Signal Harmonic Re-tuning Electronic Device (S.H.R.E.D.) Guitar Autonomous Tuning System: Final Report

Team GA-5

Ryan Barker

Duke Durot

Jules Ebaa

Shane Ma

Michael Norcia

Problem statement

The purpose of this project was to design an autonomous tuner system that could easily be mounted to any electric guitar and would allow the user to tune the instrument to 3 different tunings. There was no restriction on how much hardware to incorporate into the system; however, there was some limitations on how much physical change could be made to the guitar. Any aspect of the hardware is completely hidden from the user. Instead, team was required to develop a user friendly wireless application that would walk to user through the steps to tune his instrument correctly. Despite the complexity of our proposed solution, we did not want to take away the ability to operate the instrument.

Specifications

The main goal was to substantially improve the user's tuning experience. We chose to build a detachable system that would contain all our circuity (motors, processors) and that could be mounted on most if not all guitars supporting 3+3 tuning peg configuration. The initial idea was to implement a polyphonic tuning, but we realized after studying the project extensively that it will require considerate physical changes on our electric guitar; changes that we wanted to minimize. The team opted for a monophonic tuning using the pickup integrated in the instrument. The detachable characteristic of the tuner was so it could be mounted on the guitar's headstock, without interfering with the user ability to play in the stand-up position. The user interfaces with our system through an Android Application that will be available in the Google Play Store. The application uses Bluetooth as communication protocol and allows three different tunings: Standard, E flat and Open G. Our design incorporates software modules and hardware modules. Those modules are made up of independent components that were thoroughly tested before they were integrated in the overall system. The hardware modules feed the correct signal to our software module that performs the correct computations and send back some commands. We tried as much as we could not to reinvent the wheel and relied on reputable vendors to provide us with high quality parts that we needed. We adopted the strategical approach of implementing a fully functional system on a single tuning peg before expanding to the other 5.

Final Design

The overall final design is broken into five separate subsystems, as shown in Table B.1 in Appendix B. The audio input subsystem obtains signal from the guitar and amplifies the signal level above the noise level on the input cable. The audio sampling subsystem uses a USB audio interface in combination with the C ALSA library to read data from subsystem one at a sufficient sampling rate and passes that data to subsystem three. The frequency analysis subsystem converts the audio signal passed into the Fourier domain, analyzes the data to determine the first harmonic of the strummed string, and send commands to subsystem four to tune the guitar. The motor control subsystem connects to six motors directly connected to the guitar's tuning pegs and interprets commands from subsystem three to tune the guitar. The user interface subsystem prints data from subsystems three and four to the Pi's HDMI screen to inform the user about the tuning process and prompt them to strum each string when appropriate. Due to design issues that will be discussed later in the report, the proposed Bluetooth control application was not able to be integrated into the design. Figure B.1 in Appendix B shows the subsystem view of the final design graphically. Additionally, the main software logic for the overall design is represented as an ASM chart in Figure B.2.

For the audio input system, the guitar's native pickup, the internal electronics of the guitar, and an Austin Super 25 guitar amplifier were used. When a string on the guitar is strummed, the pickup outputs its data into the amplifier input, which then lifts the string data above the noise data, and sends the guitar signal data out of the amplifier's headphone output to be read. Note that this subsystem is different than the proposed subsystem one. The primary advantage of using the guitar's native pickup and a regular guitar amplifier over the proposed hexaphonic pickup and filtered operational amplification circuit is that our final design is able to be centered on a "plug and play" concept. During the design process, we realized that all guitarists use a variety of gear, and that our design would better accommodate end users if we could integrate our tuner with their specific amplifier. This way, all a user needs to do is plug their setup into the input of our tuner, tune, and disconnect from our autotuner rather than disconnecting their guitar from their amplifier, connecting it to the autotuner, tuning, and reconnecting back to their gear. Removing the hexaphonic pickup from subsystem one also sacrificed polyphonic tuning, but drastically simplified the designs for subsystems one through three, so we felt it was the correct design decision.

The audio sampling and frequency analysis subsystems work in tandem to read in and analyze the data from the audio input subsystem all inside of a Raspberry Pi Model B Version 2 microcontroller. This is different than the more complex proposed design of using a Tiva to sample data and a Raspberry Pi to manipulate it. It was accomplished by using a Sabrent audio interface to read audio data directly into one

of the Pi's USB ports, and made the proposed six channel analog to digital converter unnecessary. In fact, using the Sabrent audio interface allowed us to eliminate the Tiva from the design in general, again simplifying our design complexity. Because sampling was happening directly on the Pi and samples did not have to be transferred into the system, using the Sabrent interface also allows the design to sample at a much higher 4096K samples per second rather than the proposed 1024K, making the overall tuner significantly more accurate.

The audio sampling subsystem essentially acts as gateway into subsystem three. It reads raw AC voltages into the Sabrent audio interface, automatically transforms them into digital signaling inside of the interface's internal analog to digital converter, and transfers the digital output data into subsystem three. The software design of this subsystem was conveniently controlled via the C ALSA library, which contains standard Linux functions to manipulate USB audio data. ALSA was used to sample 4096 thousand samples over one second and copy the returned data into a buffer array that is later transformed into the Fourier domain. Using the library is as simple as setting several input parameters, initializing the USB interface for reads, and querying the interface for data all via repeated library function calls. Among the input parameters, the library allows the user to specify the data size to store audio samples and the desired sample rate. The key word here is desired: The library does its best to sample at the passed in sample rate, but actually samples as close to the desired sample rate as possible and returns the actual sample rate to the user. Also, when data is being read, the library returns it to the program in windows rather than all at once to allow for a smaller input buffer. For our purposes, we read audio data in as 16 bit integers and set the sampling window to 32 integers. The sampling rate is set at a supported 4096 thousand samples per second. Since we desired 4096 audio samples, each time the system reads data, it queries the audio interface for 128 windows of data. Each time a window is returned, the audio sampling software copies it into the appropriate position of a larger 4096 integer array, which is what is returned to subsystem three. A set of sampled data is shown on the left side of Figure B.3.

The frequency analysis subsystem is the computing power for processing each guitar string signal and determining how close or far each string is from in tune. It utilizes a C Fast Fourier Transform library called GPU FFT to transform each set of 4096 audio samples into frequency magnitude plots shown on the right side of Figure B.3. GPU FFT runs exclusively on the Pi's GPU, which optimizes both frequency transforms and motor control by allowing them to run on separate processors. After GPU FFT is called, the software analyzes the plots it returns to find the first harmonic of the audio sample. This is done by computing the maximum magnitude in the plot and finding an initial frequency as the first frequency with nondecreasing magnitude above a noise threshold of 45 percent of the maximum magnitude. This initial frequency is compared to a noise frequency threshold of 65 hertz in the main autotuner logic to decide if

it is noise or a harmonic of string data. If the frequency is determined to be string data, it is divided by two until it is within 50 hertz of the target string tuning frequency. This process essentially transforms the second or third harmonics of each string when they are read back down to the first harmonic of the string. Once the first harmonic of the string has been accurately obtained, it is send to the motor control subsystem to generate a motor control command.

The output motor control subsystem uses the C wiringPi software API to send motor control commands to a digital circuit of six multiplexed stepper motors and stepper motor drivers. The software takes the first harmonic of each read string sample and compares it to its corresponding tuning frequency for the tuning selected. Tuning frequencies for each string were computed using Equation B.3, which relates musical half steps and a reference tuning frequency back to tuning frequencies. The software uses the prior discussed comparison to determine the correct direction to turn the string and whether to fine or coarse tune the string. The decided motion is packed into a tuning command, which is sent to library function calls for wiringPi to step the stepper motors as decided. Fine tuning a string moves its tuning peg 32 degrees, and coarse tuning a string moves its tuning peg 16 degrees. The primary reason for using fixed fine and coarse step sizes rather than calculating the exact amount to move the tuning peg to tune the string is that it protects against bad frequency reads causing a guitar peg to spiral out of control and break a guitar string.

The hardware design of the motor control subsystem contains three digital three bit multiplexers, six digital stepper motor drivers, six 100 μF capacitors, and six stepper motors. One capacitor is coupled to the power supply pins of each stepper motor driver to protect the driver from voltage and current oscillations. The multiplexers are used to multiplex step, direction, and enable input signals via three control bits to the corresponding stepper motor driver when the software decides to turn a motor. Multiplexing the inputs allows the design to use the same step, direction, and enable GPIO pins on the Pi for each stepper motor driver. Incorporating the low-active enable line into the design allows the design to disable motors while they are not turning, which reduces the current and power consumption of the motor control circuit. Figure B.4 shows a photo of the final motor control circuit.

The user interface subsystem consists of simple C printf statements that run along the main script and prompt the user accordingly as the overall system runs. When the overall system starts up, the user interface asks the user for a tuning to tune the guitar to and error checks the specified value. When a valid tuning number is entered, the user interface informs the user that the autotuner is starting and to strum each string individually, starting with the thickest string. From here, audio is captured by the main script, transformed into a frequency, and ignored until the user strums a string. Once this occurs, the user

interface informs the user of the tuning command that was generated and continually repeats this until the string being analyzed is in tune. Then it tells the user to strum the next string, at which point the whole process repeats until the user interface informs the user that the entire guitar is in tune and asks for the next tuning command. Figure B.5 shows the final user interface for the design.

Note that the implemented user interface is drastically simpler than the proposed Android Application user interface. The proposed Bluetooth and Android application subsystems were not integrated into the final design, but this does not mean these subsystems were ignored. Both of them were fully developed, debugged, and tested between the application and a laptop running Ubuntu 14.04 LTS. Furthermore, code was developed and tested to allow either the control application or design firmware to act as either the Bluetooth client or server and still allow communication (See Bluetooth.c and the application code in the autotuner software). Figure B.6 shows screenshots of the Android application from testing Bluetooth.

There was a problem discovered late in the design process surrounding the operating system the design uses for the Raspberry Pi, which is the reason the Bluetooth subsystems were not integrated into the final design. As it so unfortunately happens, Raspbian version 3.18 has several low level Bluetooth bugs in the system configuration files which prevent it from correctly establishing a Bluetooth connection. The bugs cause the Pi to become unable to correctly follow through a Bluetooth connection to another device, causing it to deny all incoming connections from remote devices and be unable to finish outgoing connections to remote devices. A computer engineer in the design group spent a significant amount of time editing the configuration files to attempt to fix the bugs, but the code eventually became too low level for him to be able to edit. This bottlenecked the design into the command line interface, but to compensate, an automated user interface demonstration script was written into the control application to show how it would have worked, which was show with the system at all project demonstrations.

Cost Accounting

A budget spreadsheet was kept through the semester and used to generate the cost accounting table seen in appendix A. The table in appendix A reports final development cost, out-of-pocket development cost, final artifact cost, and final out-of-pocket artifact cost. Final development cost is the total cost of all items obtained for the design (though not necessarily used), final out-of-pocket development cost is final development cost minus items not purchased, final artifact cost is the cost of all items used in the final design, and final out-of-pocket artifact cost is final artifact cost minus items not purchased.

The cost accounting in appendix A heavily reflects the decision to change our group's final design, as our total development costs were at an expensive $687.54, but our final out-of-pocket artifact costs were at a much more reasonable $246.24. $254.99 of our design came from previously acquired parts, namely, the guitar and amplifier used. $169.49 was spent on superfluous parts not used in the design. Of these parts, the most expensive component was the $83.98 spent on two motor control PCBs we ordered but never integrated. This was due to an electrical mistake in the schematic used to generate the trace diagrams for the PCB, which used pinouts for analog multiplexers when the motor control circuit needed digital multiplexers to function correctly. Apart from this $60.00 was also spent on the game containing the hexaphonic pickup we wanted to, but as previously discussed, were unable to use. This semester, our group has learned that careful planning and testing are the most important parts of the design process, as executing these tasks correctly significantly saves on the design budget.

Performance Characterization and Discussion

To begin evaluating the final design, we ran a series of experiments to determine the tuning accuracy of the overall system. Tests began with the guitar either close to standard tuning or in another tuning. We ensured to use a uniform distribution of the two beginning states across our testing. After the beginning state was set, the autotuner was started, tuned the guitar was tuned to standard tuning, and the Android Application DaTuner Lite was used to measure and record the final frequency of each string after the entire guitar was in tune. After all trials were completed, the average frequency value of each string and accuracy, defined as target value minus average value, were both computed. Figure 1 shows the results of this tuning accuracy testing tabulated and graphically.

Tuning Accuracy Test Data							
		String Frequencies (Hz)					
		6	5	4	3	2	1
Target Values		82.41	110.00	146.80	196.00	246.94	329.63
Trial	Distance to Standard Tuning						
1	Close	83.21	110.50	147.00	196.12	246.35	330.00
2	Far	83.50	111.00	146.30	196.50	246.80	329.85
3	Far	82.12	109.00	146.75	197.00	246.94	328.75
4	Close	81.90	110.50	146.90	195.00	245.99	330.30
5	Far	83.10	110.12	146.80	196.50	247.10	329.63
6	Close	83.30	110.23	146.90	196.14	247.50	330.15
7	Close	82.00	110.34	147.10	196.77	246.51	329.63
8	Far	82.34	109.91	147.20	196.30	246.80	328.90
9	Close	81.89	109.24	146.80	195.85	246.50	329.80
10	Far	83.10	110.54	146.45	195.30	247.00	330.00
Average Value		82.65	110.14	146.82	196.15	246.75	329.70
Accuracy		0.24	0.14	0.02	0.15	-0.19	0.07

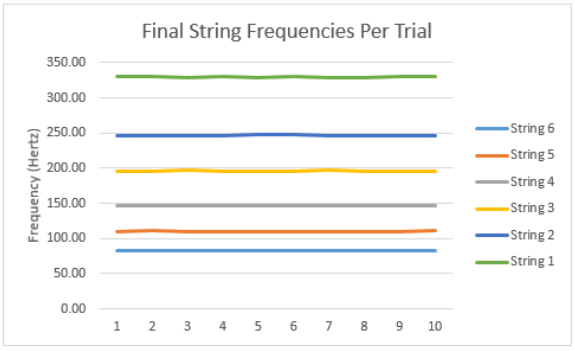

Figure 1. Tuning Accuracy Testing.

After the tuning accuracy of the system was determined, the next step was determining the average number of total strums and time required to tune the guitar. During this testing, we realized that due to the motors moving a fixed step sizes, the performance of the system varied greatly depending the

two methods of tuning the guitar. Method one, called fine tuning, is when the user starts with the guitar close to one of the three tunings, starts the autotuner, and tunes to that tuning. This is analogous to the user playing an in tune guitar on stage for three to four songs duration and then tuning the guitar. Method two, called swap tuning, is when the user starts with the guitar in one tuning and moves to a different tuning. In the tuning time experiment, we ran an equal number of fine tunings and swap tunings, recording the number of strums across all six strings and total time tuning the guitar. Figure 2 show the results of tuning time testing tabulated and graphically.

Tuning Time Test Data				
Trial	Tuning	Tuning Type	Number of Strums	Tuning Time (s)
1	Standard	Fine	21	45
2	Drop D	Fine	18	40
3	Open G	Fine	16	38
4	Standard	Swap	34	116
5	Drop D	Swap	37	120
6	Open G	Swap	32	110
7	Standard	Fine	23	47
8	Drop D	Fine	16	38
9	Open G	Fine	17	39
10	Standard	Swap	38	122
11	Drop D	Swap	34	117
12	Open G	Swap	30	107

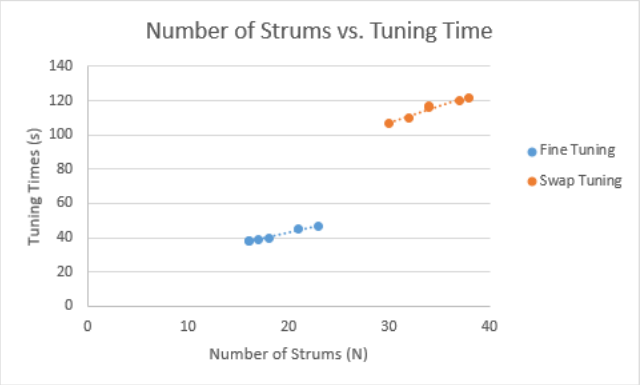

Figure 2. Tuning Time Testing.

The primary strength of our design is that the final frequencies of each string are accurate to within less than one hertz of the target tuning value, as shown in Figure 1. This reliability is due to checks in the software that do not allow a string to specified as in tune until the frequency of the string is within one hertz of the value. For this reason, the final frequencies of the string have very small variation, and the graph of final string frequency vs. trial number is flat and linear. The fact that our system is so reliable is very important to the end user, since too much variance in string frequency on a guitar autotuning system would cause dissonance while the guitar was being played, which is very undesirable.

From Figure 2, it can be found that it takes an average of 18.5 strums and 41.16 seconds to fine tune a guitar with our system and 34.17 strums and 115.33 seconds to swap tune a guitar. This means that our system performs very well when the user sticks to the same tuning, and takes longer when the user decides to switch tunings. Note a maximum variation of eight strums between swap tuning trials. This is because depending on what tunings you move between, different strings must take different numbers of half steps up or down. The worst case performance is switching between standard tuning and open G tuning, where three strings must be raised or lowered by a half step each depending on which tuning you are moving to and from. All of these variations in performance are because the motors are taking fixed step size motions rather than calculating exactly how far to move to get in tune for each tuning phase. Additionally, the number of strums per tuning versus the time per tuning exhibit close linear relationships,

as shown in the trend lines of Figure 2. These relationships again support the reliability of our system's overall performance.

The primary weakness of our design is not represented with data, but rather our command line user interface itself. Though it is clean for a command line interface, this is not the ideal way for the user to communicate with the system easily. The Bluetooth application would have been a far more intuitive way for users to interact with the system, but as already discussed, implementing Bluetooth communication was not feasible on the Raspberry Pi. This fact is what led to the development of the Bluetooth tests and demonstrations shown with the application. The command line interface is appropriate as a first step user interface for a prototype, but if the team was given more time to work on the project, the first thing we would do is integrate a new communication method with the Android application and use the interface it provides instead.

Project Postmortem

Our design had several strengths. Its primary strength was the "plug and play" idea it was centered around. The design was very compact, and attaching it to the guitar was a simple as using an Allen wrench to tighten the stepper motors onto the guitar's tuning pegs. The overall system was very accurate to a close plus or minus one hertz of precision for all tunings, and this parameter, along with other tuning constants, was adjustable via an easily accessible ANSI C header file. The software itself was written entirely in ANSI C, making adjusting the package as simple as possible, and allowing the software package to easy flip into a debug mode that used gnuplot to print out graphs of Fourier transformed data. The entire project fit very neatly into a small black box, stepper motors included. The design was easily portable as long as the destination had two five volt, two amp power supplies to power the motor control circuit and Raspberry Pi, respectively. The motor control circuitry and software only activated the motors when they were in motion, which saved power and reduced the risk of electrical components overheating.

The primary weakness in the design was the user interface. As documented in the preliminary report, this was initially meant to be set up in a Bluetooth Android application, but we unfortunately found out low level operating system bugs kept Bluetooth from working with version 4.18 of Raspbian (the version we had installed on our Pi) too close to the deadline. This issue was verified by observing our code work with a dual booted Linux box, but not the Pi. We ended up using the Pi's command line for the interface. The time crunch on the issue was mainly due to the team's poor anticipation of how challenging the motor control circuit would be. The engineer in charge of motor control ended up needing assistance from the engineer working on Bluetooth, which bottlenecked the time the Bluetooth engineer could work

on Bluetooth. If we were to do this again, we would have two electrical engineers work on motor control, so the Bluetooth computer engineer could have found a suitable alternative user interface.

Another minor weakness in our design, is though it was portable and easy to set up, the motors we used were large and heavy. We tested several smaller motors to use as alternatives throughout the semester, but found that the NEMA 17s (part number in appendix A) were the only motors with enough torque to turn each string at tension. However, the NEMA 17's were still suitable for the prototyping phase our design was in. To facilitate their mass, we attached the motors to each other in pairs of two with pivot points, but there is still a wide variety of guitars our prototype cannot fit because of the motor's sheer size. If we ever to mass-produce the design, we would custom-order more expensive, smaller motors for the tuning pegs with the same torque, voltage, current, and step size ratings.

The last weakness in our design was the amount we had to deviate from the original design specification itself. The original specification used a hexaphonic pickup and the Tiva to sample and tune all six strings at once. It also included the working Bluetooth user interface. Though the entire team wanted to meet these goals, we quickly learned that the hexaphonic pickup and Tiva added complexity to both the hardware and software design that was not practical to carry out in one semester. In addition, we learned to use the pickup, we would need to write a custom Simulink block, which one computer engineer did his best to learn in a period of two weeks, but could not find references for what he was trying to accomplish. When we decided to change design concepts, we delayed our timeline significantly, which lead to the issues discussed prior in this section. Overall, though we are disappointed we could not meet the original design specification, we feel it over-promised too much design complexity than is practical for one semester. We are also quite happy with the way our final design polished itself out to be.

A project is a success when all the technical requirements are met by the deadlines and all the functionalities are shown on presentation day. However, this success does not undermine the importance of nontechnical aspects such as project management, time management, team coordination and communication issues that facilitate the completion of the project. The management of nontechnical factors started with establishing a mode of communication: The team members exchanged phone numbers, set up a GroupMe chat, and created shared folders on Dropbox and Google Docs. The team generated a Gantt chart, which laid out the tasks to be completed and their deadlines. The software related tasks were under computer engineers' supervision Ryan Barker and Jules Ebaa and the hardware/electrical aspects under the electrical engineers Duke Durot, Michael Norcia and Shane Ma. Although the tasks were assigned with respect to the milestone requirements, team members could work on future requirements provided the current requirements are met. Also, each teams frequently cross-

collaborated on tasks. For example, Ryan and Duke spent a significant amount of time working together on the electrical work for the motor control circuit.

The group met at least twice a week in the laboratory: The first meeting at the beginning of the week (Monday) was for the members to get up-to-date and set weekly goals. The last meeting was held at the end of the week (Sunday) to go over the status meeting. Any other meeting (solicited members, day, time and location) was called based on what needed to be accomplished.

The team was well-coordinated. Members made themselves available to help one another anytime their schedules allowed. Members made sure they informed the rest of the group of any change in the design (whether positive or negative) through status update on GroupMe. Any software change was accompanied by new file uploads into our Dropbox, and previous versions of software design were kept in separate folders for backups. On the hardware-level, the team kept schematics of all circuits built. Ryan Barker, Michael Norcia and Duke Durot handled administrative tasks, kept up with the deadlines, and ensured the design met the technical requirements as best as possible. They also kept a very open communication channel through the group for the entire semester.

There was no conflict throughout our project design. The team members were very respectful of one another. Everybody's opinion was taking into consideration and each component of the final design was based off of majority decision. As per normal team situations, we had a few time conflicts, but we always figured out ways to work them out. Members had the option to work remotely if that could help their productivity.

The team tried to have each member working on something different at all times to distribute tasks as evenly as possible. Sometimes, this meant working inside of someone else's task. We believed by working in parallel, we would make optimal progress and each of us would become adept at something specific. The fast learners inside of the group aided this process by volunteering their time to help others grasp more challenging concepts.

Highlights

Our team's primary design goal was to make our product as "plug and play" as possible. This was achieved by making all of the equipment functional with standard electric guitar gear, such as ¼" audio cables, solid state or tube amplifiers, and standard output electric guitars. After making minor modifications to the headstock of the guitar, our device truly is plug and play, requiring only that you put the motors on the headstock and plug everything in.

After achieving such capabilities, the next focal point of the design process was to make the device portable. This was achieved by placing all components into a black box. The box contains the motor driver circuits, the Raspberry Pi, and storage space for the stepper motors, which are extracted for use. The only way that the user interfaces with the box is plugging in the output from the amplifier and attaching the motors onto the guitar.

Another aspect of our team's design that is unique is the arrangement of the motors. Each motor is held in place on the guitar only by another motor. This limits the hardware and frame required to use the motors while also increasing portability. In final production design, it would also reduce the total amount of time needed to spend on manufacturing.

Appendix A – Cost Accounting

Item Description	Part Number	Vendor	Unit Price	Quantity	Total Cost	Payment Type	Used in Final Design?
Epiphone Les Paul Express Electric Guitar	N/A	Sweetwater.com	$119.99	1	$119.99	Previous	Yes
Raspberry Pi	Version 2 Model B	Adafruit.com	$41.99	1	$41.99	Purchased	Yes
PowerGig: Rise of the Six String (For hexaphonic pickup)	N/A	Amazon.com	$60.00	1	$60.00	Purchased	No
Micron Single Rail Op Amp	MCP602-E/P	Microchip.com	$1.26	2	$2.52	Purchased	No
Sabrent USB Audio Interface	N/A	Amazon.com	$5.99	1	$5.99	Purchased	Yes
TH41 Profession Audio Converter Jack Pack	N/A	Amazon.com	$6.90	1	$6.90	Purchased	Yes
Guitar Cabling	Pig Hog Gold Cable	Guitar Center	$15.06	1	$15.06	Donated	Yes
Austin Guitar Amplifier	Super 25	Guitar Center	$125.00	1	$125.00	Previous	Yes
Stepper Motor	NEMA17400Z	Amazon.com	$18.00	6	$108.00	Purchased	Yes
Motor Control PCB	N/A	ExpressPCB.com	$41.99	2	$83.98	Purchased	No
6"x8" Copper Strip Board	N/A	Amazon.com	$10.00	1	$10.00	Purchased	No
Texas Instruments Tiva Launchpad	N/A	ECE Department	$12.99	1	$12.99	Donated	No
Texas Instruments Decoder Multiplexer Chip	SN74155	ECE Department Lab Kits	$1.76	3	$5.28	Donated	Yes
Pololu Low Voltage Stepper Driver Chip	DRV8834	Pololu.com	$4.95	6	$29.70	Purchased	Yes
Shaft Coupler	N/A	Amazon.com	$6.69	6	$40.14	Purchased	Yes
Circuit Wiring and Other Electrical Components	N/A	Adafruit.com	$10.00	1	$10.00	Previous	Yes
Wood frame and set screws	N/A	Lowe's	$10.00	1	$10.00	Purchased	Yes

Final Development Cost – Total cost of all Items	$687.54
Out-of-Pocket Development Cost – Total cost of NOT donated items	$393.22
Final Artifact Cost – Total Costs of items from final design	$518.05
Out-of-Pocket Final Artifact Cost – Final Artifact Cost minus previously purchased/donated items	$246.24

Appendix B – Figures and Tables

Subsystem Number	Purpose
1	Audio Input Subsystem
2	Audio Sampling Subsystem
3	Frequency Analysis Subsystem
4	Motor Control Subsystem
5	User Interface Subsystem

Table B.1. Final Design Subsystems.

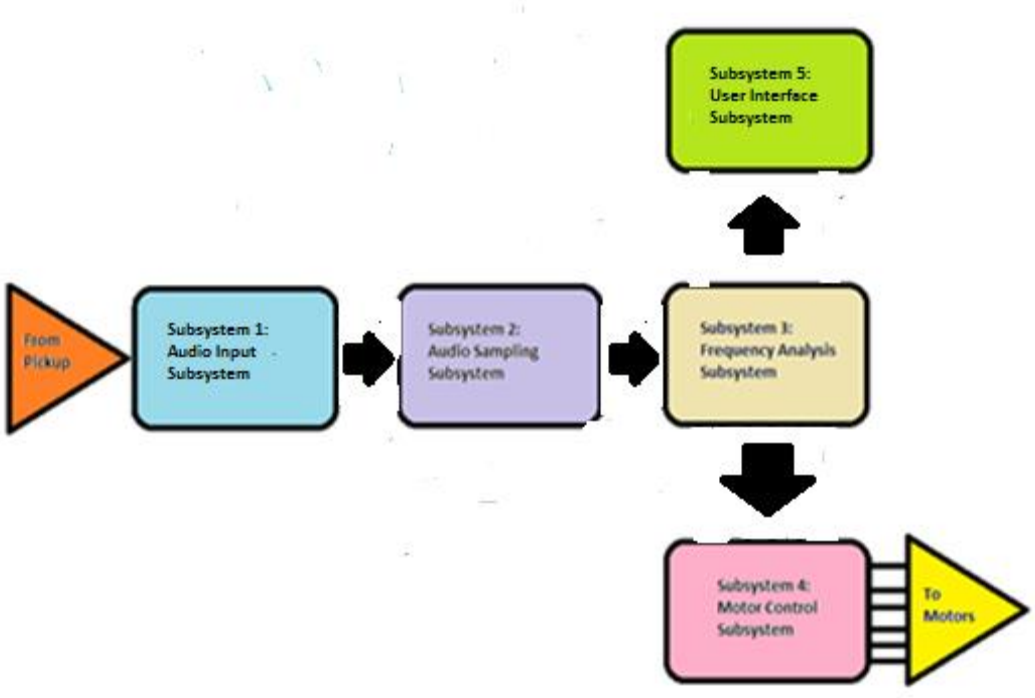

Figure B.1. Final Design Subsystems.

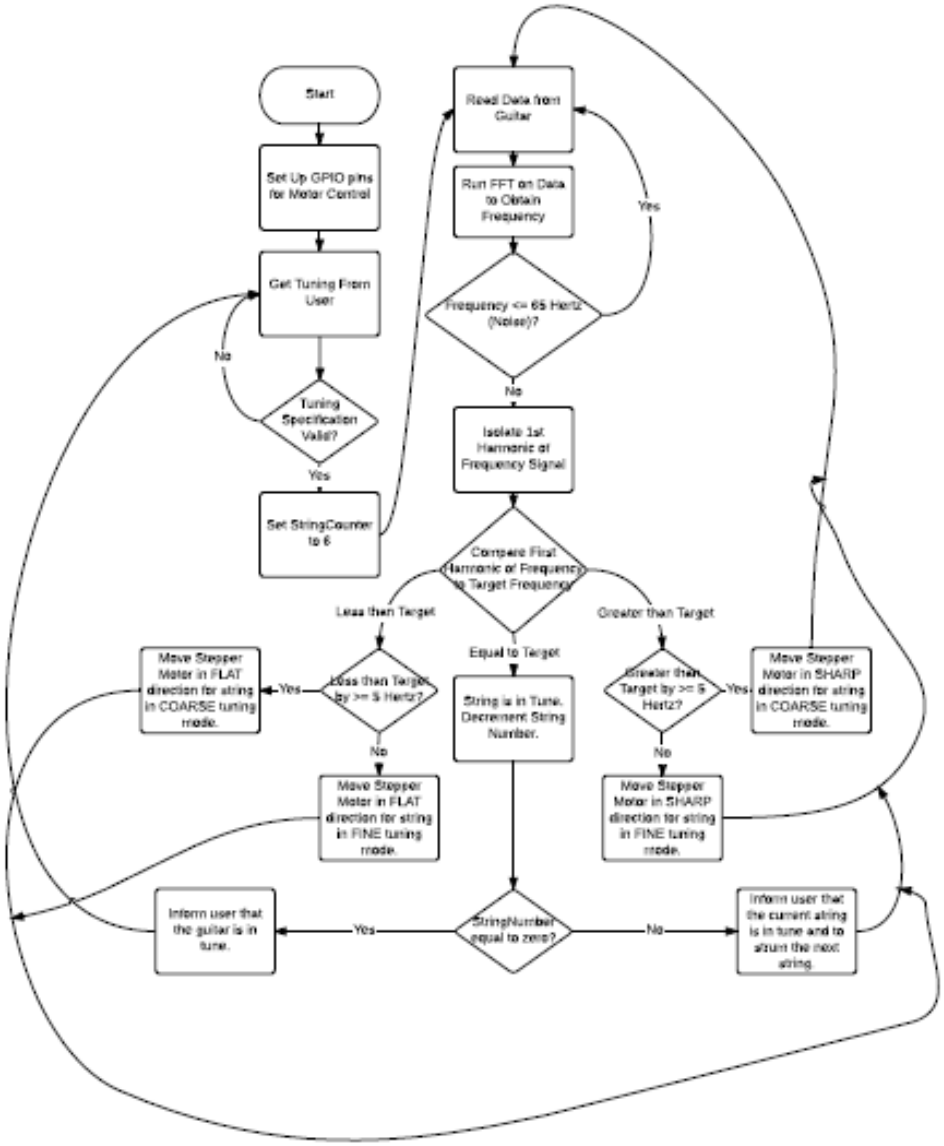

Figure B.2. Overall Software Logic.

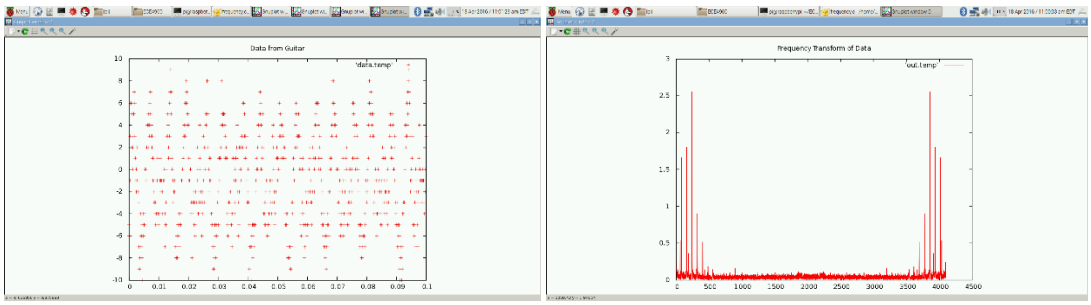

Figure B.3. Sampled Audio Data (Left), Transformed Frequency Magnitude Plot (Right).

$$f_n = f_0 \cdot a^n$$

Where:

$$f_0 = 440\ Hz\ (Reference\ Pitch)$$
$$a_n = \sqrt[12]{2}\ (Constant)$$
$$n = The\ number\ of\ half\ steps\ away\ from\ f_0.$$

Equation B.3. Tuning Frequency Calculation Formula.

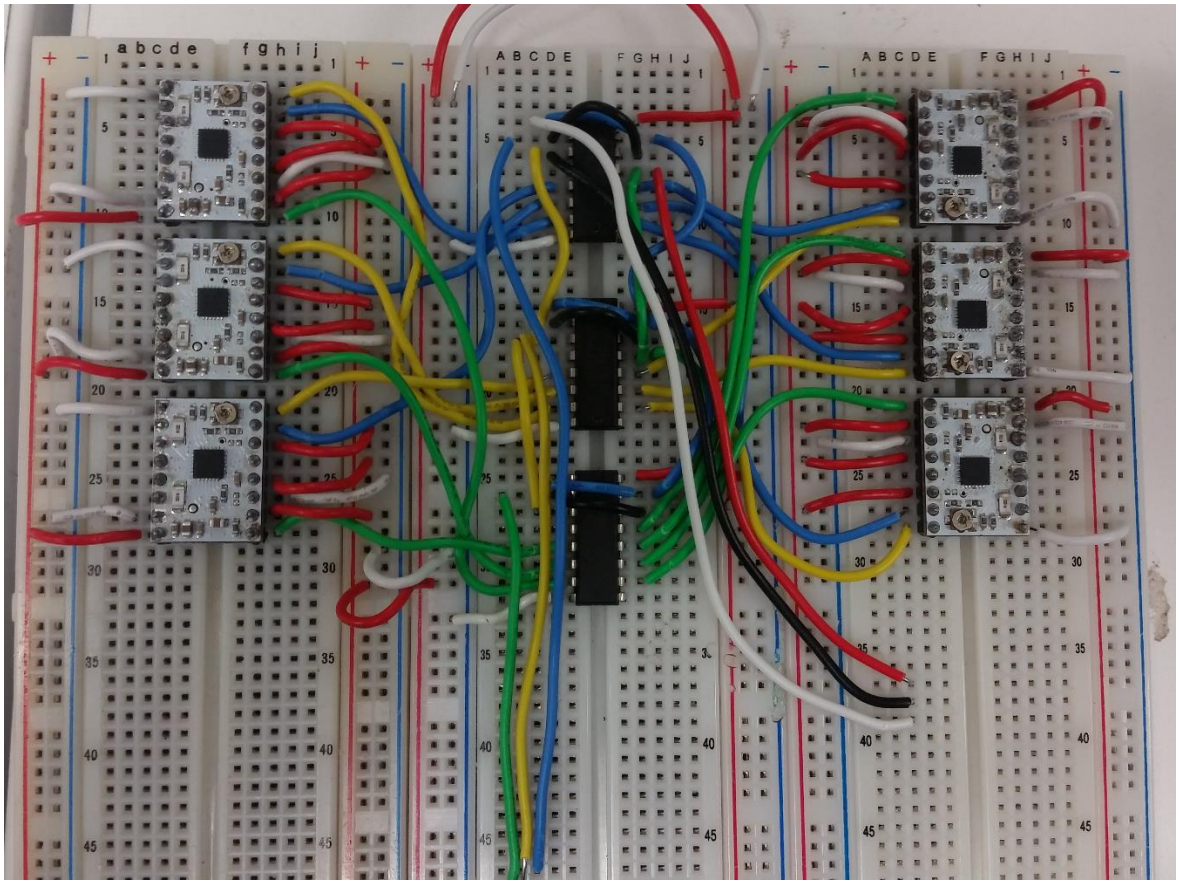

Figure B.4. Final Motor Control Circuit.

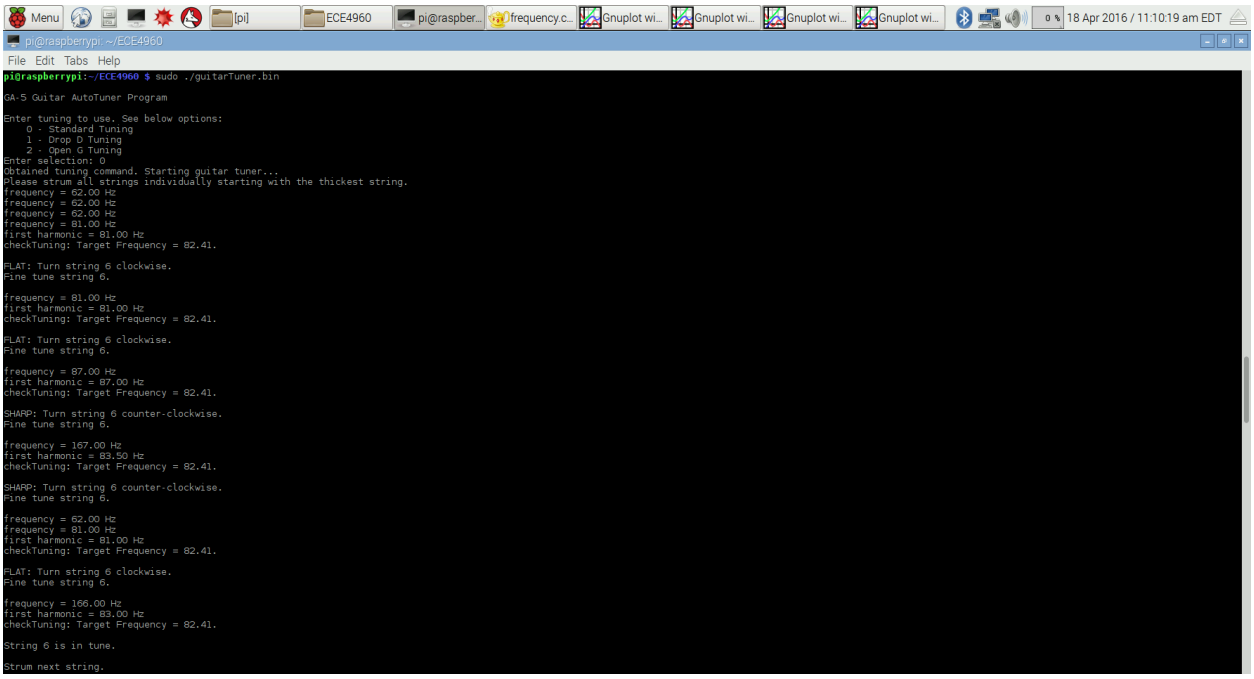

Figure B.5. Final User Interface.

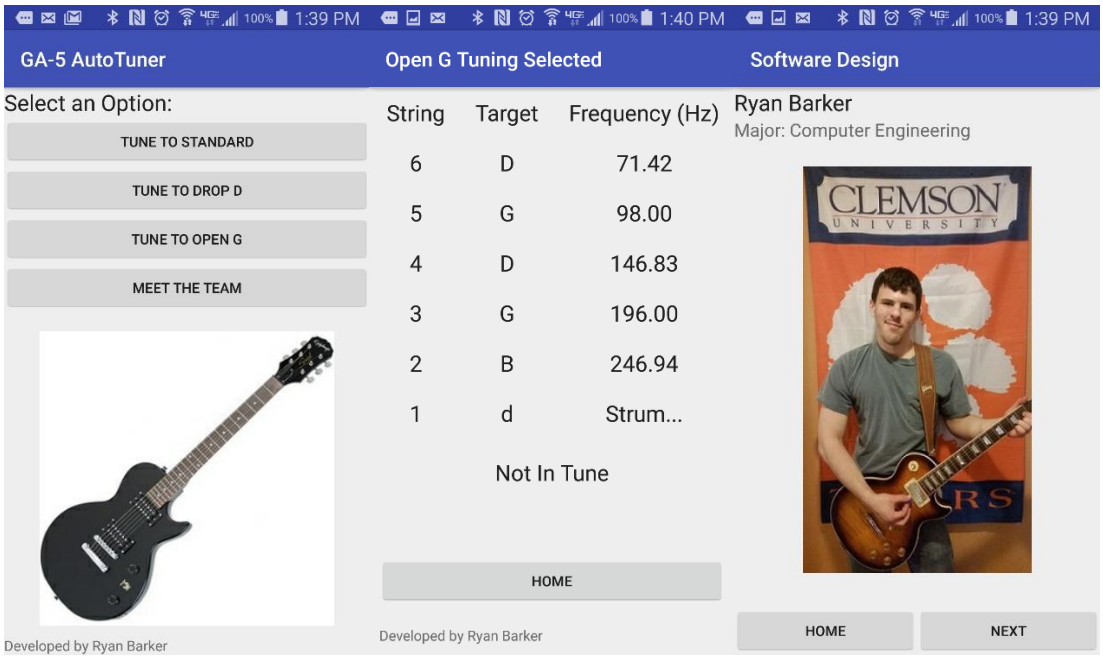

Figure B.6. Proposed Bluetooth User Interface.