics $[f_1, f_2, f_3, f_4]$ are used in order to differentiate between chords.

Table 4.2: Examples of guitar chords and their corresponding harmonic notes.

Guitar chords	Corresponding Chord Notes
Am	$[E_2, A_2, E_3, A_3, C_4, E_4]$
Am Harmonic 1	$[E_3, A_3, E_4, A_4, C_5, E_5]$
Am Harmonic 2	$[B_3, E_4, B_4, E_5, G_5, B_5]$
E	$[E_2, B_2, E_3, G_3^\#, B_3, E_4]$
E Harmonic 1	$[E_3, B_3, E_4, G_4^\#, B_4, E_5]$
E Harmonic 2	$[B_3, F_4^\#, B_4, D_5^\#, F_5^\#, B_5]$
C	$[C_3, E_3, G_3, C_4, E_4]$
C Harmonic 1	$[C_4, E_4, G_4, C_5, E_5]$
C Harmonic 2	$[G_4, B_4, D_5, G_5, B_5]$

As an example of demonstrating the use of this Fourier Transform technique to identify guitar audio signals (notes and chords). Shown below are four graphs, two time domain and two frequency domain pairs for two guitar notes (E2 and A4) and two chords (F and G). Referred to as Figure 4.2 and Figure 4.3 respectively.

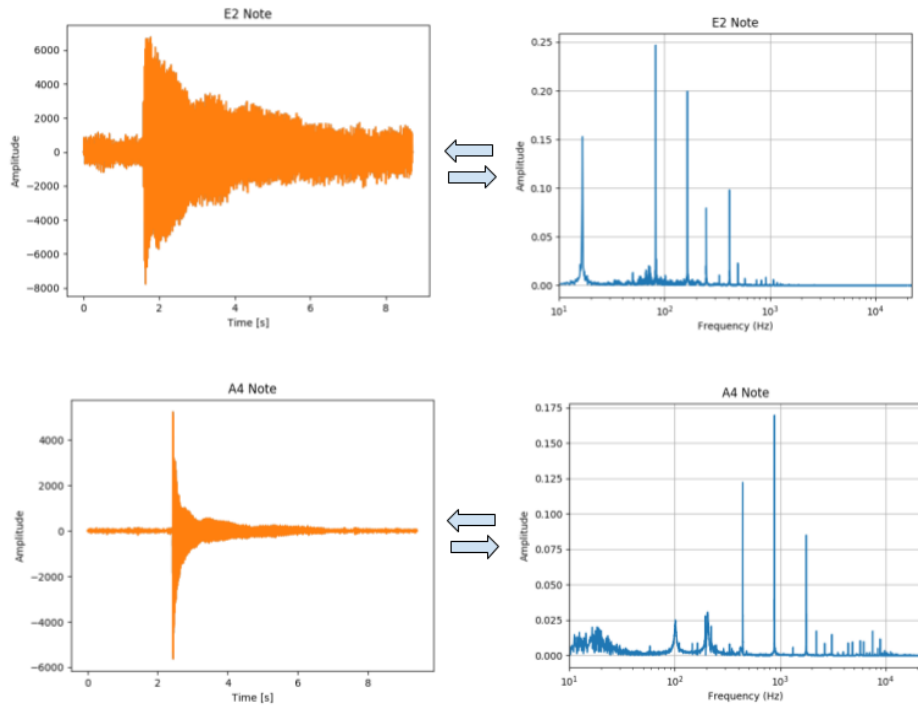


Figure 4.2

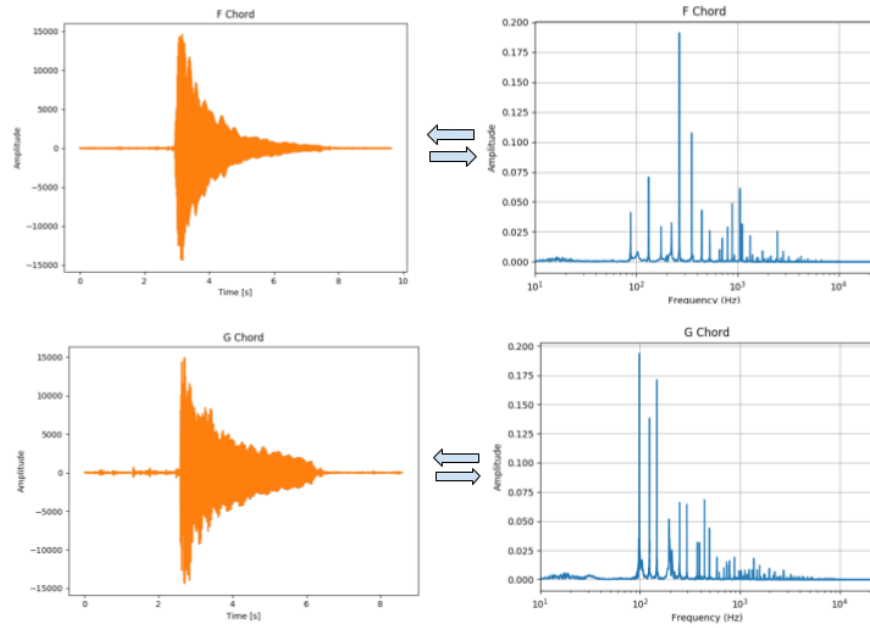


Figure 4.3

In Figures 4.2 and 4.3, the symbol \Leftrightarrow refers to the duality property of Fourier transform as we can go back and forth between time-domain and frequency-domain

$$FT[x(t)] \Leftrightarrow X(\omega) \text{ [18]} \quad \text{Equation 4.7}$$

and

$$FT^{-1}[X(\omega)] \Leftrightarrow x(t) \text{ [18]} \quad \text{Equation 4.8}$$

4.2 Referencing Guitar Notes

All fundamental harmonics of the guitar notes are held as a simple double variable. These variables are then used in an array for each string of the guitar. We have organized note storage in such a way to be easily understood by anyone looking at the code in Figure 4.4.

```
# Frequency of notes
E2 = 82.41
F2 = 87.31
F2S = 92.5
G2 = 98
G2S = 103.83
# Notes on High E string
String1 = [E4, F4, F4S, G4, G4S, A4, A4S, B4, C5, C5S, D5, D5S, E5, F5,
F5S, G5, G5S, A5]
```

Figure 4.4: Code Diagram for Note Storage

We then moved all notes into one array chromatic scale to form a complete chromatic scale of all notes on the guitar. In order for a chord to find all fundamental frequencies of the notes played you have to index the chromatic scale.

Strings on the guitar are spaced in such a way where in musical terminology every string is spaced a “fourth” apart except for the B string which is a “third” apart. What this means for our code is we can index each string based off of the location of the lowest note of the guitar and add how many “fourth” or “third” spacings we are away from it. A quick summary can be seen in the code in Figure 4.5.

```
# Complete chromatic scale for guitar
Strings = [E2, F2, F2S, G2, G2S, A2, A2S, B2, C3, C3S, D3, D3S,
           E3, F3, F3S, G3, G3S, A3, A3S, B3, C4, C4S, D4, D4S,
           E4, F4, F4S, G4, G4S, A4, A4S, B4, C5, C5S, D5, D5S,
           E5, F5, F5S, G5, G5S, A5]
# How to access notes on the chromatic scale for guitar
notes = [Strings[note_array[0]], Strings[note_array[1] + 5],
         Strings[note_array[2] + 10], Strings[note_array[3] + 15],
         Strings[note_array[4] + 19], Strings[note_array[5] + 24]]
```

Figure 4.5: Code Diagram for Note Storage

4.3 Harmonics

An array of 6 integers is generated from the android App and will be sent to the spectrum analyzer via bluetooth. This array of 6 integers contains the fret positions which first have to be converted to the indices of the fundamental frequencies f_0 that corresponds to a guitar chord, say A_m from table 4.2. The spectrum analyzer then generates another 2 harmonic indexes for each note so that every note now contains 3 indexes to search for in the FFT. These correspond to fundamental, first, and second harmonics for each entry in the array of a note. So the fundamental frequency f_0 along with the harmonics will be used to determine a played chord or note. Figure 4.6 below illustrates this process in further detail.

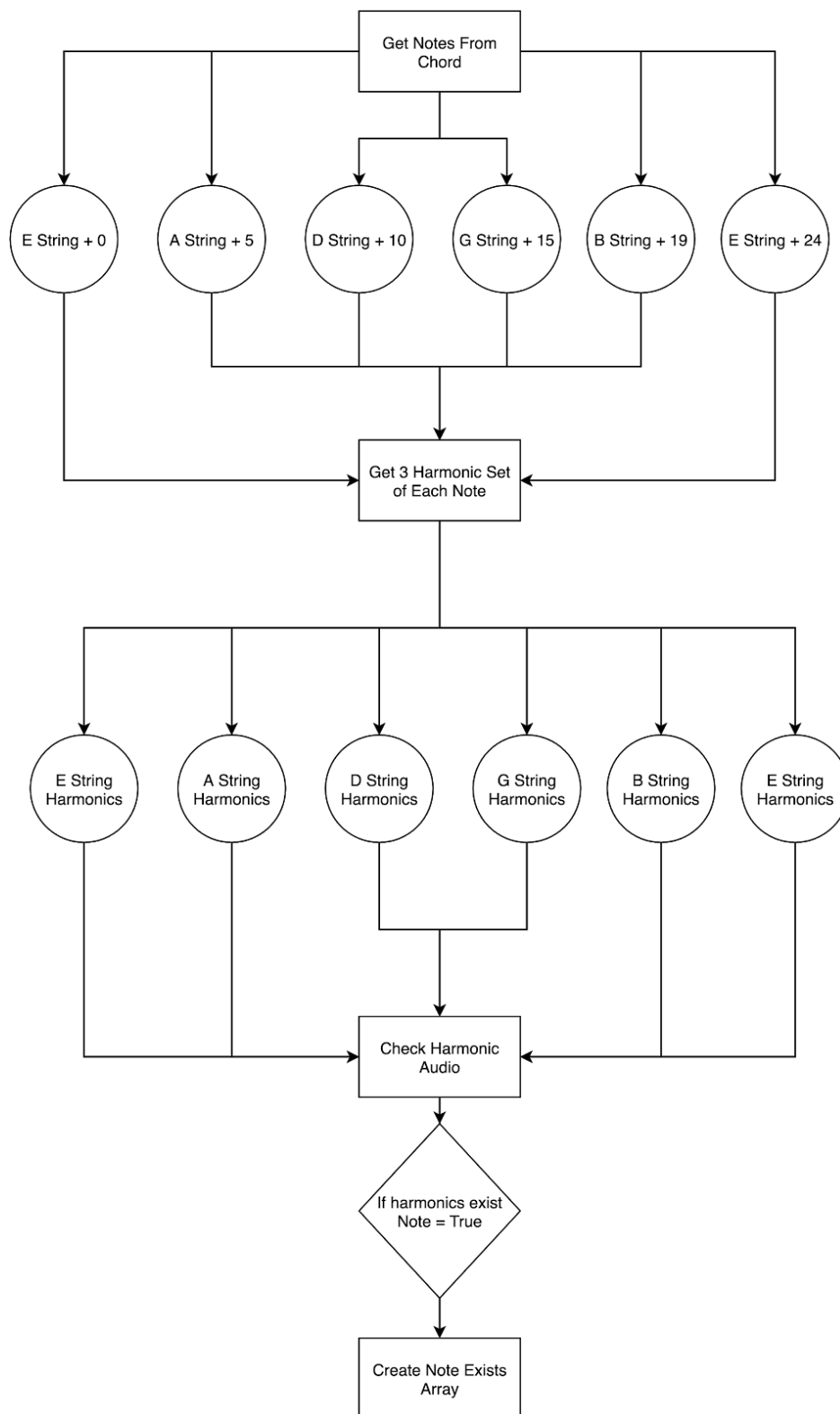


Figure 4.6: Flowchart Describing the process of making decision if the user played the right notes

4.4 Tolerance

In order to make sure, the harmonics being detected from the microphone are real harmonics and not just noise, there has been a threshold that will be used to compare how powerful those harmonics are. For better accuracy in detecting a played chord being played correct we can use the first three harmonics along with the fundamental frequency to determine if it is the chord being played is correct or not. There are three thresholds that need to be passed.

- Total power of the spectrum is great enough to be considered strummed.
- Check if power of harmonic is greater than 0.1% of the total power
- Test if the harmonic and surrounding semi semitone is greater than adjacent notes chord.

We test the total band power of a semitone surrounding the harmonic because guitars might be slightly out of tune or intonated incorrectly by up to a quarter tone and helps with accuracy of note detection. To gain a better understanding of the flow of the note testing algorithm a flowchart can be seen in Figure 4.7.

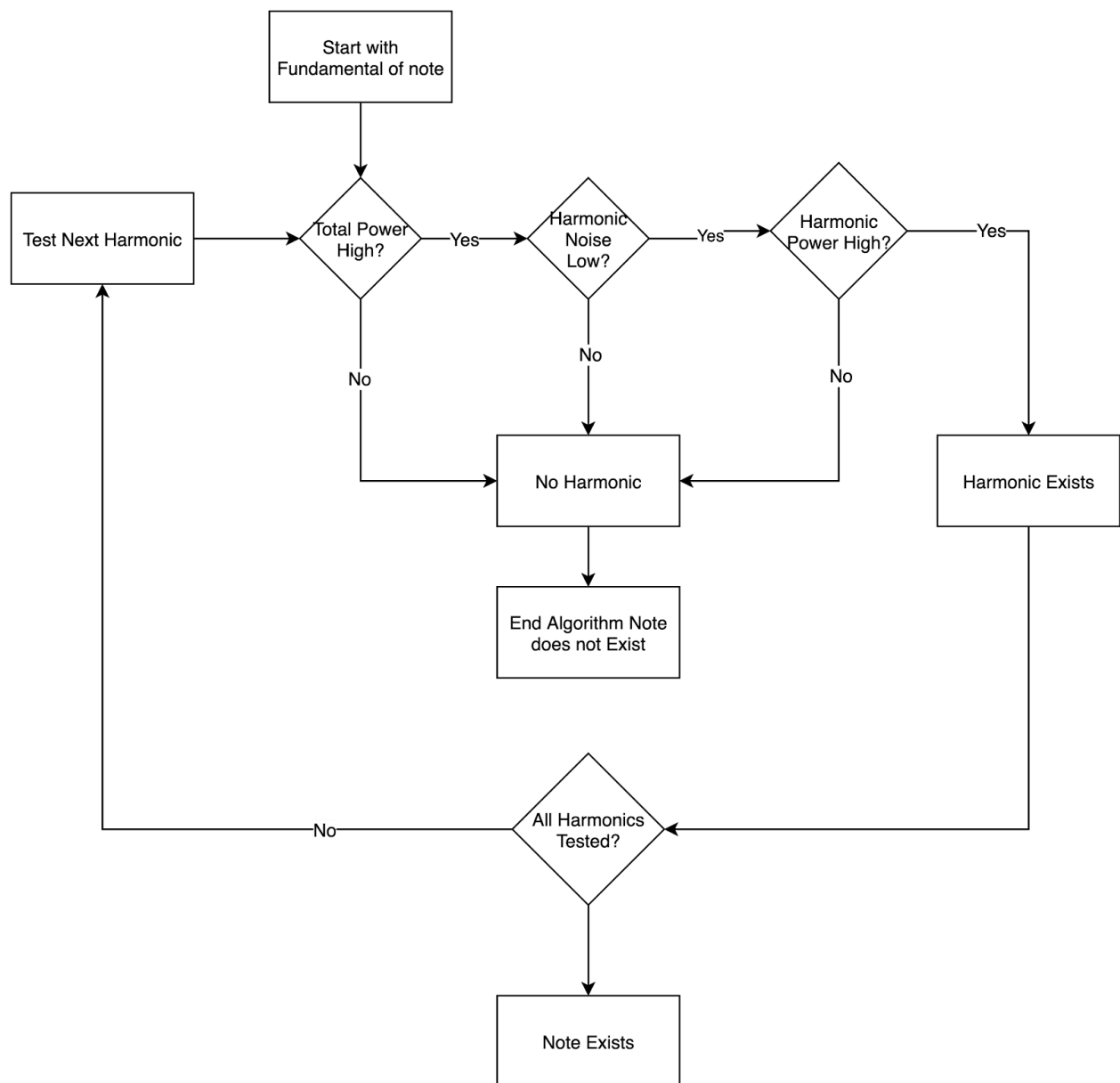


Figure 4.7: Note Decision Tolerance

4.5 FFT Buffer

In order to share the FFT data with other threads running at the same time as the spectrum analyzer we have chosen to use a shared array of data between all of them. The only time the FFT buffer is written to is after one cycle of USB reading and FFT analysis. Other threads have read-only access to help prevent data lockup and corruption throughout the run-time of the program. The only time when other threads can read the data is when the FFT buffer has been updated to help prevent reading and writing at the same time.

4.6 Power Spectrum

Power spectrum density is a technique that is used to illustrate the contributions of frequency components to the total power of the signal [15]. In other words PSD shows the most powerful frequency components of the signal. PSD is calculated by taking the square of the Fourier Transform results.

$$PSD = FT[x(t)]^2 [16] \quad \text{Equation 4.9}$$

The power density spectrum taken from the square of the FFT values refer to equation 4.9. The four figures below represent a FFT and PSD, where the PSD has a noticeably lower amount of noise.

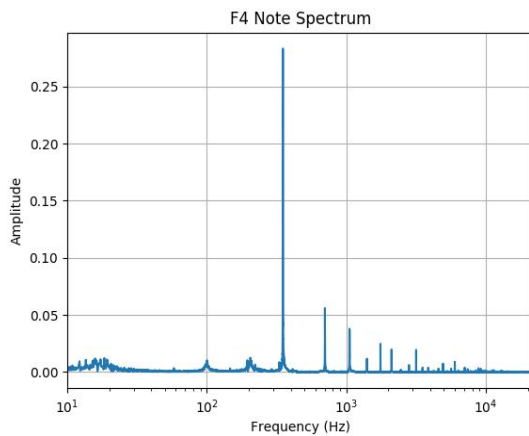


Figure 4.8 (a)

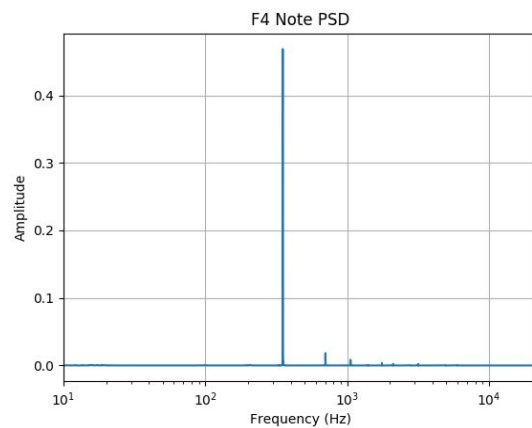


Figure 4.8 (b)

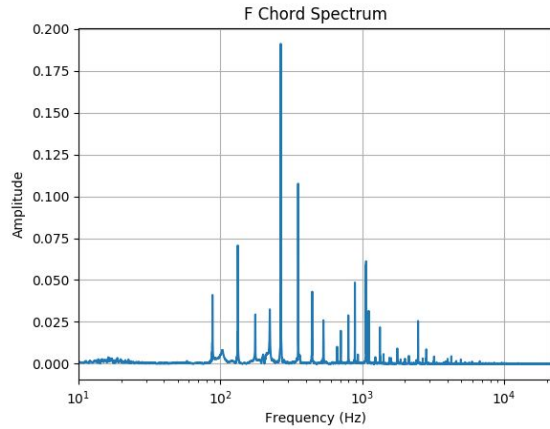


Figure 4.8 (c)

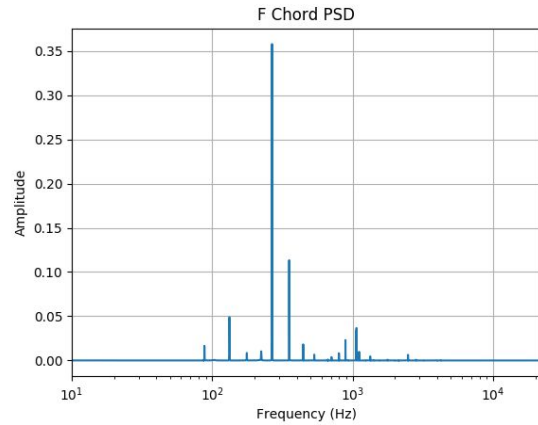


Figure 4.8 (d)

4.6 FFT vs PSD

Fourier Transform provides information about the frequency-spectrum of a time-domain signal considering the conditions mentioned in 4.1. In some situations and for the purpose of this project higher order harmonics are not needed to determine a guitar note. Even for the more complex situation when a chord is being analyzed, the spectrum analyzer we have designed is able to detect a chord is being played by only considering the fundamental frequency f_0 , first harmonic f_1 and f_2 . So there is not a need to compute higher harmonics f_3 , f_4 and moving on. Thus computing the PSD is more appropriate and fits better for the application of this project, as PSD provides better harmonics resolution and shows only the first few powerful harmonics that are necessary to determine a note or a chord.

Since PSD is taking the square of Fourier Transform in Equation 4.9, it did not seem that PSD is taking a lot more computational time compared to Fourier Transform. So we decided to implement PSD for the signal analysis in this project. Refer to table 4.3 as random computational times for both Fourier_transform and PSD have been recorded.

Table 4.3: Random computational times for Fourier-Transform and PSD

FFT computation time [ms]	PSD computation time [ms]
4.596	4.583
5.146	5.123
4.603	4.729
4.919	4.780
5.012	6.249
4.947	4.767
Average = 4.871	Average = 5.039

4.7 Overall Algorithm

As a whole there are two main threads working together to solve the signal processing problem. The first is a thread that reads a USB at a sampling rate of 44.1kHz down samples to 11.025kHz and then performs an FFT on the USB data. The FFT contains samples from four USB reads in order to increase the frequency resolution. The frequency resolution refers to the frequency spacing of the x-axis on the frequency spectrum. The overall frequency resolution for the current setup is 1.345Hz which has shown in tests to be an effective value.

The next thread is created whenever the Android device requests a beat to be detected. When this happens the note detection algorithm takes place. If the Android device requests for the notes to be detected over a specific amount of time then it detects notes multiple times. This in effect creates a periodogram of when notes start and end. The Bluetooth chapter will help explain in further detail how this works. A flow chart of the overall algorithm has been placed in the figure below to help summarize the algorithm.

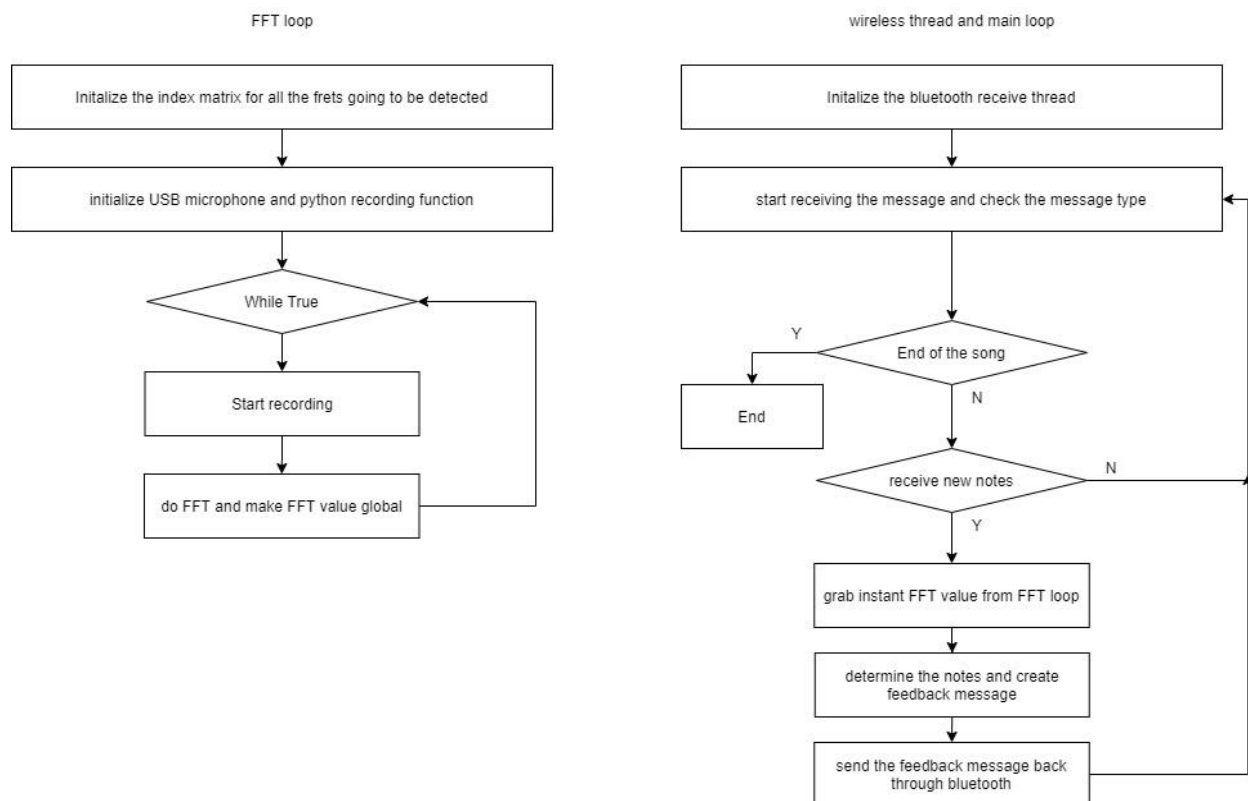


Figure 4.9

Chapter 5: Raspberry Pi to Android Bluetooth Protocol

5.1 What Is Being Communicated

5.1.1 Android

In order for the Android application to request and receive information on what is being played by the guitar a client server communication system was designed. The Android device acts as the client in this relationship and is capable of requesting three types of messages. The Android device is able to request notes that the Raspberry Pi will then monitor. The Android either asks for a continuous stream of data from the Raspberry Pi after each note calculation or a one time packet which contains the note details over time. Another request message is sent periodically in order to keep the Bluetooth connection alive.

5.1.2 Raspberry Pi

The Raspberry Pi acts as the server and as such never requests information from the Android device. Instead the Raspberry Pi waits until a data request has been sent from the Android device. Upon receipt of the message, the Raspberry Pi performs either a timed note calculation and relays it back to the Android device in one message, or performs continuous note calculations streaming the data to the device until told to stop.

5.2 Why was Bluetooth Chosen

5.2.1 Client / Server Setup

As described there is a clear client server relationship between two devices; the Android device and Raspberry Pi. Bluetooth is a convenient way of establishing a connection between the two devices without being restricted by constraints of a WiFi connection. With a WiFi connection, a unique IP address for the server is required which adds an extra level of unnecessary complication. With Bluetooth connection one can simply synchronize to any device monitoring the same port requested. Once two devices have been connected, a simple text based communication protocol can be used.

5.2.2 Error Protection

[17] There are 3 error-correction methods used by Bluetooth:

- Automatic repeat request (ARQ).
- 1/3 rate forward error correction code (FEC).
- 2/3 rate forward error correction code (FEC).

Depending on the noise level of the environment, the Bluetooth protocol is able to switch between two main modes of error correction. The first error-correction protocol uses ARQ, which allows for a TCP-like connection and checks if a packet has been corrupted. If it has it automatically requests for it again. ARQ is used in low noise environments because the overhead packet size is small and the probability of requiring the resending of messages is low due to low data corruption.

FEC can be chosen when the error rate is high enough that re-sent packages happen so frequently that they slow the datastream down too much to be useful. FEC has more packet overhead but is able to fix corrupted data upon receipt which means that packets don't have to be resent.

Error checking is a protocol based requirement which means that all devices do this automatically through the Bluetooth communication chip on the board. This simplifies the software because data can always be considered reliable and the overall algorithm does not need to worry about such circumstances.

5.2.3 Availability

Bluetooth is commonly available on both the Raspberry Pi and Android. While both devices require permission from the user to connect to devices using Bluetooth this was not found to be an issue. Once a device has been synced to the Raspberry Pi, it becomes a trusted device until removed by the user. On the Android side it is as simple as requesting permission from the user one time once the app has been installed.

5.3 Requirements For Connecting

5.3.1 Port

In order for the server to connect with a device properly, a common port needs to be selected by both the Raspberry Pi and Android device. Port 5 was selected because it is a port not commonly used by other apps on either the Android device or Raspberry Pi. While there were issues with getting the Android to use Ports instead of UUID another commonly used Bluetooth protocol meaning universal unique identifier. UUID is often used in conjunction with Android Apps to connect to multiple devices at once and acts like an IP address. It enables a randomly chosen port number to communicate with multiple devices at once. Since there is only one device that is being communicated with in the case of the Raspberry Pi, the use of the Ports was chosen because it was simpler to construct the server.

5.3.2 Text Protocol

A basic data protocol has been put in place on both the Android device and Raspberry Pi. This transmission protocol simply sends text based JSON objects where each packet is separated by the character “*” in order to easily find the beginning and end of each message. This “end” character has been chosen because none of the JSON fields use special characters and will only contain numbers or simple text. This was only necessary because the receiving thread might cut in half way between a message transmission and needs to know whether or not the data it got was a complete message or not.

5.4 Message Structure

5.4.1 JSON

Javascript Object Notation (JSON) is a program language independent data structure. This data structure allows programs to store basic data in a dictionary and allows for fast lookups using a key. Keys are used to index a JSON object and find the associated value being searched for. The advantage of using the JSON data structure is that it can be used in multiple different programming languages such as Python and Java. Although JSON can only use elementary data such as integers, Boolean and character strings, it is not necessary to transmit other high level data such as objects. For that reason JSON formatting was chosen to send information between the Raspberry Pi and Android devices.

5.4.2 Packet

In the example shown below is a sample of the message an Android device would send to a Raspberry Pi. A list of types found below describes what each message type is used for. The “type” key defines what message type it is and thus the kind of data that the message contains.

Table 5.1

Type	Name	Description
1	Keep Alive	Used to keep Bluetooth connection awake and sent by Android device
2	Timed Beat Request	Sent by Android device to request feedback from a set of six notes in a time specified by the Android app.
3	Beat Feedback	Message sent by Raspberry Pi containing the feedback from the set of notes form either type 2 or 4 messages
4	Beat Stream Request	A request from the Android device containing six notes whose feedback is to be continually streamed by the Raspberry Pi
5	Stop Stream	Used to stop the stream of beats designated by key 4

Once the Raspberry Pi determines the type of message using its “type” key, it is then able to take the notes from the “values” key. In the example shown in Figure 5.1 the values show are for an E major chord. The note calculations take place for a total of 500 microseconds found in the “duration” key. The “sequence” key is used by the Android device to tell which set of note values that were received on the Raspberry Pi have finished data processing. For example if the Android device sends the twelfth note of a song to be detected then the sequence would contain a value of twelve. Once the twelfth beat is detected it will again be packaged in a JSON object with the sequence number being twelve so the Android device knows which beat to designate the feedback too.

```
{ "SEND"  
  "type":2,  
  "values": [3,2,0,0,0,3],  
  "sequence":20,  
  "duration": 500  
}*
```

Figure 5.1

In the example shown in Figure 5.2, a response from the Raspberry Pi is depicted. Notice that it uses the same sequence number as the message above. The 3 “type” key tells the Android device that this is a data message from the Raspberry Pi containing one beat. The “value” key now contains a set of numbers; one being that the note played was correct and zero if incorrect. The Android device would then be able to update the user information that the B (5th) string was played incorrectly. In both of the examples above the “*” character is used at the end of the object. Once a “*” character reads everything between one end character and another is used as a JSON object and is checked to make sure that the object contains valid information.

```
{  
  "type":3,  
  "values": [0,0,1,1,1,1],  
  "sequence":20,  
}*
```

Figure 5.2

5.5 Raspberry Pi

5.5.1 Threading

In order for the Bluetooth receiver and data processing algorithm on the Raspberry Pi to work concurrently together, threading was used to allow for sufficient time for each algorithm. The examples below show what threads were used in coordination with the Bluetooth messaging to make everything functionable. There are three threads made to handle the Bluetooth; 1) a receiver thread, 2) timed data sending thread, and 3) a data streaming thread.

Also it is important to remember that the signal processing thread is running constantly in the background and any FFT data made from that thread is being shared by the timed and streamed threads. The limiting factor for response time is the time it takes for the USB to be read in the signal processing thread which depends on the variable setup it can be set to be between 100 to 300 microseconds. These are small with respect to the response time by the user. This “setup time” is primarily seen in the streaming thread as new values are sent every time an FFT is done. The timed thread, on the other hand, responds in the amount of time specified by the Android device.

5.5.2 Receiving Thread

The main task of the receive thread is to first listen to Bluetooth port 5 until the Android device has successfully connected. If the Android device disconnects at any point thereafter the receive thread will safely exit and reboot and then wait for a new connection. Once a connection has been made the receive data will wait for a message to be sent.

Once new data has been detected in the connection it will be validated by an error checker which first checks if it is a proper JSON object. If the JSON object does not contain all information and keys needed to process the packet then it will be rejected and ignored. As of now five possible types can be sent and received and can be seen in the Packet section in the chapter. These message types also need to contain specific information such as what beat to test for and how much time. If the Android device were to mistakenly not send such information then it will not receive a response. In the future an invalid message response should be sent in order to inform the Android device that some information was left out.

Once the receive thread safely detects the message type, it will be able to start up the necessary thread to execute the next steps. The benefit of using threads is that requests from a client are handled while looking for new messages at the same time, all the while continuing to

do the signal processing required by each request. Further detail on the workings of the receive thread are shown in Figure 5.3 below.

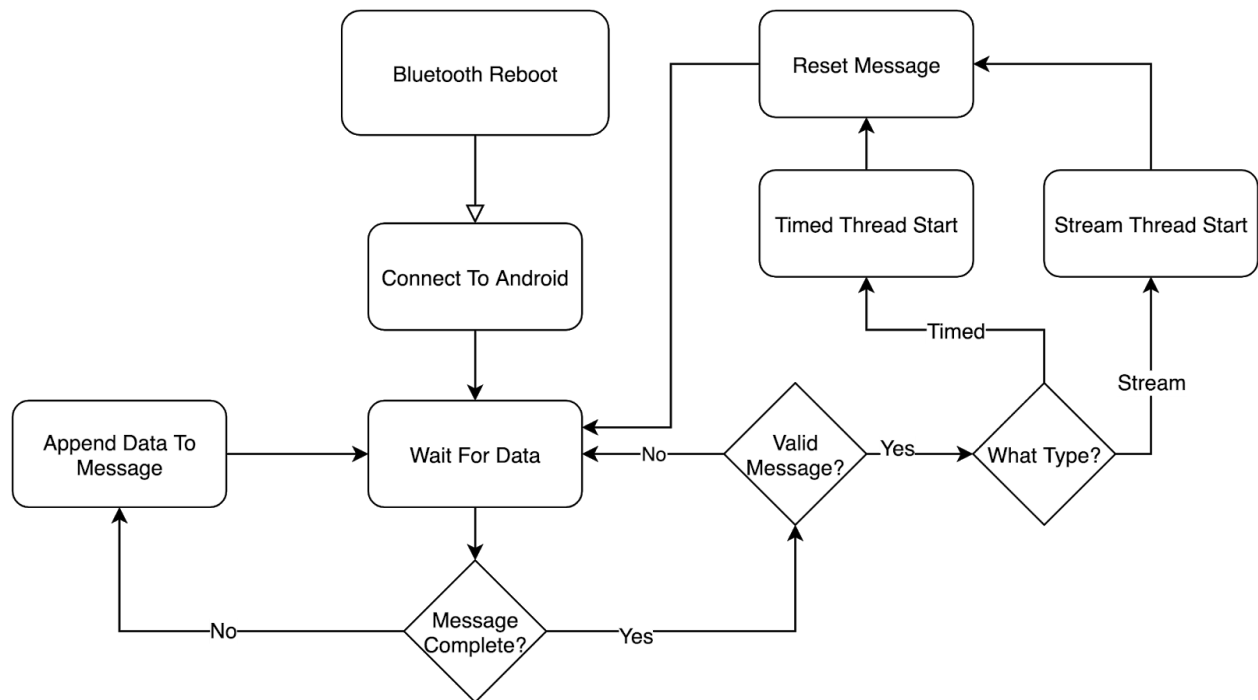


Figure 5.3

5.5.3 Stream Thread

The stream thread's main use is to be able to send the note calculations, which contain a true or false value for each string of the guitar, back to the Android device every time a calculation has been completed on the signal processing thread. This thread is able to send a response to the Android device approximately every 200 microseconds in the current implementation or every time that the USB reads new data into the buffer.

First the JSON object, received from the receive thread, is further examined by examining the notes that are to be streamed. If the stream is already on from a previously streamed notes set, then all that is required is to exchange the set of notes to be examined by the signal processing thread. Every time the FFT in the signal processing thread completed its analysis, the stream thread then takes the data and judges whether the string and note played was correct. After all strings have been tested, the Boolean array is created in which each index represents one string of the guitar. This information is then relayed back to the Android device and where it waits for the next FFT calculation from the signal processing thread.

The stream thread constantly polls the receive thread in order to know whether it should still be streaming data. Once a stop command has been issued from the receive thread the stream thread will safely be shut down and all processing ends.

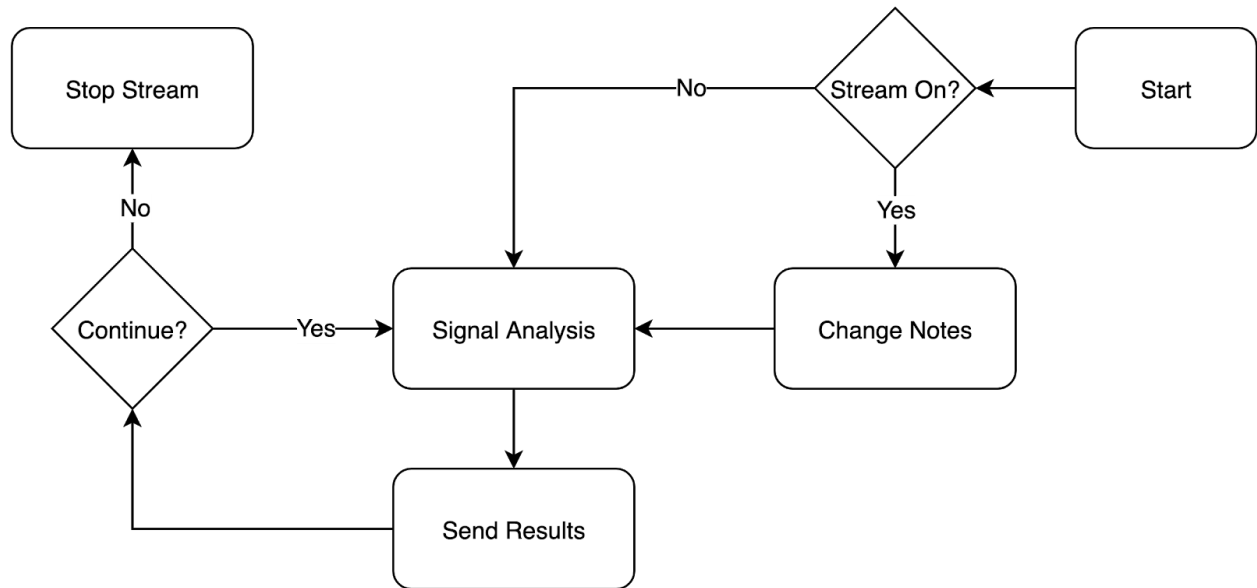


Figure 5.4

5.5.4 Timed Thread

The timed thread is meant to continually do the “note” calculations for a set period of time, as specified by the Android device. Once this time has elapsed the timed thread will send the data packet, which contains the average of the data collected by the sequence of FFT’s taken through the signal processing thread, to the Android device.

The main advantage of this thread layout is that the Android device will know approximately when to expect feedback from any given timed thread. This is useful on the Android end when the Android device needs to receive feedback within a certain amount of time. This is used when the Android app is playing a song on screen and needs to update a beat with feedback before it disappears off the screen. If a beat is updated too soon before the user gets a chance to play it then the feedback won’t be able to assist the user in note correction. If the feedback is sent too late then the user won’t be able to see the feedback and there would be no point having a signal processing unit to begin with. If the feedback is received on time then the user would be able to observe the update and understand that they did something right or wrong. Using this process the Raspberry Pi analyses the FFT over a specific period of time effectively creating a “periodogram” that was described earlier in the signal processing section.

The periodogram contains multiple FFT's spaced apart by the USB's sample rate. This periodogram allows for a more detailed algorithm to calculate whether the played notes are accurate and in what portion of time they exist.

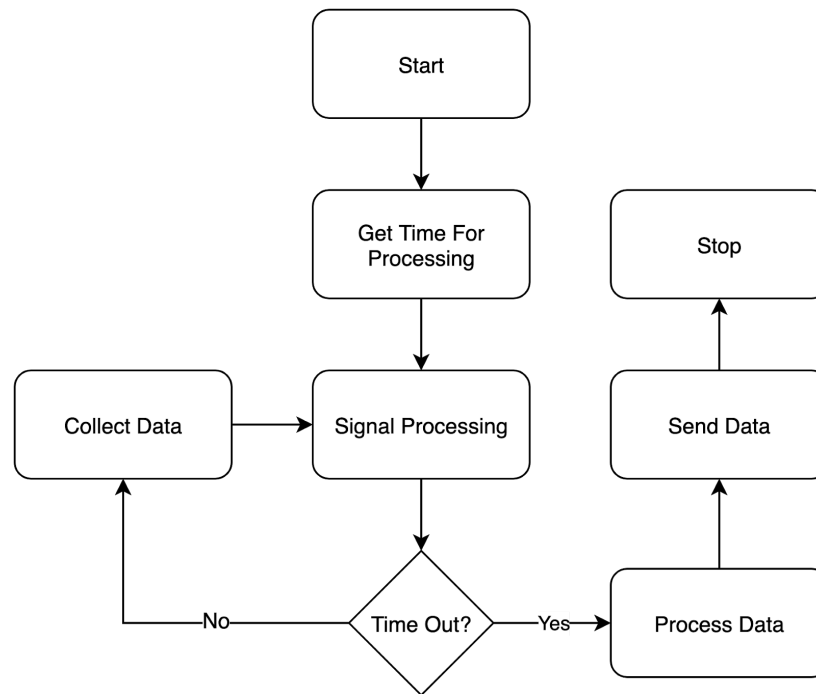


Figure 5.5

5.6 Summary

In this chapter, Bluetooth Protocol, as a main issue is completely explained in three parts: why we choose to use Bluetooth as a communication protocol, how we set up and code the program to use Bluetooth and what we send between Raspberry Pi and Android device through Bluetooth. With the method of thread, we can keep the Bluetooth connection concurrently and send/receive information between the Android device and Raspberry Pi anytime. After establishing the connection, we can move on to the Android application that lets the user control and access the training function.

Chapter 6: Android Application

6.1 Why Android

This project was developed using Android Studio. It is the official development environment for Google's Android operating system, and features a comprehensive library with documentation which is maintained by Google.

Having selected Android Studio the development environment, the supporting languages, Java and Kotlin, were evaluated. While Java, of course, is more common and familiar, Kotlin is a similar, newer language that has recently become the preferred language from android app developers. While initially there was great interest in Kotlin, for the development of the full app, Java was used in order to capitalize on the much greater supply of online support material.

6.2 Overview

The purpose of the android app is twofold. It controls the Raspberry Pi's processes via Bluetooth, and visually presents these processes as a human-usable guitar trainer. At a high level, the app is structured in the style of conventional mobile games where the user is presented with a title screen, followed by a series of options that can select a chord dictionary, songs, a tempo game and Bluetooth connection mode.

In Android Studio, an "activity" is a process that runs within an app. The currently active activity has full control of the screen. Each screen in an android app is implemented via an activity. Two fundamental game activities have been implemented. The first is a scrolling fretboard in the style of "Guitar Hero", which prompts the user to play songs in real-time and gives feedback on each note. The second is a training mode which gives the user as much time as needed to play an indicated note while displaying the required finger positions. From these fundamental game modes, more specific exercises were derived. A scrolling mode allows the user to play full songs or repeatedly practice segments of a given song. It can also be used as a tempo exercise that requires the user to play randomly generated sequences of chords and notes to work on the right hand's dexterity through speed and coordination. The training mode is used either to indefinitely practice a single chord, or to step through a sequence of chords, either manually or when a certain success threshold is met for each chord prompt. These modes constitute the tools with which a user can refine their skills with the guitar.

6.3 Android Bluetooth

The app is manually connected to the Raspberry Pi via Bluetooth at the launch of the app, after which the app controls the Raspberry Pi's processes in real-time by passing command messages. The latency in this message-passing has been determined to be small enough such that the app's control of the Raspberry Pi can be considered real-time with respect to human user response time. Commands are sent to the Raspberry Pi to trigger the FFT processes on the Raspberry Pi's audio input, as required by the current app activity. The spectrum analysis is similar enough that the different game modes can be realized by simply changing the parameters in these commands. For example, the command packets include a time duration over which the Raspberry Pi is to look for the indicated chord. For the training mode this parameter can be modified to have the search continue until the app sends a terminating command. In this way, the Bluetooth commands can be generated similarly throughout the app, allowing the Bluetooth classes to be relatively simple and largely independent of the main app.

The Bluetooth connection is established from a dedicated app screen accessible from the title screen. Though this screen initially had functionality to scan for and pair with devices, in the interests of simplicity this was removed. The android device and Raspberry Pi must be paired manually outside of the app, using their native Bluetooth functions. When done correctly, the Bluetooth app activity loads a list of devices that are Bluetooth paired with the Android device on the Bluetooth connection screen of figure 6.1 below. The user may tap a device on this list to establish a connection with it. If the connection is successful and the target device is running the associated Python code, the user is given access to the app's game modes.

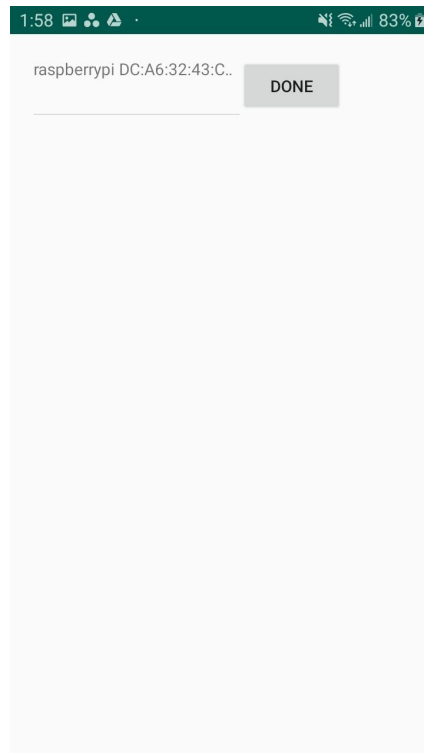


Figure 6.1

In order to maintain the Bluetooth connection across various app activities, it is placed in an “Android service”, which is a background application component accessible from any activity in the app. An Android service is a service that allows access to background processing in the operating system. When the app is interrupted by texts or other sudden notifications it continues to run until turned off by our app. This service is instantiated by the user’s Bluetooth interface activity, such that it runs as a background process once the activity is terminated. Within the Android service an instance of the `ConnectedThread` class exists, which maintains the Bluetooth connection and handles message passing.

The `ConnectedThread` contains simple methods to pass byte array messages through the Bluetooth connection, while running a pair of background processes. The first process scans the Bluetooth connection for incoming messages, delineating them into packets by searching for a specified “end of message” character, “*”, as mentioned previously. The most recent message received is stored in string character form and can be accessed by other app components via a getter method. The second process repeatedly sends a specific message through the Bluetooth connection every few seconds. This message is in the same JSON format as other messages, but contains a reserved tag that indicates that the Python code should ignore it. This message is

necessary because the Bluetooth connection would otherwise self-terminate after a period of inactivity.

This `ConnectedThread` instance is accessible to all other classes via a `BluetoothServiceInterface` class, which provides a pointer to the `ConnectedThread` for any calling app component. With this setup, all app activities have easy access to the Bluetooth connection, and the connection exists largely independently of the rest of the app. As a direct result, the connection was found to run consistently even in the presence of errors in other parts of the app.

6.4 Low-Level Implementation

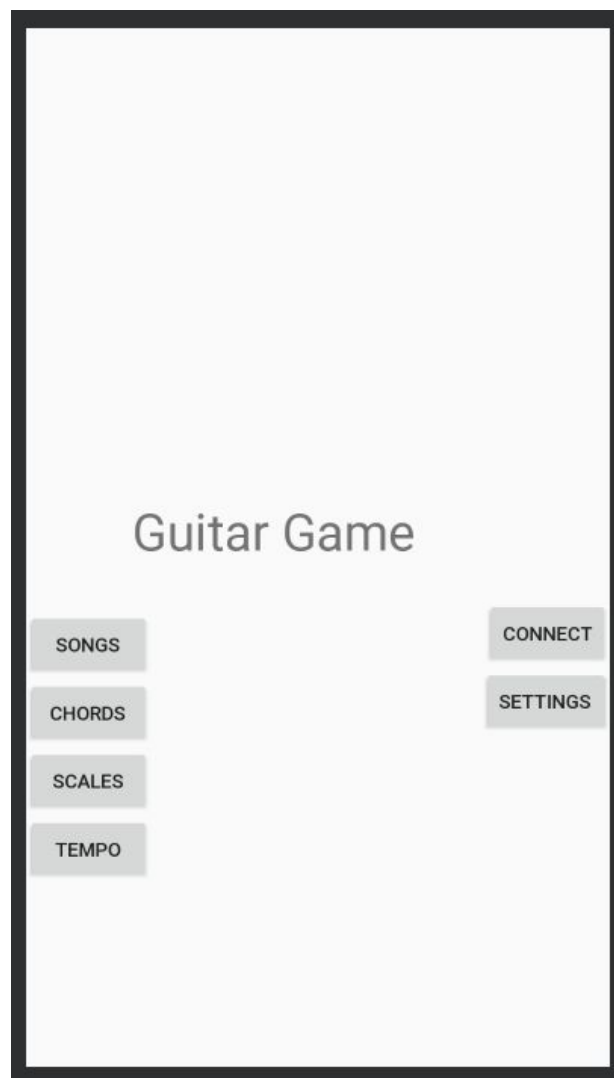
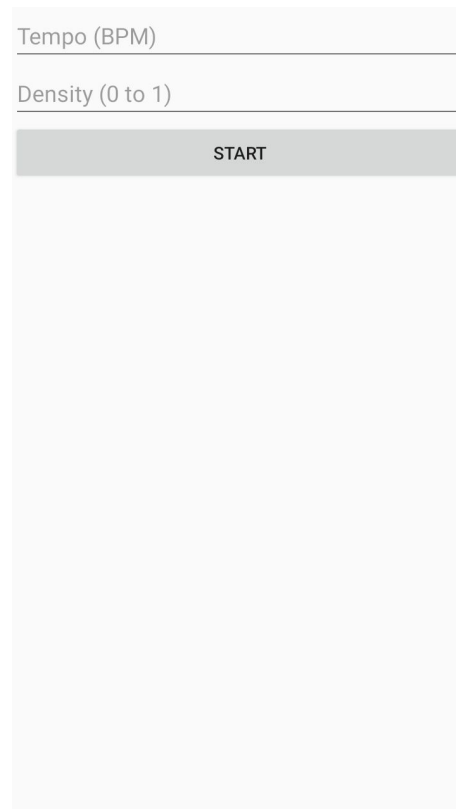


Figure 6.2

With the Bluetooth service established, the user returns to the title screen of figure 6.2 and enters one of the game modes listed above, via a series of options. The first of these options

is presented by four buttons on the left side of the title screen. The “song” option loads a new page, which populates itself with the names of songs found in the internal file “songs.json”, as well as the user’s best score for that song if a score exists, which are stored in an external file “scores.txt”. Choosing one of these songs takes the user to a new screen where they select whether to play the full song in the scrolling fretboard mode, or a segment of the song in training mode. Indices segmenting the songs are loaded from “songs.json”. A selection here initiates the chosen game activity. Algorithms for the other options are similar in that the “chords” and “scales” buttons load from JSON files of the same name and then initiate the training mode. The “tempo” button loads the screen of figure 6.3, which prompts the user for numeric parameters to use in generating a random beat sequence. The scrolling fretboard activity is then loaded with this randomly generated beat sequence.



Tempo (BPM)
Density (0 to 1)
START

Figure 6.3

6.4.1 Beats

Ultimately, the scrolling game activity or the training activity begins. The activity first extracts the full chord sequence it has been given and instantiates a list of objects of type “Beat.java” to represent the chord sequence. The Beat class is a container and controller for six instances of the “Note.java” class, such that each Note represents the note to be played on one string at a given time, and their Beat represents the full chord to be played at that given time. Notes are given an integer between 0 and 29 to indicate the necessary finger position on that guitar string, although not all are valid positions, and for simplicity only the first twelve are supported in our project. Using these Notes, the Beat draws a bitmap sprite consisting of six vertically stacked boxes that contain the finger position of the given note. This sprite is shown to the user as a prompt to play the indicated chord. There is also a reserved note value of negative two, indicating that the string should not be strummed. In this case nothing is drawn in this position on the Beat sprite.

6.4.2 Fretboard Activity

Exactly how the Beats are used differs between the scrolling game activity and the training activity. In the scrolling fretboard mode of figure 6.4 below, a fretboard is drawn on screen, and the Beat sprites scroll across the screen from right to left. Near the left end of the screen is a translucent red “trigger line” through which the Beat sprites pass. When the Beat sprite overlaps with this line, the player will be given feedback on playing that chord. When the Beat sprite first collides with the trigger line, the Raspberry Pi is prompted to search for that cord and give feedback. Once feedback is received from the Raspberry Pi, the Beat sprite is modified such that if a note was played correctly, the corresponding box in the Beat sprite is redrawn with lowered opacity, and is thus visually faded. The entire sprite fading is an indication that the chord was played correctly, while the sprite remaining unchanged is an indication that none of the component notes were detected by the Raspberry Pi. The Beat sprites continue to traverse the screen in this manner until the end of the song, or indefinitely if the previous menu selections set a flag to repeat the song endlessly.

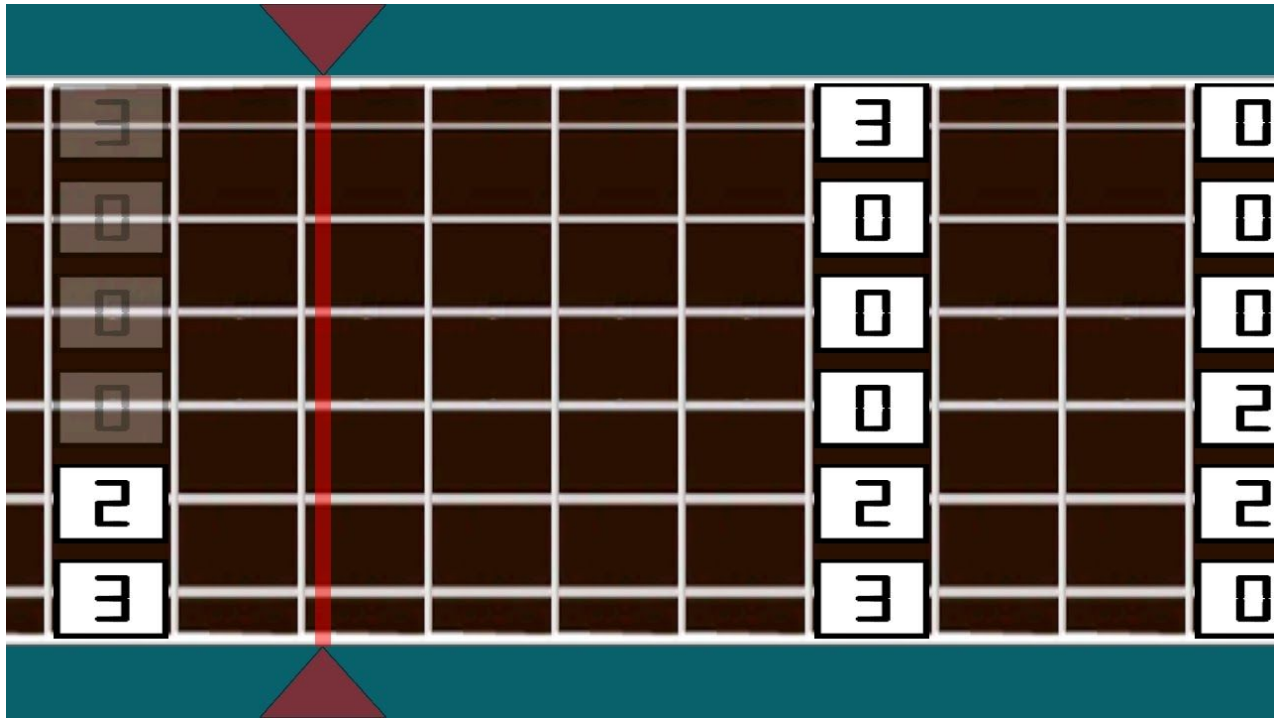


Figure 6.4

The rate at which the Beats traverse the screen is a constant value, determined by a function of the rendering speed of the app, the width of the Beat sprite in pixels, the amount of empty space between each Beat sprite, and the tempo of the current song in beats/minute or BPM for short. In a similar manner, the duration of collision between a Beat sprite and the trigger line is known, and is used as a parameter in the command sent to the Raspberry Pi. When a Beat sprite first touches the trigger line, the Raspberry Pi is instructed to search for that chord, for a duration equal to the duration of the aforementioned sprite collision. At the end of this duration, the Raspberry Pi sends back the results of its search. The feedback is a set of 6 Booleans values, indicating whether each note was found by the Raspberry Pi. On each redraw, the activity checks the latest message received by the `ConnectedThread` and sends it to each Beat that has touched the trigger line and not yet received feedback. The Beats and messages are indexed such that a feedback message is applied to the correct beat. This system was designed to handle multiple Beats awaiting feedback, but in practice the very small Bluetooth latency ensures that there will only be one Beat awaiting feedback at any given time.

At the end of a non-repeating song in the scrolling game mode, the ratio of successes to failures in received feedback is summed and used to compute a score for that attempt. The score is a linear percentage of individual notes played correctly. This score is displayed to the user and

saved to an external file called “scores.txt” in JSON format. Following this, the game activity terminates and the user is returned to the title screen.

6.4.3 Training Activity

In the training activity of figure 6.5 below, the current Beat is stationary, and drawn on the left side of the screen. The note values of this Beat are used to compute and draw a fretboard sprite, on which finger positions are indicated. The full guitar neck is not drawn. Only five contiguous frets are drawn, but they are identifiable because they include the same dot pattern present on that segment of the guitar neck. In conjunction with the numbers in the Beat sprite, this is sufficient information for a user to follow the indicated finger positions. However, drawing only five neck segments means that only certain finger placement combinations can be depicted, but these represent the most common and simplest chords.

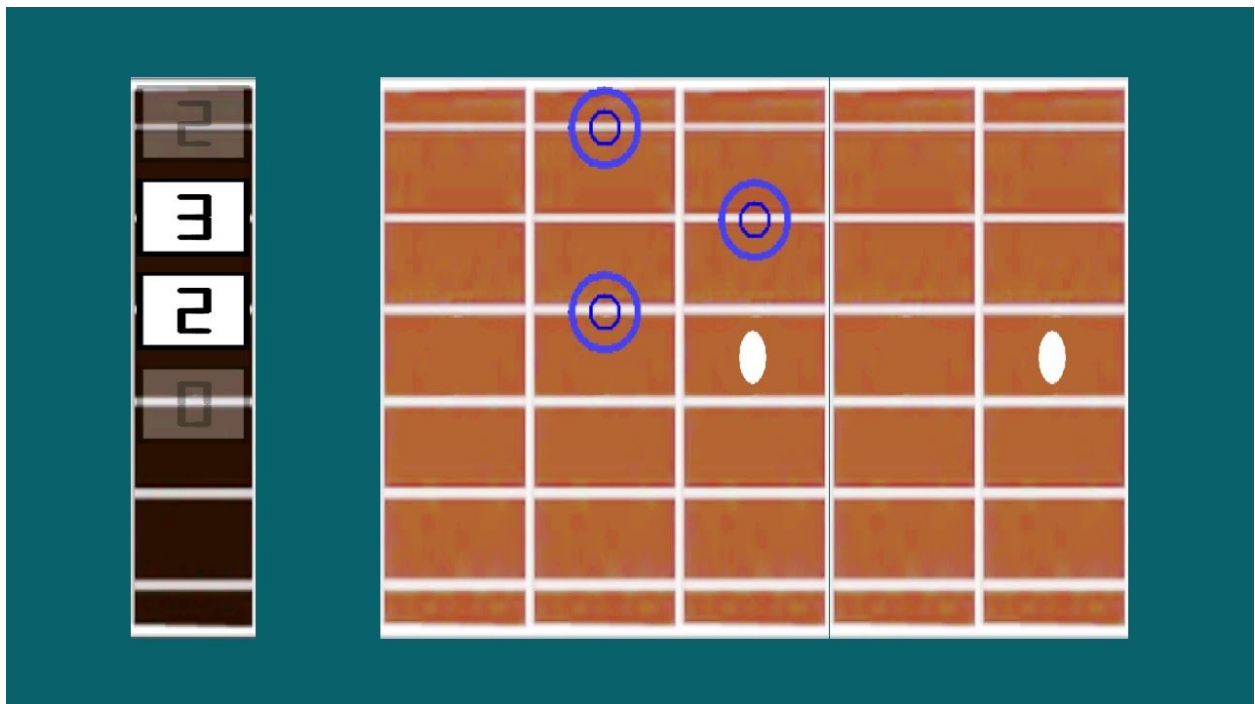


Figure 6.5

This Beat sprite displays feedback in the same manner as in the scrolling game activity, though the timing works differently. The Raspberry Pi is given a single command to search for this chord indefinitely, and returns the feedback at a high rate. The Beat sprite therefore visually updates almost immediately as the note is played, then resets itself once the note is no longer being played. Depending on an app setting, the user may either manually advance to the next Beat via an onscreen button, or the chord sequence progresses automatically when a certain

success threshold is met. In either case, the Raspberry Pi is given a command to terminate its search. The app then loads the next Beat and repeats this process, redrawing all sprites and sending another command. This continues until the training activity is manually terminated.

6.4.4 Settings

The final button on the title screen is the “settings”, which gives the user the following options as shown in table 6.1. These are stored in an external file “settings.txt”.

Table 6.1

Setting	Function
Tempo Override	When enabled, forces the tempo of the scrolling fretboard activity to a user-specified value
Always Training Mode	When enabled, training activity will run instead of scrolling fretboard activity
Manual Training Mode Progression	When enabled, the training activity will only progress through beats via button press
Unlock All Songs	When enabled, songs are no longer locked initially

Collectively, these app activities implement the various screens used by the guitar training app.

6.5 Sprites

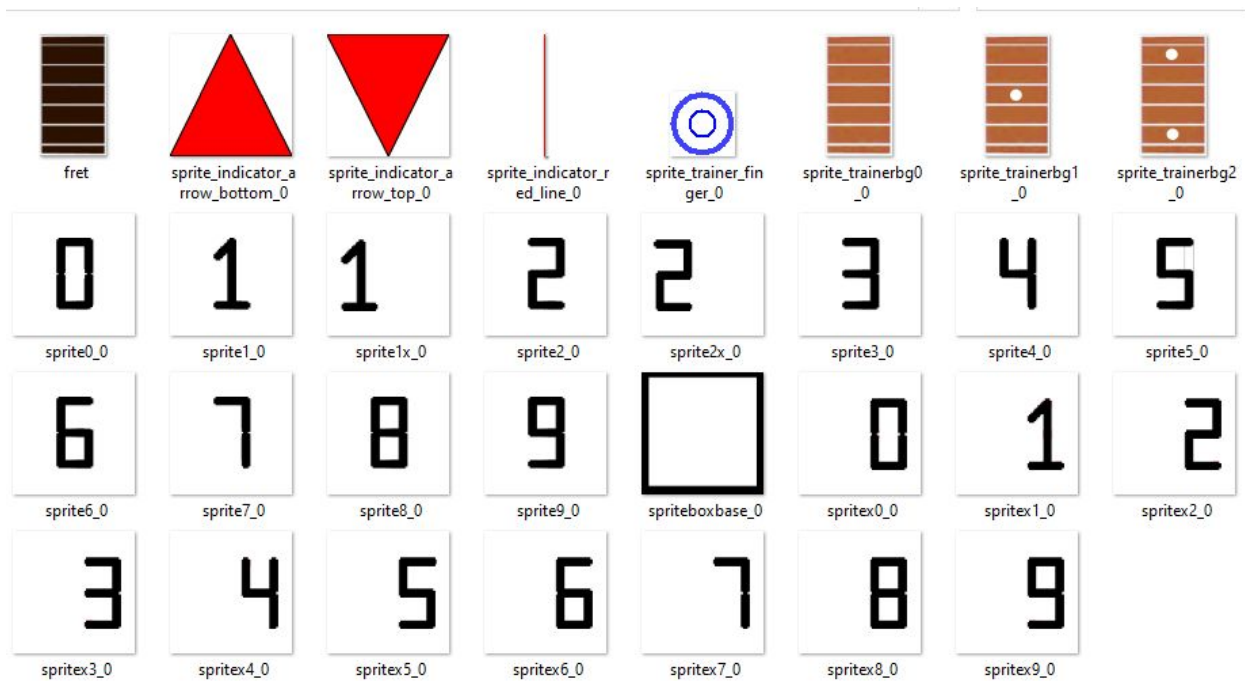


Figure 6.6

Figure 6.6 above shows the bitmap sprites used to construct app visuals. The number sprites are handmade so that they can be placed atop one another to create properly-aligned numbers 0 to 29. The “_0” at the end of their names is present because they were created in GameMaker Studio, which indices sprites with the assumption that there will be multiple images forming each animation.

The fundamental element of the visuals in the app activities is the bitmap sprite. Using a small number of basic bitmaps included in the app’s files, the app algorithmically constructs the visuals shown to the user. Composite sprites are created by drawing a pair of bitmaps onto a Canvas object, then converting that Canvas into a sprite. This basic method is extended with position parameters in some cases so that sprites can be merged with a positional offset relative to one another. The basic version of the function is shown below in figure 6.7.

```
private Bitmap overlay(Bitmap bmp1, Bitmap bmp2) {
    Bitmap bmOverlay = Bitmap.createBitmap(bmp1.getWidth(),
    bmp1.getHeight(), bmp1.getConfig());
    Canvas canvas = new Canvas(bmOverlay);
    canvas.drawBitmap(bmp1, new Matrix(), null);
    canvas.drawBitmap(bmp2, new Matrix(), null);
    return bmOverlay;
}
```

Figure 6.7

There are three complex objects that are constructed from these sprites; beats, the fretboard of the scrolling game activity, and the fretboard of the training activity on which finger positions are shown.



Figure 6.8

The Beat sprite, shown in figure 6.8, is assembled by using the Canvas method to build sprites for each of the six individual note blocks. These are then stacked vertically to depict the full Beat. The Note first loads a box-shaped background, then combines it with one or two custom-made numerical bitmaps, as determined by the value of the Note. These six Note sprites are then drawn onto the Beat sprite using a modified version of the above function that places them in the correct vertical positions. When feedback is applied to the Beat, the Beat sprite is generated anew, with each Note sprite drawn with reduced opacity if the given feedback

indicated that the Note had been played correctly. This new Beat sprite is immediately assigned to the Beat object, so that the Note sprites update in the same draw cycle.

The fretboard, on which these Beats are drawn, is not a single sprite but a horizontal sequence of full Beat sprites. The Beat sprite described above begins with a “fret” sprite, onto which the six notes are drawn. An empty space is inserted into the song by adding “empty” Beats and drawing their sprites. The fretboard is a continuous chain of Beat sprites that scrolls along the screen. A constant integer `fretsOnScreen` determines how many of these Beats are drawn at once. This constant is set as $N+1$, where integer N is the pixel width of the Android phone screen divided by the pixel width of the Beat sprite, rounded down. When a Beat sprite has fully traversed the screen and is no longer visible, it is removed and another Beat sprite is added to the beginning of the fretboard. This “treadmill” algorithm ensures that the fretboard looks continuous at all times, and crosses the width of the Android phone screen. It also ensures that the minimum number of sprites is rendered at any time. Although the corresponding Beat objects are instantiated when the activity is initially loaded, their sprites are instantiated only when the Beat is placed on the fretboard “treadmill”. The sprite is then released once the Beat has fully traversed the screen. Using this paradigm a noticeable improvement in load-times was found when compared to those that preloaded sprites for the full song. The difference was up to twenty seconds in normal testing conditions.

The final component of the fretboard is the translucent red line that represents the prompt to play a beat. This is drawn above the fretboard at a constant position on each draw cycle. As described previously, these sprites make up the entire visual component of the scrolling game activity.

The training fretboard, while visually similar to the scrolling fretboard described above, is quite different in its operation. The current Beat in the training activity is visually depicted by algorithmically combining a number of bitmaps. This process repeats when the Beat changes. The foundation of this sprite is five sections of a modified version of the “fret” sprite. This sprite has three variants: zero, one, and two dots, which depict the dots of the neck of a guitar. These are combined to construct a representation of the relevant section of the guitar neck. With the assumption that all finger positions in a chord are reasonably close to one another, the lowest finger position present is found first, and the neck sprite is constructed based on a hard-coded description of the number of dots on a given fret.

Following this, blue “target” sprites are algorithmically drawn on the “neck” sprite to indicate the finger positions. The modified UpdateBitmap function is used, with the coordinates that are determined by the string and neck position for each target sprite. In the manner described above, the visuals for these app activities are assembled from a small number of component bitmap sprites.

6.6 Data structures and Memory I/O

To maintain consistency throughout the project, the JSON format described above is used for all app I/O functionality. As well as for Bluetooth messages, this format is used for both storage of static resources and external user-state files. The songs, chords, and scales used by the app are stored in respectively-named JSON files, and extracted by the app at runtime. These JSON files are composed of JSON objects representing each chord sequence, with additional fields for other information, as shown in figure 6.9 below.

```
"Island In the Sun": {  
  "song": [  
    [-2,-2,-2,4,5,3],  
    [-2,-2,-2,-2,-2,-2],  
    [-2,-2,-2,4,5,3],  
    [-2,-2,-2,4,5,3],  
    [-2,-2,-2,5,5,5],  
    [-2,-2,-2,-2,-2,-2],  
    [-2,-2,-2,5,5,5],  
    [-2,-2,-2,5,5,5],  
    [-2,-2,-2,-2,-2,-2],  
    [-2,-2,0,2,1,2],  
    [-2,-2,0,2,1,2],  
    [-2,-2,0,2,1,2],  
    [-2,-2,-2,4,3,3],  
    [-2,-2,-2,-2,-2,-2],  
    [-2,-2,-2,4,3,3],  
    [-2,-2,-2,4,3,3],  
    [-2,-2,-2,-2,-2,-2]  
  ],  
  "Parts": [  
    [0,5],  
    [6,10]],  
  "partNames": ["part1","part2"],  
  "bpm": 120 }
```

Figure 6.9

Above is the segment of “songs.json” representing “Island in the Sun”. The integer arrays within “song” represent finger positions for each beat of the song. Values of -2 indicate that the string is not strummed in that chord. For example the first beat of the song contains [-2, -2, -2, 4, 5, 3]. The low E, A and D string should not be strummed, while the 4th fret of the G string, 5th fret of the B string and the 3rd fret of the high E string should be strummed. The second integer array consists of six -2 values, indicating an empty beat.

Similar formatting is used for the external files used to store user settings and user scores. This memory I/O is handled by a static class `MemoryInterface` with basic functionality. When a value from memory is needed, the `MemoryInterface` class reads the full file into a JSON object, then extracts the required field. Since JSON objects are immutable, the `MemoryInterface` class modifies the file by reading in the external file and creating a new JSON object with the read values and any modifications needed. It then overwrites the external file. Since the Android app requires permission for memory I/O, the `MemoryInterface` class prompts the user for permission via a popup. Once permission is granted, it is maintained and does not need to be requested again in a session.

6.7 Portability

The app was developed and tested on three similar Android Galaxy-series phones, running APIs between API 24 and API 28. Android is theoretically portable across many devices, but ensuring this is out of the scope of the project. The custom bitmap sprite visuals include code that should automatically rescale the pixel-size of each sprite relative to the pixel-size of the screen. Therefore the image composition of an app state should be consistent across devices with varying screen sizes and resolutions. However, this has not actually been tested and likely needs to be refined. While the app could be generalized and ported without serious issues, this will not be attempted in the scope of the project.

6.8 Education

The full system presents a user with a set of tools with which they can refine their guitar skills. The training and tempo activities present instruction and rapid feedback for mechanical skills. The scrolling game activity evaluates performance over the course of a song. Together, these systems constitute an effective training tool for a player with some prior guitar knowledge.

A beginner is not expected to be able to learn guitar solely with this system. The instructional interface relies on a small amount of background knowledge to interpret the slightly

abstract prompts given, such as the sextex of integers representing finger positions for a chord. While a fully inexperienced user may manage to interpret this from various interface aspects, we were unable to verify this experimentally. There is also practical knowledge, such as guitar tuning, bending notes, sliding down the fretboard and strumming which are not addressed within the app. As the app is meant to function as a practice tool rather than a tutorial for complete beginners, such instruction is considered outside the scope of the project.

In order to give users some structure, a simple progression system requires users to unlock each individual song by playing its component segments in the training activity. Each song is split into two or more segments, as depicted in figure 6.11 below. When a song is selected, the final configuration prompt is whether to play the full song in the scrolling game mode, or segmented in the training activity. Initially, these options are disabled with the exception of the first segment. Completing this segment unlocks the next, and so on, until all are complete and the full song option is unlocked. Since chord sequences in the training mode repeat indefinitely, a screen prompt within that activity indicates to the user when the segment is considered complete. Although there is no explicit threshold for success of a segment, the training activity will only progress through a chord sequence once a certain success threshold is met on the current chord, so this threshold must be met for each chord at least once. The status of this progression system is stored in JSON form in an external file, in the same manner as the external files described above. This progression system can be manually disabled from the Settings screen for demonstration purposes.

1:49 84%	1:54 83%
America - Horse With No Name* Highscore: 0	Full Song
C Major Scale Sequence Exercise* Highscore: 0	Verse1
C Major Scale Sequence Exercise 2* Highscore: 0	Chorus 1
Dexterity Exercise #1* Highscore: 0	Verse 2
Dexterity Exercise #2* Highscore: 0	Chorus 2
G Major Scale Sequence Closed* Highscore: 0	Verse 3
G Major Scale Sequence Open Strings* Highscore: 0	Chorus 3
Happy Birthday* Highscore: 0	
House of the Rising Sun - Animals* Highscore: 0	
La Bamba* Highscore: 0	
Old MacDonald Had a Farm* Highscore: 0	
Otherside - Red Hot Chili Peppers* Highscore: 0	
Put Your Head On My Shoulder* Highscore:	

Figure 6.10

Figure 6.11

While the app’s educational systems are fairly hands-off, this has the advantage of allowing users to practice skills at their discretion with minimal distractions. The progression system ensures that users cannot access songs they are unprepared for, and gives users the concrete goal of unlocking songs. Following this, the scoring system provides a long-term goal and measure of progression. Provided the user is not a complete beginner, this system provides a means to indefinitely refine basic guitar skills.

6.9 Summary

As described above, the Android app controls the spectrum analyzer run on the Raspberry Pi to present a tool for practicing the guitar. This app is manually connected to the Raspberry Pi via Bluetooth, and controls the associated Python code by passing messages over the Bluetooth. This Bluetooth connection is maintained in an “Android service” and runs largely independently of the other app processes. The app has a real-time scrolling fretboard activity for

playing full songs, and a training activity that instructs the user to play specific notes. App visuals are constructed by algorithmically combining a number of simple bitmap sprites. A simple progression system locks each song until the user plays its constituent parts in the training activity. While the app's education value is not to the standards that we would like all backbone components function properly and have been developed in such a way to be easily expandable if more educational content was added. A multitude of songs have been put in the app which could potentially help with memorization and finger dexterity.

Chapter 7: Testing and Validation

Three different types of tests were undertaken. These included single note comparison, chord detection and different guitar tests that were taken to prove that the program can give the user accurate and reliable feedback. With these three types of step by step tests, the code algorithm and the accuracy of the feedback was improved. The system worked best on an electric guitar rather than an acoustic guitar.

7.1 Single Note Comparison

The system performed very reliably for instances where only a single note detection was required. This was a result of single notes not having any interfering harmonics as opposed to chords where the octave and fifth notes interfere with the root. One single note requires that the fundamental frequency and up to four harmonics must exist in the spectrum. Through testing this algorithm, the only factor that seemed to affect if a note failed or not was if it was properly tuned.

7.2 Chords

The extraction of the frequencies for an entire chord has proven to be more of a challenge for the system to analyze. This is due to the multiple notes (up to six, with all their harmonics) that were present. For every string played there exists a multitude of harmonics that may interfere with other expected frequencies. For example, the G chord requires the following notes.

Table 7.1: fundamental, first harmonics and second harmonics of G chord

Guitar chords	Corresponding Chord Notes	Corresponding Chord Frequencies (Hz)
G	$[G_2, B_2, D_3, G_3, B_3, G_4]$	[98.00, 123.47, 146.83, 196.00, 246.94, 392.00]
G Harmonic 1	$[G_3, B_3, D_4, G_4, B_4, G_5]$	[196.00, 246.94, 293.66, 392.00, 493.88, 783.99]
G Harmonic 2	$[D_4, F\#_4, A_4, D_5, F\#_5, D_6]$	[293.66, 369.99, 440.00, 587.33, 739.99, 1174.66]

As shown in Table 7.1, the fundamental frequencies of the notes on the higher strings are also present as harmonic frequencies of lower strings. This can result in false positive results for chords played if the higher strings are not strummed. This is because the frequencies of the strings not played will be present, albeit at a much lower amplitude, in the harmonic frequencies of the lower strings.

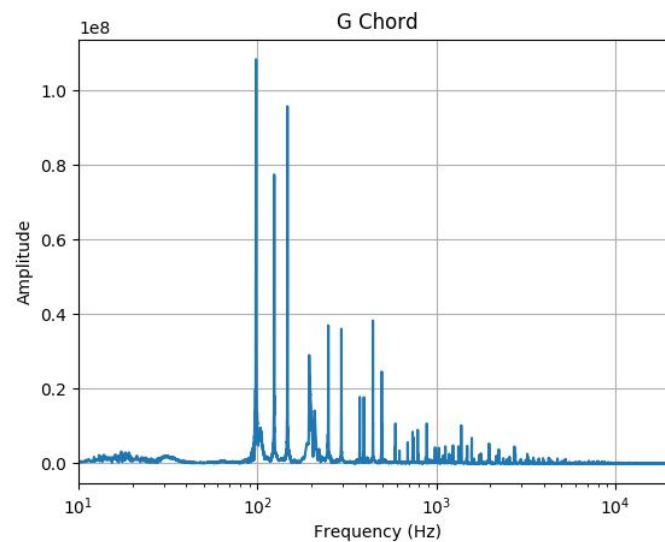


Figure 7.1 : G chord in frequency domain

7.3 Acoustic Vs Electric

The amount of noise present in the signal also affected the note evaluation step. Since the FFT normalizes the magnitudes of its frequency components, any undesired noise present could contribute to lowering the amplitude of the desired frequency such that it would be less than the threshold required for it to pass. There is more noise present when using an acoustic guitar because a microphone was used rather than the pickups of the electric guitar. In Figure 7.2 the frequency components are shown for the acoustic guitar cases where no notes were being played where the low frequency noise exists. When a D3 was played on an electric, the fundamental and harmonics were quite noticeable, as seen in Figure 7.3, as is the lack of low frequency noise.

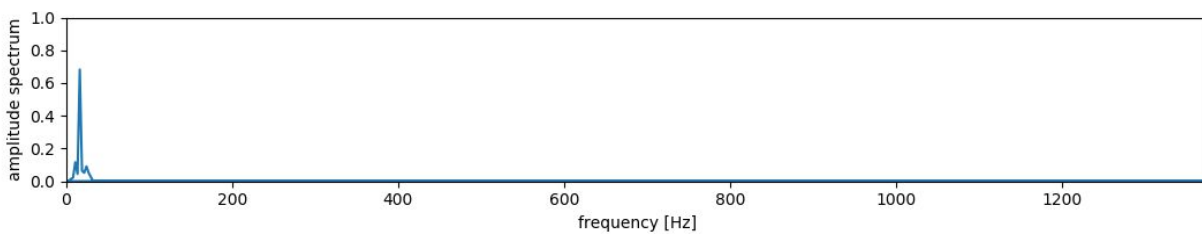


Figure 7.2 : Frequency spectrum of noise from a microphone

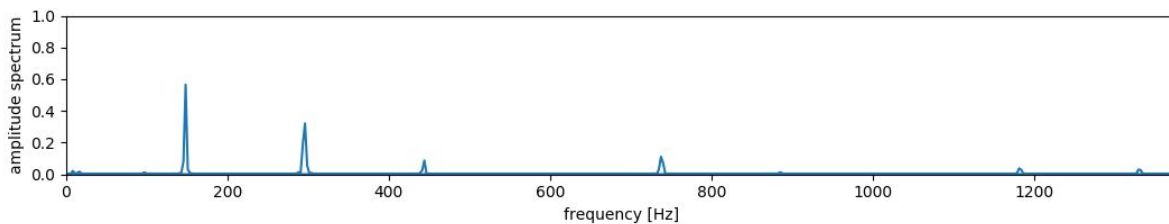


Figure 7.3 : Frequency spectrum of a D3 played on an electric guitar

Chapter 8: Future Work And Conclusion

8.1 Future Work

There are four core aspects of this project that could warrant further development. These include the app's technical mechanics, the app's visuals, the Python spectrum analyzer, and the educational systems.

Although the app is functional, it was this team's first serious Android Studio project, and therefore perhaps lacks some of the refinement of a professionally designed Android application. Ensured portability across devices is one such feature that would improve its operability. More generally, professional development would increase the app's performance and robustness, which are traits which were not optimized within the scope of this project. However, similar work has all been done before. For professional Android developers, most technical improvements would be a series of previously solved problems.

Similarly, while the app's visuals could be greatly improved, many other apps have explored this design space. Attempts to improve the visuals would ultimately mean obtaining higher-quality art assets, which was beyond the design scope of this engineering problem. Artistic or stylistic refinements that could have been implemented with the current resources would have required graphic design skills beyond that of this team.

While there are some aspects of the spectrum analyzer that could be refined, this team does not currently have the detailed requisite knowledge nor experience to do so. As a result, any improvements to the quality of feedback, such as including more frets or differentiating the harmonics, was not possible at this time.

The educational side of the system contains much more potential for further work. There are many potential high-level systems that could guide a user through the app's various exercises and songs. These could include text-based instruction and context for activities. While such a system could be implemented with the structural tools present in the app, this project focused more on developing and implementing the training tools than utilizing them.

Regarding more specific educational features, there were three main aspects that were missing throughout the education section of the app such as left and right hand exercises, strumming practice and a link between the progressions of the user for each song. With the

addition of these three subjects to the app it would make it a much more powerful tool for learning.

In terms of left and right hand exercises, it was originally intended to add games that specifically focus on practicing the skills for one hand. This would have been done first for the left hand working on basic up and down picking focusing on rhythm and speed. The next step for the left hand would be to work on common strumming patterns for chord progressions that way it would help with the users comfort level playing new songs. To make the user's right hand more adept at moving around the fretboard basic finger exercises, focusing moving one finger at a time and later progressing to moving slowly between common chords to improve switching speed would have been desirable.

Since strumming requires coordination between both hands this would be the next logical thing to incorporate into this app. In order to make the user more adept at strumming, strum directions could have been added to a common chord progression tab. This would start slowly, going through each chord one by one and picking up the speed as the user progresses.

Finally it would have been an improvement to add text lessons and descriptions to each song to teach the basics of what is happening and provide helpful tips to the player. Along with that a song by song progression could have been added in order to make the user feel like they are leveling up throughout the progression of the game.

These items were not added to the game because of lack of time and perhaps some lack of foresight into design requirements. This project was focussed on making the backbone of an app, which could give feedback to the user in real time, which was discovered to be not such an easy task to accomplish. Now that the backbone has been set in place it will be easier to add these aspects in as they come up through development because the app has been designed in an expandable way.

8.2 Summary

The goal of this project is to develop a tool to help someone improve their skill with the guitar. The tool comes in the form of an application that can be downloaded onto an Android powered device such as a smartphone. The application prompts a user to play guitar chords and displays feedback for each note. A Raspberry Pi, controlled by the app via Bluetooth, performs a series of calculations on the audio signal from the guitar strings to determine if the expected frequencies are present. After this evaluation, the results are sent back to the Android device via Bluetooth and displayed to the user.

The program provides instruction and evaluation for the first five frets of each string on the guitar, enough for many chords and full songs to be potentially playable. A more advanced educational system has not been implemented due to time constraints. The app contains a library of songs, chords, and basic finger dexterity exercises. The spectrum analysis works consistently within the supported fret range, and the app provides correct feedback in real-time relative to the human user.

Reference

- [1] Encyclopedia Britannica. (2020). Octave | music. [online] Available at: <https://www.britannica.com/art/octave-music>.
- [2] M.basicmusictheory.com. (2020). basicmusictheory.com: C major triad chord. [online] Available at: <https://m.basicmusictheory.com/c-major-triad-chord>.
- [3] Back 2 Guitar I. (2020). Comment Lire une Tablature de Guitare ? [+ de 60 Symboles Expliqués]. [online] Available at: <https://back2guitar.com/lire-tablature-guitare>.
- [4] Guitar Lesson World. (2020). Parts of a Guitar - Learn the Guitar's Anatomy | Guitar Lesson World. [online] Available at: <https://www.guitarlessonworld.com/lessons/parts-guitar-learn-guitars-anatomy/>.
- [5] Stringjoy. (2020). Coated Guitar Strings: Are They Right For You? | Stringjoy. [online] Available at: <https://www.stringjoy.com/coated-guitar-strings-right/>.
- [6] Long-mcquade.com. (2020). Strings Attached - A Guide to Acoustic Guitar Strings. [online] Available at: https://www.long-mcquade.com/blog/71/Strings_Attached_-_A_Guide_to_Acoustic_Guitar_Strings.htm.
- [7] Newt.phys.unsw.edu.au. (2020). Guitar construction. [online] Available at: <https://newt.phys.unsw.edu.au/music/guitaracoustics/construction.html#coupling>.
- [8] Fender.com. (2020). How to Remember Guitar String Order & Names | Fender. [online] Available at: <https://www.fender.com/articles/play/never-forget-string-names-again>.
- [9] Mottola, R. (2020). Liutaio Mottola Lutherie Information Website. [online] Liutaio Mottola Lutherie Information Website. Available at: <https://www.liutaiomottola.com/formulae/fret.htm>.
- [10] Chorder.com. (2020). Guitar Fingering explained - Learn guitar chords fingering | Chorder.com. [online] Available at: <https://www.chorder.com/guitarfingering>.
- [11] ChordBank. (2020). C Chord - C Major Guitar Chord for Beginners. [online] Available at: <https://www.chordbank.com/chords/c-major/>.

[12] Academic Dictionaries and Encyclopedias. (2020). Moodswinger. [online] Available at: <https://enacademic.com/dic.nsf/enwiki/11764349>.

[13] In-text: (Audio-technica.com, 2020)

Your Bibliography: Audio-technica.com. (2020). [online] Available at: https://www.audio-technica.com/cms/resource_library/literature/49f63e6efc082082/at2020_english.pdf.

[14] Fullcompass.com. (2020). [online] Available at: <https://www.fullcompass.com/common/files/24700-BehringerUMC404HDDatasheet.pdf>.

[15] TikZ, R., Cao, S. and Cao, S. (2020). Replicate the Fourier transform time-frequency domains correspondence illustration using TikZ. [online] TeX - LaTeX Stack Exchange. Available at: <https://tex.stackexchange.com/questions/127375/replicate-the-fourier-transform-time-frequency-domains-correspondence-illustration>

[16] User.it.uu.se. (2020). [online] Available at: <http://user.it.uu.se/~ps/SAS-new.pdf>

[17] Cs.jhu.edu. (2020). [online] Available at: <https://www.cs.jhu.edu/~habib/papers/Bluetooth.pdf>

[18] B. P. Lathi, Linear systems and signals. New York: Oxford University Press, 2010.

Appendix

Python Code For Raspberry Pi

```
from bluetooth import *
import time
import threading
import numpy as np
import pyaudio
import json
import matplotlib.pyplot as plt

# change this to match the channels of the input device
CHANNELS = 1
# change this to match the number in input device id in output
DEV_IND = 2
# most devices are set to a sampling rate of 44.1 kHz
FS = 44100
# Down sample after USB read
FS_DIV = 4
# USB Buffer Size
DOWN_CHUNK = 2 ** 11
# How many large the FFT Buffer Will Be
BUF_MULTIPLE = 4
# Harmonics to test
HARMONICS = 3
# Harmonics required to pass check
HARMONICS_REQ = 3

# Standard 16bit integer from USB
FORMAT = pyaudio.paInt16

# Global variables for fft
NORMAL = 0
OPEN_AIR = 0
OPEN_AIR_SPEC = []
FFT = []

# Global variables to calculate for USB
RATE = FS//FS_DIV
DATA_CHUNK = FS_DIV * DOWN_CHUNK
BUFFER = DOWN_CHUNK * BUF_MULTIPLE
CONSTANT = RATE / BUFFER
```

Frequency of notes

E2 = 82.41
F2 = 87.31
F2S = 92.5
G2 = 98
G2S = 103.83
A2 = 110
A2S = 116.54
B2 = 123.47
C3 = 130.81
C3S = 138.59
D3 = 146.83
D3S = 155.56

E3 = 164.81
F3 = 174.61
F3S = 185.00
G3 = 196.00
G3S = 207.65
A3 = 220.00
A3S = 233.08
B3 = 246.94
C4 = 261.63
C4S = 277.18
D4 = 293.66
D4S = 311.13

E4 = 329.63
F4 = 349.23
F4S = 369.99
G4 = 392.00
G4S = 415.30
A4 = 440.00
A4S = 466.16
B4 = 493.88
C5 = 523.25
C5S = 554.37
D5 = 587.33
D5S = 622.25

E5 = 659.25
F5 = 698.46
F5S = 739.99

```

G5 = 783.99
G5S = 830.61
A5 = 880.00

# Notes on each string
String1 = [E4, F4, F4S, G4, G4S, A4, A4S, B4, C5, C5S, D5, D5S, E5, F5,
F5S, G5, G5S, A5] # High E String (4th Int)
String2 = [B3, C4, C4S, D4, D4S, E4, F4, F4S, G4, G4S, A4, A4S, B4, C5,
C5S, D5, D5S, E5] # B String (3rd Int)
String3 = [G3, G3S, A3, A3S, B3, C4, C4S, D4, D4S, E4, F4, F4S, G4, G4S,
A4, A4S, B4, C5] # G String (4th Int)
String4 = [D3, D3S, E3, F3, F3S, G3, G3S, A3, A3S, B3, C4, C4S, D4, D4S,
E4, F4, F4S, G4] # D String (4th Int)
String5 = [A2, A2S, B2, C3, C3S, D3, D3S, E3, F3, F3S, G3, G3S, A3, A3S,
B3, C4, C4S, D4] # A String (4th Int)
String6 = [E2, F2, F2S, G2, G2S, A2, A2S, B2, C3, C3S, D3, D3S, E3, F3,
F3S, G3, G3S, A3] # Low E String (Start)

# Complete chromatic scale for guitar
Strings = [E2, F2, F2S, G2, G2S, A2, A2S, B2, C3, C3S, D3, D3S,
E3, F3, F3S, G3, G3S, A3, A3S, B3, C4, C4S, D4, D4S,
E4, F4, F4S, G4, G4S, A4, A4S, B4, C5, C5S, D5, D5S,
E5, F5, F5S, G5, G5S, A5]

class SpectrumAnalyzer(threading.Thread):
    # Class Variables
    spec_x = np.fft.fftfreq(BUFFER, d=1.0 / RATE)
    data = []
    audio = []
    stream = []
    run_count = 0

    def setup(self):
        print("Spacing: ", self.spec_x[1] - self.spec_x[0])
        self.audio = pyaudio.PyAudio()
        info = self.audio.get_host_api_info_by_index(0)
        num_devices = info.get('deviceCount')
        for i in range(0, num_devices):
            if (self.audio.get_device_info_by_host_api_device_index(0,
i).get('maxInputChannels')) > 0:
                print("Input Device id ", i, " - ",
self.audio.get_device_info_by_host_api_device_index(0,
i).get('name'))

```

```

self.init_usb()

def init_usb(self):
    self.audio = pyaudio.PyAudio()
    self.stream = self.audio.open(format=FORMAT,
                                   channels=CHANNELS,
                                   rate=FS,
                                   input_device_index=DEV_IND,
                                   input=True,
                                   output=False,
                                   frames_per_buffer=DATA_CHUNK)

def kill_usb(self):
    # stop stream
    self.stream.stop_stream()
    self.stream.close()
    # close PyAudio
    self.audio.terminate()

def audio_input(self):
    global CHANNELS, DOWN_CHUNK, BUFFER
    data = np.frombuffer(self.stream.read(DATA_CHUNK, False),
dtype=np.int16) # read data
    if CHANNELS == 2: # Stereo to mono
        data = data[1::2]
    data = data[1::FS_DIV]
    data = data / 2 ** 15 # Convert To Float
    if DOWN_CHUNK != BUFFER:
        # Buffer is larger than USB read buffer
        self.data.extend(data)
        if len(self.data) > BUFFER:
            self.data = self.data[DOWN_CHUNK:]
    else:
        # Buffer is the size of USB buffer
        self.data = data

def fft(self):
    global FFT, NORMAL, OPEN_AIR, OPEN_AIR_SPEC
    # FFT = abs(np.fft.fft(self.data))
    FFT = np.abs(np.fft.fft(self.data))**2
    NORMAL = np.linalg.norm(FFT)
    # NORMAL = np.mean(FFT) * 8192.0
    FFT = FFT / NORMAL
    if self.run_count < 20:

```

```

        if self.run_count == 0:
            OPEN_AIR = NORMAL * 0.05
            OPEN_AIR_SPEC = FFT * 0.05
        else:
            OPEN_AIR = OPEN_AIR + NORMAL * 0.05
            OPEN_AIR_SPEC = OPEN_AIR_SPEC + FFT * 0.05
        self.run_count = self.run_count + 1

def graph_plot(self):
    global FFT
    plt.clf()
    plt.subplot(311)
    plt.plot(self.spec_x, FFT)
    plt.axis([0, RATE / 8, 0, 1])
    plt.xlabel("frequency [Hz]")
    plt.ylabel("amplitude spectrum")
    plt.pause(.0001)

def run(self):
    global BUFFER
    self.setup()
    try:
        while True:
            self.audio_input()
            if len(self.data) == BUFFER:
                self.fft()
                test_chord()
                # self.graph_plot()

    except KeyboardInterrupt:
        self.kill_usb()
        print("End...")

class Server:

    server_sock = []
    client_socket = []
    client_info = []
    port = []
    specThread = []
    receiveThread = []
    sendTimeThread = []
    sendContinuousThread = []

```



```

data = []
sendContinuousThreadAlive = False
def_value = True

def default(self):
    self.def_value = True

def search_note_time(self):
    notes, duration, sequence = self.check_message()
    if notes != [-2, -2, -2, -2, -2, -2]:
        self.sendTimeThread =
threading.Thread(target=self.eval_time_thread, args=(notes, duration,
sequence))
        self.sendTimeThread.start()

def search_note_continue(self):
    notes, duration, sequence = self.check_message()
    if notes != [-2, -2, -2, -2, -2, -2]:
        self.sendContinuousThreadAlive = True
        self.sendContinuousThread =
threading.Thread(target=self.eval_continue_thread, args=(notes, sequence))
        self.sendContinuousThread.start()

def stop_search_note(self):
    self.sendContinuousThreadAlive = False

switcher = {
    0: default,
    1: default,
    2: search_note_time,
    3: default,
    4: search_note_continue,
    5: stop_search_note
}

def receive_function(self, argument):
    func = self.switcher.get(argument, "nothing")
    return func(self)

def __init__(self):
    self.specThread = SpectrumAnalyzer()
    self.specThread.start()
    self.init_server()

```

```

def init_server(self):
    self.server_sock = BluetoothSocket(RFCOMM)
    self.server_sock.bind("", 5)
    self.server_sock.listen(1)
    self.port = self.server_sock.getsockname()[1]
    print("Waiting for connection on RFCOMM channel", self.port)
    self.client_socket, self.client_info = self.server_sock.accept()
    print("Accepted connection from", self.client_info)
    self.receiveThread = threading.Thread(target=self.receive_thread,
args=())
    self.receiveThread.start()

def receive_thread(self):
    print("Starting Receive")
    try:
        json_rem = ""
        while True:
            bt_text = self.client_socket.recv(1024)
            bt_text = bt_text.decode("utf8")
            print(bt_text)
            json_text = json_rem
            json_rem = ""
            is_rem = False
            for elem in bt_text:
                if elem == "*":
                    is_rem = True
                elif is_rem:
                    json_rem = json_rem + elem
                else:
                    json_text = json_text + elem
            data = json_text
            if len(data) > 1:
                self.data = json.loads(data)
                self.receive_function(self.data["type"])
    except btcommon.BluetoothError as error:
        self.client_socket.close()
        self.server_sock.close()
        print("Caught Bluetooth Error:", error)
        self.init_server()
    pass

def check_message(self):
    notes = self.data["values"]
    duration = self.data["duration"] / 1000

```

```

sequence = self.data["sequence"]
return notes, duration, sequence

def eval_time_thread(self, notes, duration, sequence):
    global FFT, NORMAL
    notes = np.array(notes)
    results = np.array([0, 0, 0, 0, 0, 0])
    normal_temp = 0
    count = 0
    t0 = time.time()
    while time.time()-t0 < duration or duration == -1:
        if NORMAL != normal_temp:
            normal_temp = NORMAL
            count += 1
            results += check_three_sets(notes)

    for i in range(6):
        if results[i] > count * 0.5:
            results[i] = 1
        else:
            results[i] = 0

    results = results.tolist()
    self.send_note(sequence, results)

def eval_continue_thread(self, notes, sequence):
    global FFT, NORMAL
    notes = np.array(notes)
    normal_temp = 0
    while self.sendContinuousThreadAlive:
        if NORMAL != normal_temp:
            normal_temp = NORMAL
            results = check_three_sets(notes)
            self.send_note(sequence, results)

def send_note(self, sequence, results):
    my_json = {
        "type": 3,
        "sequence": sequence,
        "timeStamp": 0,
        "values": json.dumps(results),
        "duration": -1
    }
    json_message = json.dumps(my_json) + "*"

```

```

        print("Sent:      " + json_message)
        json_message = bytes(json_message, "utf8")
        self.client_socket.send(json_message)
        return json_message

def init_matrix(string):
    # Setup harmonics on each string
    string_fft = [[[0, i, j] for i in range(HARMONICS)] for j in
range(len(string))]
    for x in range(len(string)):
        for y in range(HARMONICS):
            string_fft[x][y][0] = round((y + 1) * (string[x] / CONSTANT)) #
Base
            base = string_fft[x][y][0]
            if x == 0:
                span = (y + 1) * np.floor((round(string[1] / CONSTANT) -
string_fft[0][0][0])) / 4)
            else:
                span = (y + 1) * np.floor((string_fft[x][0][0] -
string_fft[x - 1][0][0])) / 4)
            string_fft[x][y][1] = int(base - span) # Left
            string_fft[x][y][2] = int(base + span) # Right
        return string_fft

def check_power():
    return NORMAL > OPEN_AIR * 200.0

def check_seg_noise(thr, sum_air):
    return thr > sum_air * 200.0

def check_seg_power(sum_fft_seg, sum_fft):
    return (sum_fft_seg / sum_fft) > 0.001

def check_point(thr, j):
    return (FFT[j] * NORMAL > thr) or (FFT[j - 1] * NORMAL > thr) or (FFT[j
+ 1] * NORMAL > thr)

def test_chord():

```

```

test_notes = np.array([0, 0, 2, 2, 2, 0])
print(check_three_sets(test_notes))

def check_three_sets(notes):
    ev = eval_notes(notes)
    low = eval_notes(notes - 1)
    high = eval_notes(notes + 1)
    for i in range(6):
        if (low[i] == 1) or (high[i] == 1):
            ev[i] = 0
    return ev

def eval_notes(note_array):
    sum_fft = sum(FFT)
    do_string = [True, True, True, True, True, True]
    for i in range(6):
        if note_array[i] < 0:
            note_array[i] = 0
            do_string[i] = False
        else:
            do_string[i] = True

    notes = [StringFFTIn[note_array[0]], StringFFTIn[note_array[1] + 5],
             StringFFTIn[note_array[2] + 10], StringFFTIn[note_array[3] +
15],
             StringFFTIn[note_array[4] + 19], StringFFTIn[note_array[5] +
24]]

    results = [0, 0, 0, 0, 0, 0]
    for i in range(6):
        bb = 0
        if do_string[i]:
            for x in range(HARMONICS):
                note = notes[i][x]
                sum_fft_seg = sum(FFT[note[1]:note[2]])
                sum_air = sum(OPEN_AIR_SPEC[note[1]:note[2]])
                thr = sum_fft_seg * NORMAL
                if check_power() and check_seg_noise(thr, sum_air) and
check_seg_power(sum_fft_seg, sum_fft):
                    bb += 1
            if bb >= HARMONICS_REQ:
                results[i] += 1

```

```
return results
```

```
StringFFTIn = init_matrix(Strings)  
# server = Server()  
specThread = SpectrumAnalyzer()  
specThread.run()
```

Android Github Link

<https://github.com/marcpaulroy/Group17Android>