DREXEL UNIVERSITY
Senior Design
Electrical and Computer Engineering

PROGRESS REPORT
Spring 2018-2019

Team Number ECE-29

Automatic Violin Tuner

Team Members

Name	Department	Email
Yoshin Govender	ECE	yg353@drexel.edu
Mike Barnes	ECE	mrb372@drexel.edu
Steven Ngan	ECE	sjn36@drexel.edu
Anthony Santoro	ECE	acs385@drexel.edu

Team Advisor(s)

Name	Department/Company	Email
Dr. Kevin Scoles	ECE	scoleskjdrexel.edu

Group Leader's Signature: Yoshin Govender

Advisor Signature (or attach e-approval): Kevin Scoles

Abstract

Violin tuning is one of the foundations for playing with consistent and accurate intonation. Tuning, whether it is done through the violin pegs or fine tuners, changes the frequency and pitch of the notes. From a young age, violinists are taught this skill and subsequently require years of constant exposure in order to develop and maintain the skill set to tune independently. This device, an automated tuner for the violin fine tuners, is designed to help younger musicians expedite this skill to develop the capabilities for recognizing the correct pitch intuitively. By comparing the out of tune pitch to the pitch corrected by the device, students can compare what strings are supposed to sound like with proper intonation.

As the target audience is mainly for children, an interface which is easy to use and self-explanatory should be ideal for the design. As a result, a menu display allows for easy navigation among various stringed instruments as well as their string set.

The project required months of experimentation and design revisions to determine the proper actuators, sensors, and controllers to best accomplish the task. The final design is powered by a Raspberry Pi Zero and a Teensy 3.5. It uses a strong NEMA 11 stepper motor for tuning and works with a standard clip-on contact mic that can be simply clamped to any stringed instrument. Overall, all changes and solutions to the problems presented were addressed and implemented which allowed for accurate and quick tuning.

Table of Contents

List of Figures

Problem Description

Tuning is one of the first techniques taught to beginner violinists. The violin is a four-stringed instrument that can be tuned via two methods: through its pegs for major tuning and through its fine tuners for more precise and controlled tuning. These fine tuners are located on the bottom portion of the violin, resting above the tail piece. Each fine tuner controls its respective string (G, D, A and E). The problem addressed is the development of a handheld device aimed at helping help beginner violinists tune by automatically adjusting the violin fine tuners to its desired pitch. The target audience is beginner and less experienced violinists to help improve musical pitch detection. Ultimately, this device is designed to help students obtain what is known as perfect pitch.

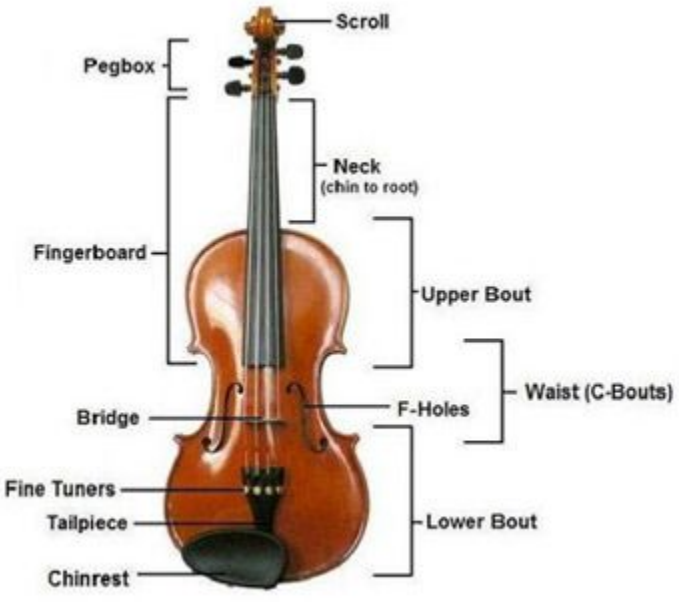


Figure 1. Violin Body Diagram

Operational Description

The automatic tuner will receive audio data through a condenser or contact microphone and will use that data to fully tune a violin by articulating the fine-tuner screws using a stepper motor. This device will have multiple profiles that will enable it to be used on different instruments with different string tunings. The user will be able to select an instrument preset and then tune each string one by one by placing the device on the fine-tuner screw and plucking the string. When the particular string emits the pitch that is correct for that instrument profile, the closed-loop tuning process will stop and return to the string selection screen. Figure 2 is a state diagram that will help visualize this procedure.

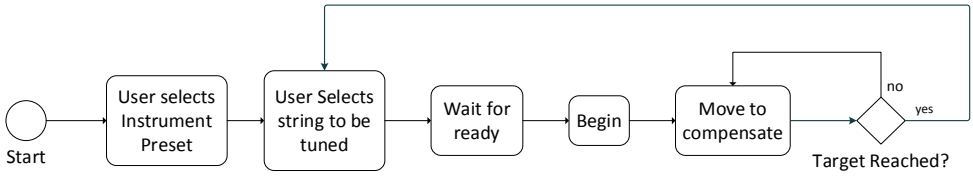


Figure 2: Operational State Diagram

Completed Work

Firmware

The core control system of this device runs on a Teensy 3.5 microcontroller (Figure 3). This device has a 32-bit 120 MHz MCU and is well suited to actuator operation and real-time audio analysis. The Teensy 3.5 is responsible for the core functionality of the device, performing the pitch detection and motor control needed by the closed-loop control system.

Figure 3: Teensy 3.5 Development Board

Kalman Filter

The pitch detection method used by the device is called the Yin Algorithm [1]. The process is based on traditional autocorrelation and has the advantage of being time efficient and high-performance when compared to a more traditional FFT. With that said, one primary issue with this algorithm is the presence of detection noise. Figure 4 shows a collection of 500 measurements taken of a sine wave at 440 Hz using the Yin pitch detection process from the Teensy Audio System library.

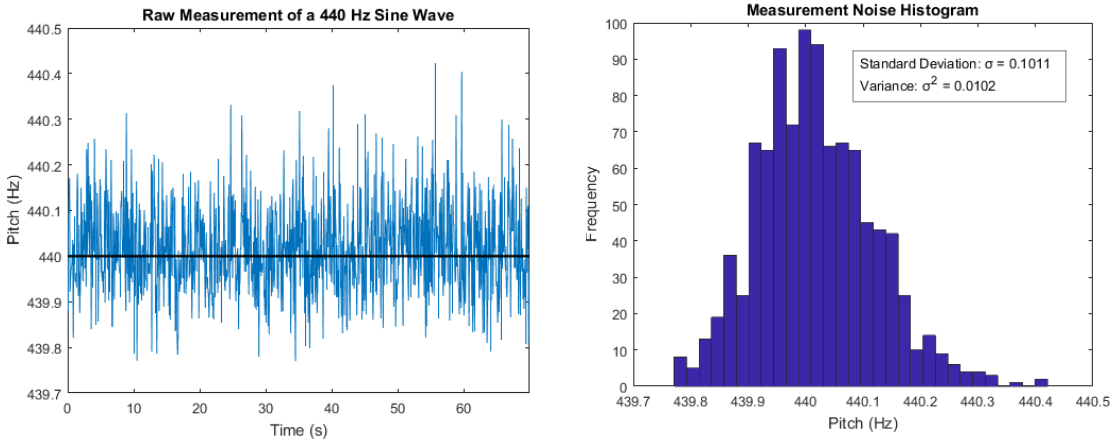


Figure 4: Measurement Noise of a 440 Hz sine wave from Yin Process

The collected data shows that the measurement fluctuates quite significantly even when the ground truth remains constant. Not only does this make accurate measurement difficult, it also can be detrimental to closed-loop systems which are sensitive to sudden measurement changes. This is where smoothing algorithms are useful. There are many algorithms than can be used to smooth noisy data from telemetry analysis, the most common one being a gaussian smoothing filter shown in Figure 5.

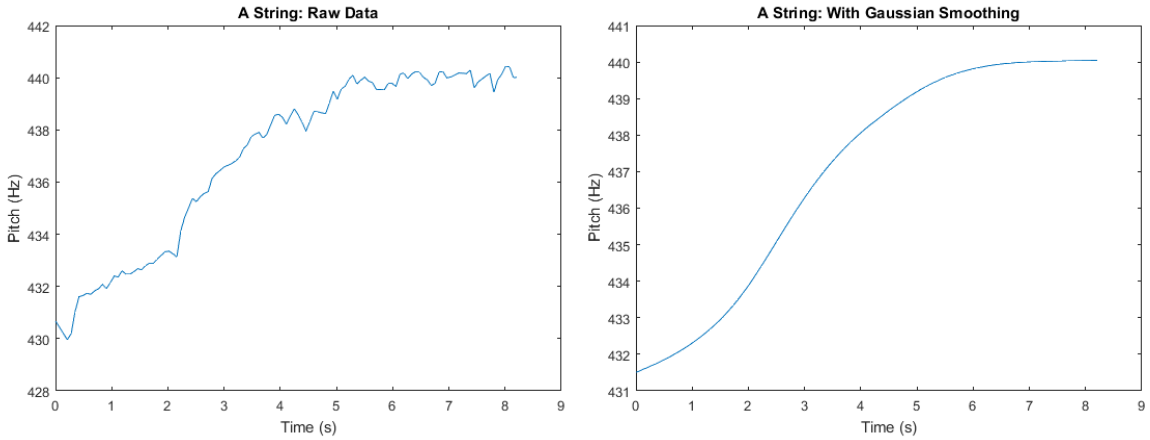


Figure 5: Effect of Gaussian Smoothing on Pitch Data

Although this works well for purely visualization purposes, this kind of filter is poorly suited to real-time applications since it relies entirely on moving a filter across already measured data. When using live, continuously changing telemetry for control purposes, it is necessary to use an algorithm that can deal with the relative uncertainty of a measurement. A Kalman filter [2] is an excellent solution to this issue. Unlike other filters that rely on a significant amount of past data, a Kalman filter simply uses information about the current and previous states to make a prediction of what the system is going to do next. The Kalman process features many variations suited to different tasks, however, they all follow a similar procedure:

(a) Predict the next state $$x(t) = F * x(t-1) + B * u \quad \textbf{(1)}$$

(b) Predict the next covariance $$P(t) = F * P(t-1) * F^T + Q \quad \textbf{(2)}$$

(c) Compute the Kalman gain $$K = P(t) * \frac{H^T}{H * P(t) * H^T + R} \quad \textbf{(3)}$$

(d) Update the state estimate $$x(t) = x(t) + K * (measurement(t) - H * x(t)) \quad \textbf{(4)}$$

(e) Update the covariance estimation $$P(t) = (I - K * H) * P(t) \quad \textbf{(5)}$$

Where x is the state, P is covariance, K is the Kalman gain, R is the sensor noise covariance and Q is the process noise covariance. Written into code, this process can be expressed as follows:

```
float R = 6.1E-3;  // Observation Noise Covariance
float Q = 7e-4;    // Process Noise Covariance
float Pc = 0.0;
float G = 0.0;     // Kalman Gain
float P = 1.0;
float Xp = 0.0;
float Zp = 0.0;
float Xe = 0.0;

float StepFilter(float measured_pitch){
  // Shift Calculated Predictions
  Xp = Xe;
  Zp = Xp;

  // Calculate PC covariance
  Pc = P + Q;

  // Calculate Gain
  G = Pc / (Pc + R);
  P = (1-G)* Pc;

  // Calculate Estimate
  Xe = G*(measured_pitch-Zp) + Xp;

  return Xe;
}
```

The sensor (observation) noise covariance and the process noise covariance are the two values used to “tune” the filter. The sensor noise covariance can be attained by taking many measurements from a sensor where the ground truth is known, calculate the state vectors, and finally calculate the sample covariance on that set of vectors. For a SISO (Single-Input Single-Output) system like the pitch detection process, this value can be obtained simply by performing many measurement samples of a fixed real value and calculating the variance of the set. Figure 4 showed this derivation for a pure sine wave and Figure 6 shows this measurement for an A played on an actual violin.

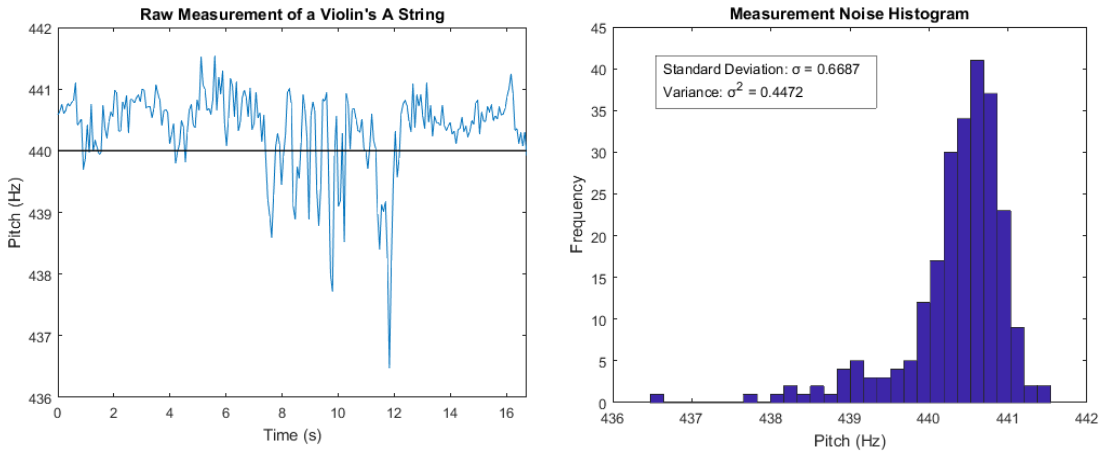


Figure 6: Measurement Noise of a Violin's A from Yin Process

The process covariance can be obtained using a similar method by taking measurements at multiple real-life values, however, this value will usually be further adjusted by hand, since it has a significant bearing on the reaction time of the filtered values. Tuning a Kalman filter has an inherent tradeoff: the more aggressive the smoothing, the greater the delay in the system. Because of this, the sensor and noise covariances must be set such that the system can react fact enough while also applying sufficient smoothing. Figure 7 shows a comparison between the raw pitch data and the filtered data, note the slight delay between the raw and filtered pitch data.

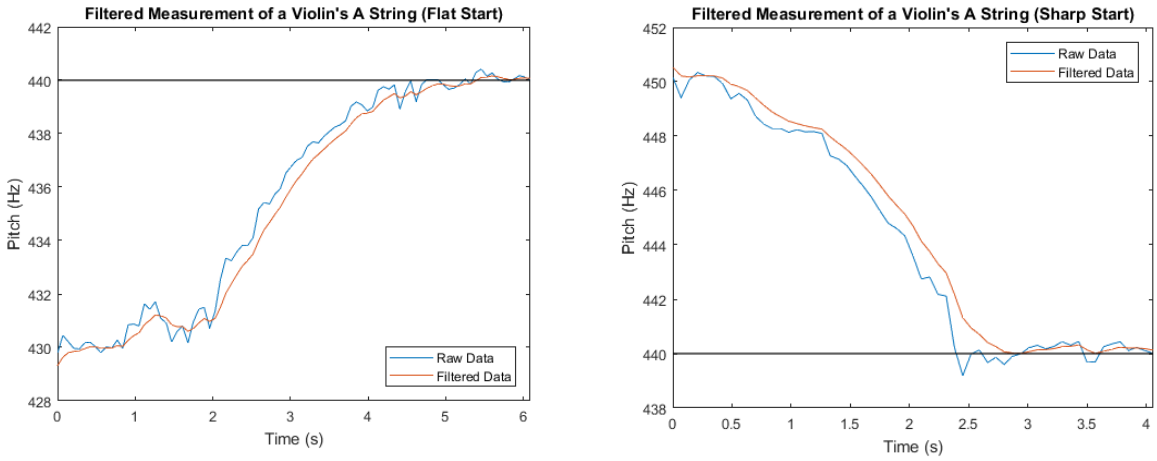


Figure 7: Result of Kalman Filter being used during tuning process:

PID Control

The tuning process of this system can be considered a closed-loop, error-driven feedback system. In this context, the output of the system $Y(s)$ is the actual measured pitch of the violin string and the input of the system $R(s)$ is a set point which acts as the target. Error-driven here means that the movement of the actuator is determined by the difference $E(s)$ between the measured output and the setpoint. Figure 8 shows a block diagram of this kind of system.

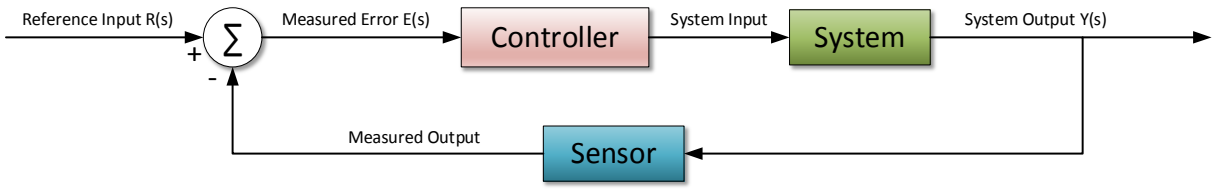


Figure 8: Block Diagram of a Negative Feedback Control System

A common type of controller used in this configuration is known as a PID controller (Figure 9). This control mechanism works by generating an actuator signal that is the sum of the system's error, integral, and derivative multiplied by constant gains; K_p, K_i, and K_d [3].

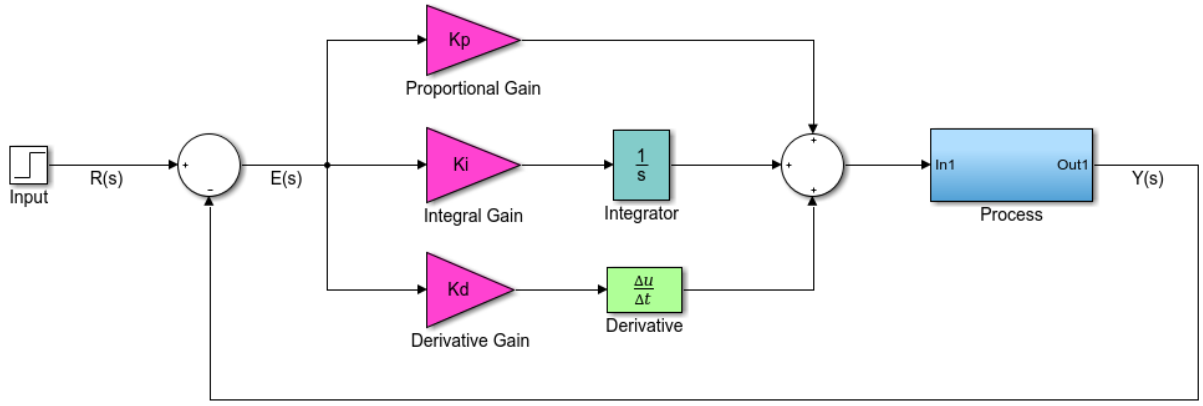


Figure 9: PID Block Diagram

These three parameters can be tuned to achieve different rise times, settling times, and damping ratios. From a high-level, the proportional gain K_p can be the magnitude of pull towards the setpoint. High proportional gains will lead to a faster response time, however, setting this value too high can lead to excessive overshoot, and undamped characteristics such as oscillation. To counteract this, the derivative gain K_d can be thought of as a vector that pulls away from the setpoint and can help dampen the system. Proportional and derivative control alone works well, but it can lead to the buildup of an error offset e_0. The integral gain K_i can account for this by ensuring that the system does not spend too much time on one side of the setpoint. The implementation of this controller in the Laplace and time domains are given in (6) and (7).

$$G(s) = K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s} \qquad \textbf{(6)}$$

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\mathrm{d}\tau + K_d \frac{d}{dt} e(t) \qquad \textbf{(7)}$$

These equations hold for a continuous time system; however, a microcontroller operates in discrete time. The system must make iterative calculations one after another at a fixed rate Δt. Here, first-order derivatives are made by backward finite differences. Equation 3 shows the PID equation migrated to the discrete domain, where the value k represents the calculation number, $T_i = \frac{K_p}{K_i}$, and $T_d = \frac{K_d}{K_p}$.

$$u(t_k) = u(t_{k-1}) + K_p\left[\left(1 + \frac{\Delta t}{T_i} + \frac{T_d}{\Delta t}\right)e(T_k) + \left(-1 - \frac{2T_d}{\Delta t}\right)e(t_{k-1}) + \frac{T_d}{\Delta t}e(t_{k-2})\right] \quad \textbf{(8)}$$

Equation 3 shows that the integral and derivative components are nothing but sum and difference equations. Because of this, the code for this type of controller is extremely simple, and can be abstracted into the pseudocode below, wherein the function *calculate()* is executed at a frequency of $\frac{1}{\Delta t}$.

```
previous_error = 0
error_sum = 0

void calculate(){
    error = setpoint - measured_value
    error_sum += error
    output = Kp*error + Kd*(error - previous_error) + Ki*(error_sum)
    previous_error = error
}
```

Closed-Loop Tuning System

The tuning process itself is at the very heart of the violin tuner and encapsulates its core functionality. It operates using a PID compensation system discussed previously and uses the YIN algorithm to calculate the fundamental frequency of the violin string in real time. In this system, the process is the violin string whose input is the position of the fine-tuner screw and output is the fundamental frequency in Hertz (Hz), the input frequency is the setpoint, and the actuator is the stepper motor.

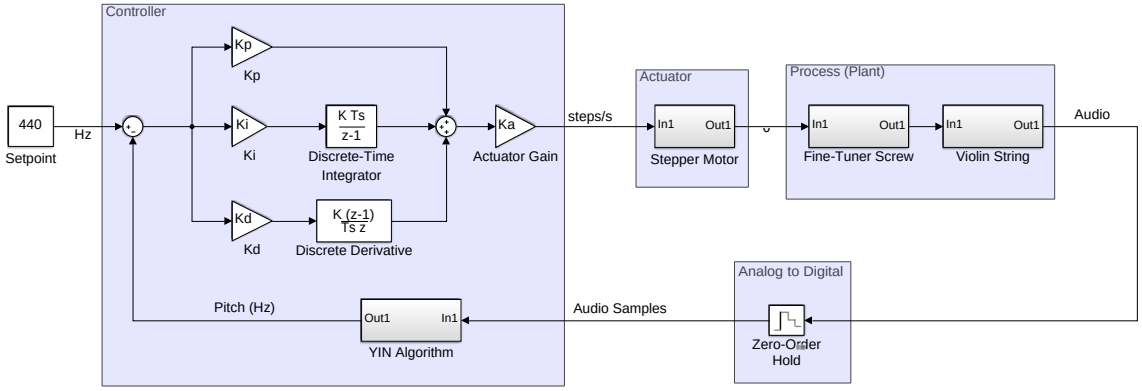


Figure 10: Automatic Tuner System Block Diagram

Applying this algorithm to the hardware was simply a matter of scheduling the PID calculation to occur as fast as possible and adjusting the stepper motor's angular velocity in real time.

To evaluate the performance of this control system, each string was tuned to their proper value from above and below the setpoint to generate a set of step response plots (shown in Figure 11, Figure 12, Figure 13, and Figure 14). The G, D, and A string were all initialized from a frequency 10 Hz away from the target, while the E string was initialized from 20 Hz away due to its steep tuning characteristics. Each plot has a dotted line to indicate the setpoint, along with upper and

lower bounds to indicate the 1 Hz human-perceptible range for the target. For ease of analysis, each of these response curves have the gaussian filter applied to the pitch data.

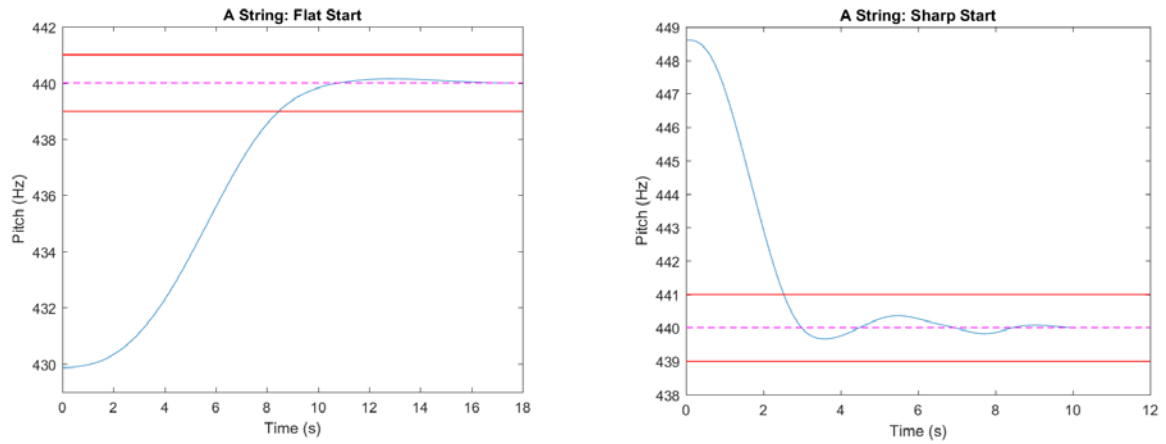


Figure 11: A String Step Response Plots

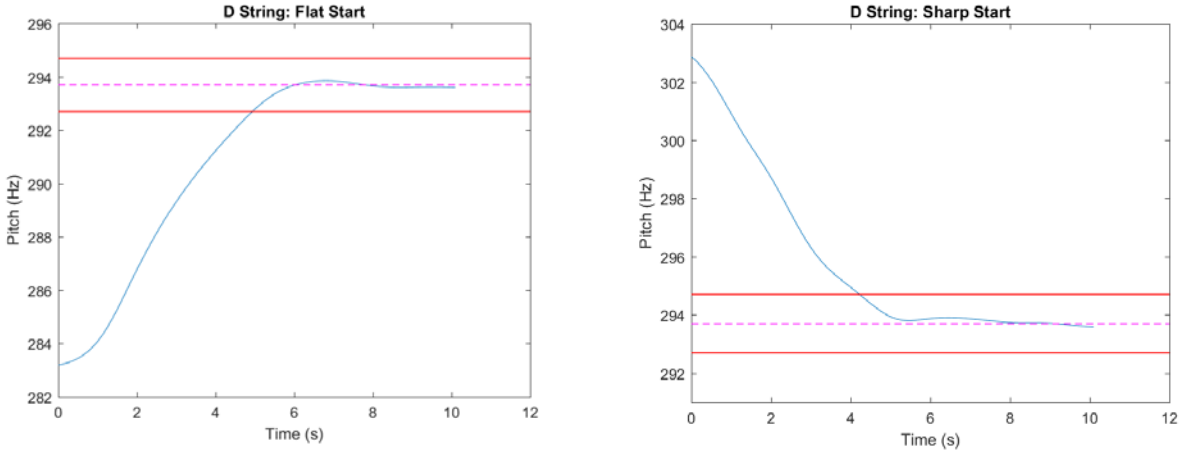


Figure 12: D String Step Response Plots

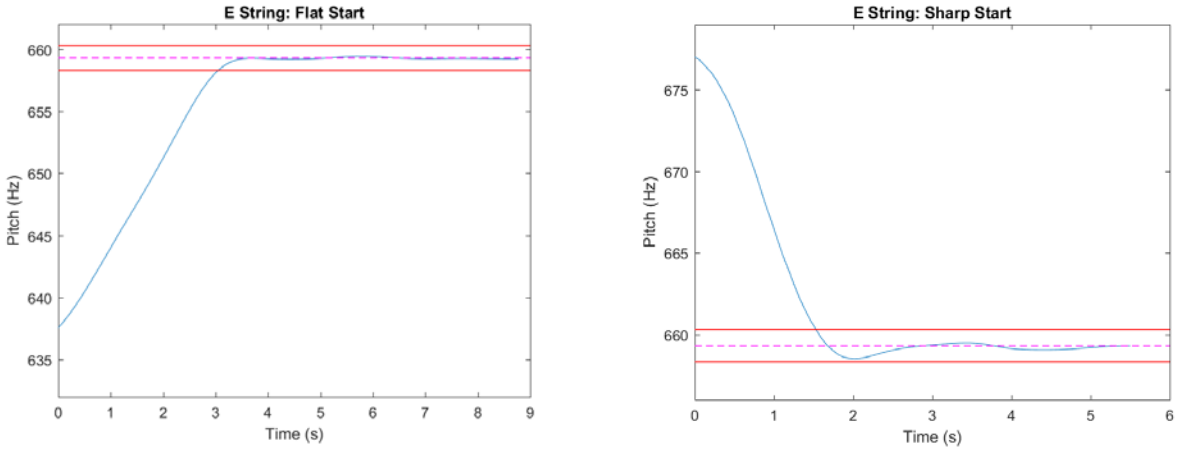


Figure 13: E String Step Response Plots

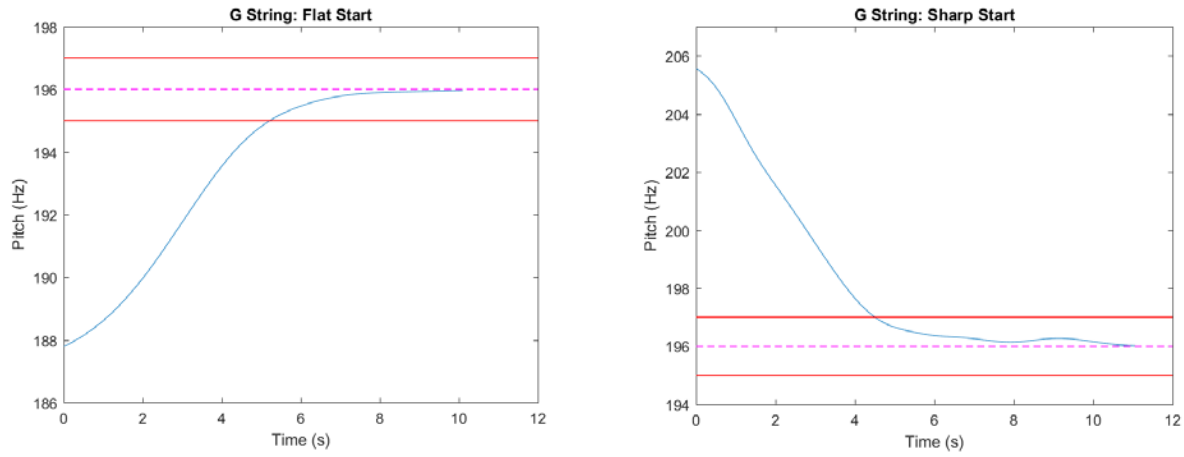


Figure 14: G String Step Response Plots

Looking at the step response plots shown previously, there does appear to be a difference in performance depending on whether the starting pitch is above or below the target. Strings which are started flat (i.e. below the target) approach the setpoint in a steady manner and dampen quite quickly. Conversely, this kind of behavior is not always observed when strings are started sharp; the E string (Figure 13) displays considerable overshoot during a downward step response, and the A string (shown in Figure 11) exhibits that as well as a more significant oscillatory behavior. This hysteresis may be due to the nature of how tension determines pitch. When the string is adjusted to make its tone sharper the actuator is going against the force of tension to further tighten the string, and the opposite is true when adjusting a string to make it flatter. The dampening of the tuning curve is greater when tuning up since the device is fighting tension, but since the actuator moves with tension on the way down, the system becomes less damped. Overall, all curves exhibit the traditional sigmoid characteristic that is expected of PID systems, and each string manages to complete its tuning process in under 10 seconds under the worst conditions. Table 1 shows a full evaluation of the closed-loop performance.

Table 1: Tuning Process Rising Step Info

	Rise Time (s)	Settling Time (s)	Settling Min (Hz)	Settling Max (Hz)	Overshoot (Hz)	Peak (Hz)	Peak Time (Hz)
A String	5.691	9.187	439.016	440.153	0.0349	440.154	12.812
D String	3.815	5.314	292.712	293.855	0.0528	293.855	6.824
E String	2.418	2.806	657.399	659.429	0.0196	659.429	5.779
G String	4.469	6.299	195.203	195.968	0	195.968	10.096

Interface

The Teensy 3.5 communicates with the Raspberry Pi (RPi) Zero over UART [1] under a master-slave relationship. A custom serial protocol was designed using the open source CmdMessenger library which allows the RPi Zero to send commands and data requests to the Teensy. Command packets are all structured in the following format:

[COMMAND ID],{OPR1},{OPR2},…,{OPRn};\r\n

While response packets are all of the form:

[RESPONSE ID],[STATE],[ERROR],{ OPR1},{OPR2},…,{OPRn};\r\n

With this format, the master will be updated with the state information of the microcontroller during every send/receive operation. Every command has a specific response ID coupled with it, so the master will always be able to appropriately decipher the meaning of the packet's operands. Table 2 shows the various commands which can be used to communicate with the microcontroller. This protocol structure was chosen because it allows for simplistic flow control, however, if more time were allowed, a fully asynchronous protocol which would allow for streams would be more ideal, as it would enable the Teensy to send information to the master at any point in time. The Kalman filter, Closed-Loop PID system, and transport layer make up the primary portions of the Teensy's Firmware, which can be reviewed in its entirety in Appendix B (pIII).

Table 2: Serial Command Protocol

ID	Name	Description	Operand(s)
0	GetStatus	Getter operation for the status and error	n/a
1	GetStatusAck		
2	GetPitchData	Getter operation for the latest measured pitch	n/a
3	GetPitchDataAck		
4	SetTarget	Setter operation for the target pitch (string)	Target, Pitch
5	SetTargetAck		
6	BeginTuning	Command to begin the tuning process	n/a
7	BeginTuningAck		
8	StopTuning	Cancel tuning process if it is running	n/a
9	StopTuningAck		
10	Calibrate	Calibrate Actuators	n/a
11	CalibrateAck		
12	ToggleRawStream	Toggle pitch teleetry stream for debug	n/a
13	ToggleRawStreamAck		

[1] Universal Asynchronous Receiver/Transmitter

Hardware

General Architecture

From a high-level, there are four main sections of the hardware architecture; the front-end, the controller, the actuators and the sensors. The sensor for this device is a piezo contact microphone that is driven by an amplification circuit that scales the signal to between 0 and 1.2 V. The actuator is a small-form geared stepper motor that is driven by an H-bridge. Everything involved in the control system is operated by a Teensy 3.5 microcontroller, which receives commands over serial. The top-most layer is a Raspberry Pi Zero, which manages the user interface (UI) and sends control commands to the microcontroller unit (MCU) below it.

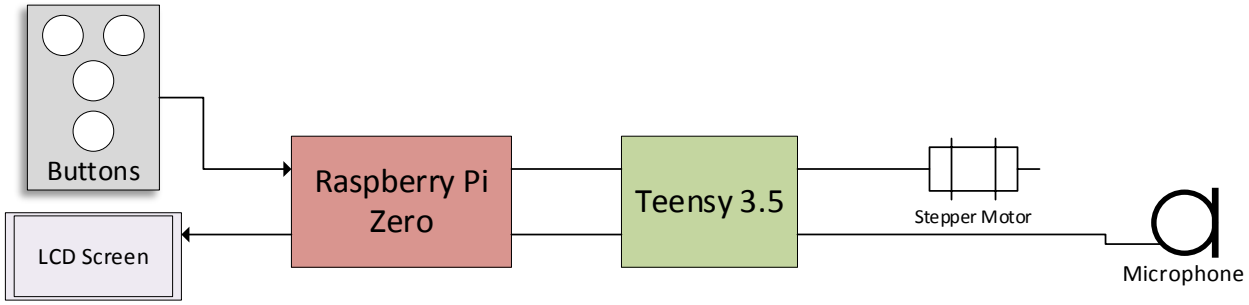
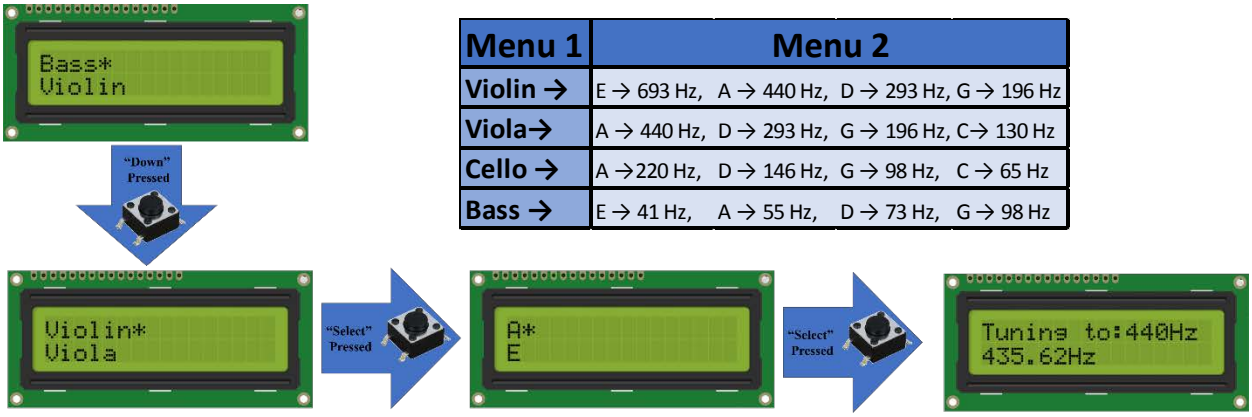


Figure 15: Simplified Hardware Architecture

Raspberry Pi

The device facilitating the operation of the front end for the tuner was chosen to be a Raspberry Pi Zero. This single board computer, in implementation, would be utilized in the communication between the front and back end systems and the aggregation, processing, and display of data from the front end into an intuitive graphical user interface (GUI). Both the Teensy and the Pi will have an identically ordered enumerated list of commands, which are detailed in Table 2, that can be both sent and received onto which callback functions can be attached for correct software handling. The software for the Pi (Shown in Appendix C) was created as a python script relying mainly on both the threading and PyCmdMessenger libraries. From the threading libraries two primary threads were set up, one for communications between the Pi and Teensy and another for the GUI. The communications thread will always be updating and mainly be responsible for state synchronization between the front and back end alerting the user to important state changes and errors when needed. The GUI thread will be responsible for relaying all data necessary for the user to see to the LCD screen while also processing inputs from the user and important information from state synchronization thread. The Pi's GPIO will be used to connect to the screen as well as buttons for use in the thread.

Graphical User Interface



Menu 1	Menu 2
Violin →	E → 693 Hz, A → 440 Hz, D → 293 Hz, G → 196 Hz
Viola→	A → 440 Hz, D → 293 Hz, G → 196 Hz, C→ 130 Hz
Cello →	A →220 Hz, D → 146 Hz, G → 98 Hz, C → 65 Hz
Bass →	E → 41 Hz, A → 55 Hz, D → 73 Hz, G → 98 Hz

Figure 16: GUI Menu Operation and Linked List Structure

The Graphical User Interface is displayed on an LCD screen connected to the Pi Zero through six digital pins for updating the data on screen and with two power bus lines connected to the pixel contrast and backlight. The display will allow the user to choose an instrument to tune, followed by a selection for the string to be tuned. A representation of this menu structure can be seen in Figure 16. User operation of the GUI was facilitated with four buttons wired to digital pins that were pulled up to 5V through the Pi's internal resistors. The positioning of the buttons can be seen in Figure 18 and their relative position to the LCD screen, which will be affixed to the enclosure, can be best observed in Figure 19. A thread was created to both handle user inputs and update the screen when necessary. This GUI thread would coordinate with the state synchronization thread that communicates to the Teensy using event objects from the threading library. Keeping these two operational loops separate was very important because the tasks of each may not coincide with each other synchronously since the state synchronization thread needs to constantly send out commands and the GUI thread is only tasked with updates to the screen when needed. The GUI thread's operation will access a linked list containing the menu items displayed to the user with each element made from a tuple containing the menu element name followed subsequent menu options or data for that element as an array. A visual representation of this linked list can also be seen in Figure 16 as well. When a button is pushed, the 5V pin is pulled to ground which triggers a callback function for that pin to raise an event flag. The GUI thread keeps track of the menu depth and index of the item in said menu in order to properly handle event flags raised by button presses. The buttons available to the user are labeled as "Select", "Back/Cancel", "Up", and "Down". The up and down buttons will increment or decrement the current menu index when the event is raised. The select and back/cancel buttons will increment or decrement the menu depth index respectively. If the select event flag is raised while the GUI thread is at the highest menu depth (listing the notes of the strings), a tuning event flag will be raised causing the state synchronization thread to send commands to the Teensy to prepare for and begin tuning.

Microphone Circuit

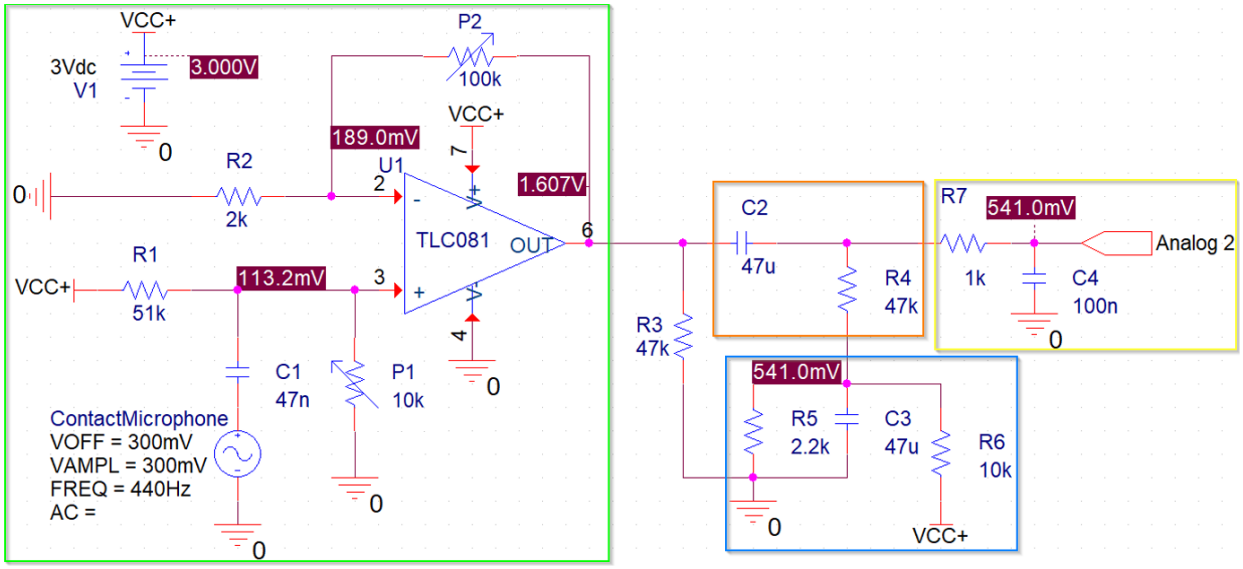


Figure 17: Analog Microphone Circuit

The bridge-pickup microphone is the input of the violin, and connects to the first stage of the microphone circuit, a differential amplifier. This amplifier was designed to produce a gain sufficient to swing the signal to cover the full input range of the Teensy's ADC. Doing this will allow for the signal to be processed with the highest resolution. The schematic of the differential amplifier stage can be seen in the green box of Figure 17. The circuit consists of a Texas Instruments TLC080A operational amplifier, one capacitor, two resistors, and two potentiometers. The output of the contact microphone will be a differential signal, this is referenced to ground by attaching one of the signal wires to ground and the other to the input of the circuit. The operational amplifier takes the signal from the contact microphone after it passes through C1. In addition to the microphone signal being fed into the amplifier input a DC offset is also added. The C1 capacitor will block any DC voltage from going across the microphone so as not to damage it when the signal is offset. The 51 kΩ resistor R1 and 10 kΩ potentiometer P1 are responsible for DC biasing the input signal going to the op-amp to offset the signal in order to shift the negative swing of the signal to being only positive; in the case of a 3.3 V supply the offset can be set anywhere from 0-0.541 V when changing the value of P1. This shift is necessary due to the op-amp only being supplied a positive voltage therefore it will not have the ability to swing the output signal below 0 V. This voltage divider circuit then feeds the DC offset audio signal into the non-inverting input of the op-amp. The 2 kΩ resistor R2 and 100 kΩ potentiometer P2 control the gain of the op-amp and can allow for a variable gain anywhere from 0-50V/V when changing the value of P2. The ideal values for the potentiometers P1 and P2 should be ones that allow for the offset of the signal to be slightly higher than the magnitude of the negative swing and gain to keep the peak to peak voltage of the signal just under 1.2V. The value of both the offset and gain will generally stay the same for most string instruments and have been included in the circuit to ensure the device can accommodate any orchestral string instrument that may use it.

After the differential amplifier stage, the signal should be only positive with a peak to peak voltage around 1.2V. At this point the signal will enter a high-pass filter, which can be seen inside of the orange box in Figure 17 This is constructed as a standard passive RC high-pass filter consisting of a series capacitor C2 and a shunt resistor R4. The cutoff frequency for this filter was selected at close to 1Hz which can be seen in Equation 9.

$$f_{c\_HP} = \frac{1}{2\pi R_4 C_2} = \frac{1}{2\pi * 47k\Omega * 47\mu F} = 0.072Hz \qquad \textbf{(9)}$$

This will effectively eliminate any DC offset output from the op-amp and allow any instrument frequencies to pass through. This is crucial in order to accomplish the next stage which is another voltage divider circuit that adds a DC offset to the signal, which is shown in a blue box in Figure 17. This offset is accomplished by choosing the ratio of resistors R5 and R6 and will center the signal at about 0.6V in order to ensure the amplified signal be able to have the maximum voltage swing possible, ideally from 0V to 1.2V, without clipping the signal when being read into the Teensy's Analog 2 pin.

With the signal now swinging from 0V to 1.2V it enters the last stage of the microphone circuit, a low-pass filter. This filter is constructed as a standard passive RC low-pass filter with a series resistor R7 and shunt capacitor C4. The circuit is designed with a cutoff frequency of about 1600Hz, and the resistor and capacitor values were chosen to match up with this selected cutoff (10).

$$f_{c\_LP} = \frac{1}{2\pi R_7 C_4} = \frac{1}{2\pi * 1k\Omega * 100nF} = 1591.55Hz \qquad \textbf{(10)}$$

This filter will ensure that any noise that may have been potentially picked up by digital hardware such as the Teensy, motor controller, or external electromagnetic interference will be filtered out and not affect the audio signal going into the Teensy's analog pin.

Power Supply

In order for the device to be truly portable, it was necessary to implement battery power. A lithium polymer (LIPO) battery was the best choice because they are easily rechargeable and compact. The LIPO battery being used is a one-cell LIPO at 3.7 V nominal with a 3000 mAh capacity. This battery has a C rating of 2, meaning that it can discharge at 2*3000 mAh or 6000 mA max burst, and a charge C rating of 1, meaning it can handle a maximum charge current of 1*3000 mAh or 3000 mA. The charge, passthrough, protection, and boost circuit being used is the opensource PowerBoost 1000c from Adafruit [4]. This board has all the desired features of a battery management controller; It takes a LIPO battery voltage that ranges from 2.9-4.2 V and boosts it to 5.2 V, and has under/over voltage and short circuit protection. The PowerBoost charges the battery at 1 A while having a passthrough for the 5 V of the micro USB charging cable allowing the tuner to be used while charging.

Motor

Several different motors were used as actuators during the length of the development process. Ultimately, stepper motors were chosen since they are easy to precisely control at lower speeds and will move consistently even with open-loop control. The first prototype utilized a geared 32:1

28byj-48 stepper motor. This motor had the advantage of being very small and also low-cost, however, its overall torque and speed resulted in performance issues such as long tuning times and actuator suppression. In addition, the motor was unipolar, which made it difficult to control with more robust controller boards. To counter these problems, the final version uses a NEMA 11 stepper motor, which can operate at a much higher torque and move at much higher speeds. There were a few downsides to this choice, including increased cost and larger size, but the biggest downside was that the negation of gears meant a loss of precision in motor movement. This setback was supplemented by switching to a more advanced controller called the MP6500, which allowed for microstepping all the way down to an eighth step if needed. For this device, quarter steps were sufficient, which allowed for effectively 800 intervals per revolution.

Circuit Layout and PCB Design

The printed circuit board (PCB) was designed using the amplifier circuit and the interconnect layout in Figure 18. The reasons the PCB was created is that loose connections along with signal and ground loops were causing trouble with design, development, and testing of the tuner. The PCB also gave us a set footprint to design the housing and a way to interface the entire board with a single micro USB port. The schematic of the PCB is shown in Figure 18 is the design used in the prototype that has been created for spring term. The PCB was designed using EasyEDA [5] online PCB design tool that is linked to JLCCPCB [6] a manufacturing facility that prints circuit boards.

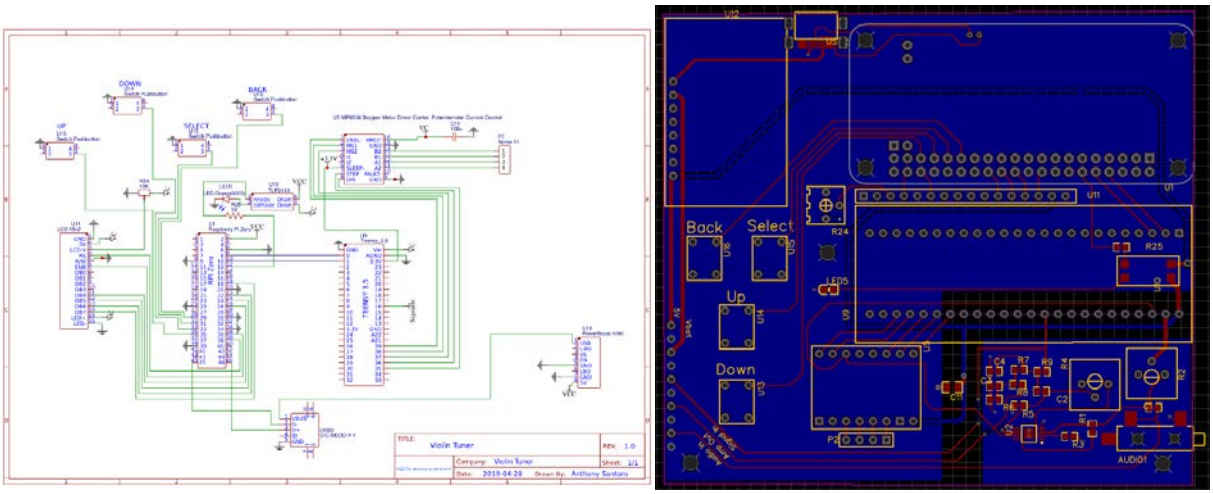


Figure 18 Component connection schematic (Left) PCB layout (Right).

Enclosure

The case for the tuner was designed certain key factors in mind: it must be easy to hold and operate by a young student, efficient to manufacture, intuitive to use, and be designed in a way which allows the user to see the screen during tuning. The case is designed as two halves that are screwed together, allowing the enclosure along with all its components to be injection molded for easy and reliable mass manufacturing. Setting the LCD screen above the control buttons allows the user the familiarity of hand placement below the screen eye level like in hand-held gaming devices. The buttons are imprinted with the function they provide: SEL for select, BAK for back, and up and down arrows for scrolling. The motor is mounted in line with the primary holding hand

to offset the torque for more accurate placement. With the motor mounted on the back, the screen can be easily seen during the tuning process. Fully dimensioned drawings for each part of the case are located in Appendix D.

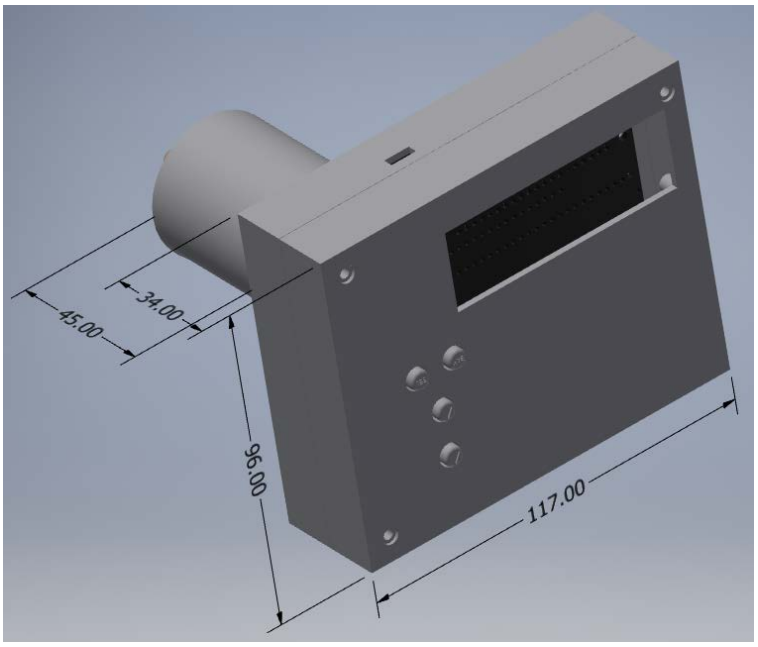


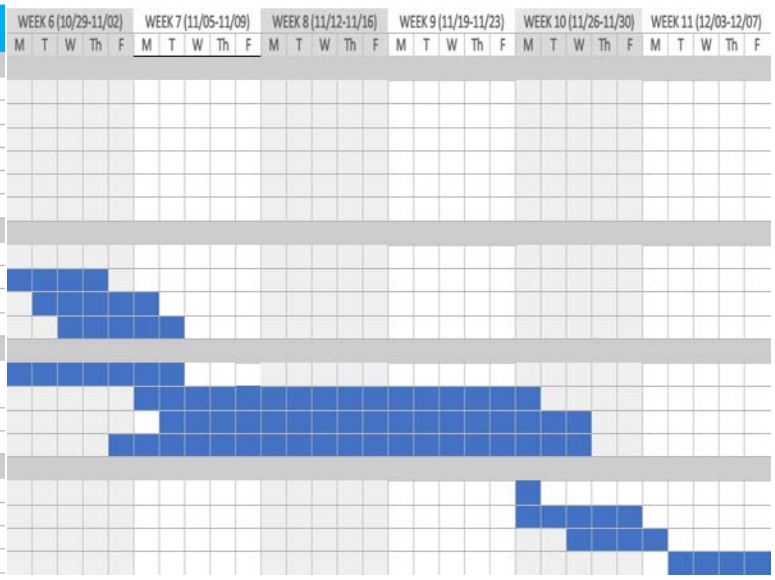
Figure 19: 3D Render of device Back and UI Elements

Work Schedule & Final Timeline

GANTT CHART FALL TERM '18

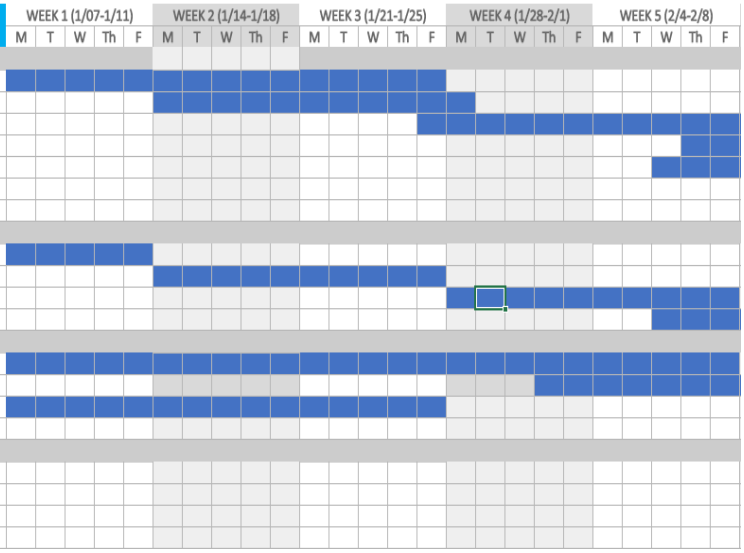
TASK NAME	START DATE	END DATE	DURATION (WORK DAYS)	WEEK 1 (9/24-9/28)					WEEK 2 (10/01-10/05)					WEEK 3 (10/8-10/12)					WEEK 4 (10/15-10/19)					WEEK 5 (10/22-10/26)				
				M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F	M	T	W	Th	F
Initial Brainstorm																												
Initial Brainstorm Process	9/24	9/28	5																									
Hold First Meeting	10/1	10/1	1																									
Decide Quailty Goals	10/1	10/2	2																									
Evaluate Interest In Project	10/3	10/5	3																									
Review With Advisor	10/12	10/12	1																									
Submit to Coordinator	10/18	10/18	1																									
Background Research																												
Exisiting Patent Search	10/22	10/26	5																									
Research on String Physics of the Violin	10/26	11/1	5																									
Research on Potential Microcontrollers	10/30	11/5	5																									
Research on Code Involving Pitch Detection	10/31	11/6	5																									
Control System and Construction																												
Test Rig Design & Build			0																									
Model Derivation			0																									
Pitch Detection Development			0																									
Control System Design			0																									
Presentation Preparation for Fall Term																												
Discuss Presentation with Advisor	11/26	11/26	1																									
Prepare Presentation and Report	4/8	4/10	2																									
Rehearse Presentation	11/28	12/3	4																									
Revise Presentation	12/4	12/7	4																									

TASK NAME	START DATE	END DATE	DURATION (WORK DAYS)
Initial Brainstorm			
Initial Brainstorm Process	9/24	9/28	5
Hold First Meeting	10/1	10/1	1
Decide Quality Goals	10/1	10/2	2
Evaluate Interest In Project	10/3	10/5	3
Review With Advisor	10/12	10/12	1
Submit to Coordinator	10/18	10/18	1
Background Research			
Exisiting Patent Search	10/22	10/26	5
Research on String Physics of the Violin	10/26	11/1	5
Research on Potential Microcontrollers	10/30	11/5	5
Research on Code Involving Pitch Detection	10/31	11/6	5
Control System and Construction			
Test Rig Design & Build			0
Model Derivation			0
Pitch Detection Development			0
Control System Design			0
Presentation Preparation for Fall Term			
Discuss Presentation with Advisor	11/26	11/26	1
Prepare Presentation and Report	4/8	4/10	2
Rehearse Presentation	11/28	12/3	4
Revise Presentation	12/4	12/7	4

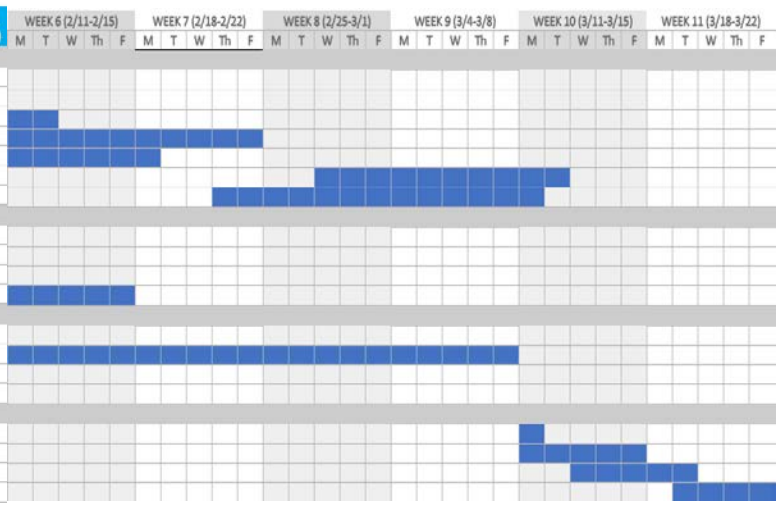


GANTT CHART WINTER TERM '19

TASK NAME	START DATE	END DATE	DURATION (WORK DAYS)
Control System			
Hardware Evaluation	1/7	1/25	15
Feedback System	1/14	1/28	11
Process Tuning	1/25	2/12	13
Algorithm Development	2/7	2/22	12
Communication Protocol	2/6	2/18	9
Front-End Development	2/26	3/12	11
Hardware and Control-Chain Integration	2/21	3/11	13
Research on Human Pitch Detection			
Meeting with Music Teacher and Colleagues	1/7	1/11	5
Conduct Experiments on Variability of Pitch	1/14	1/25	10
Derive Model for Acceptable Frequency Range	1/28	2/8	10
Conduct Experiments to Verify Model	2/6	2/15	8
Physical Design			
Concept Development	1/7	2/8	25
CAD Modeling	1/31	3/8	27
Actuator Research	1/7	1/25	15
			0
Presentation Preparation for Winter Term			
Discuss Presentation with Advisor	3/11	3/11	1
Prepare Presentation and Report	3/11	3/15	5
Rehearse Presentation	3/13	3/19	5
Revise Presentation	3/19	3/22	4

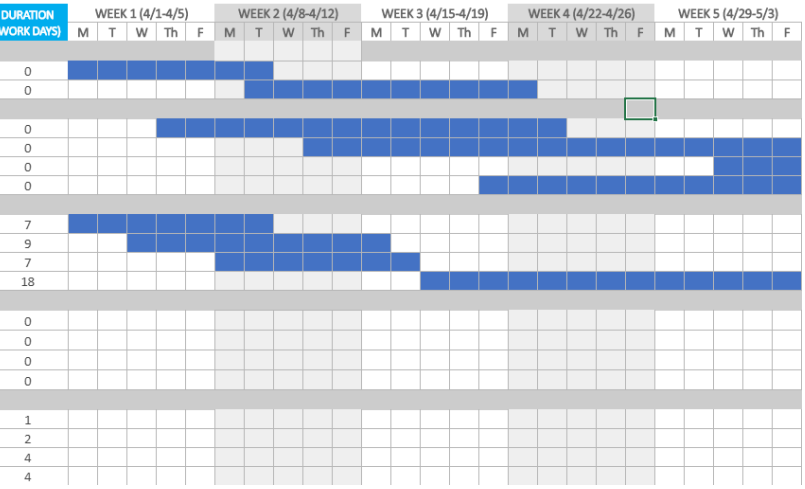


TASK NAME	START DATE	END DATE	DURATION (WORK DAYS)
Control System			
Hardware Evaluation	1/7	1/25	15
Feedback System	1/14	1/28	11
Process Tuning	1/25	2/12	13
Algorithm Development	2/7	2/22	12
Communication Protocol	2/6	2/18	9
Front-End Development	2/26	3/12	11
Hardware and Control Chain Integration	2/21	3/11	13
Research on Human Pitch Detection			
Meeting with Music Teacher and Colleagues	1/7	1/11	5
Conduct Experiments on Variability of Pitch	1/14	1/25	10
Derive Model for Acceptable Frequency Range	1/28	2/8	10
Conduct Experiments to Verify Model	2/6	2/15	8
Physical Design			
Concept Development	1/7	2/8	25
CAD Modeling	1/31	3/8	27
Actuator Research	1/7	1/25	15
			0
Presentation Preparation for Winter Term			
Discuss Presentation with Advisor	3/11	3/11	1
Prepare Presentation and Report	3/11	3/15	5
Rehearse Presentation	3/13	3/19	5
Revise Presentation	3/19	3/22	4

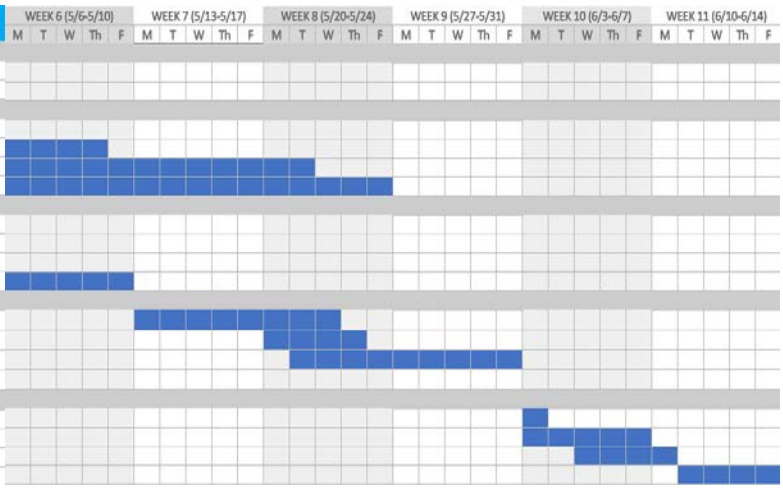


GANTT CHART SPRING TERM '19

TASK NAME	START DATE	END DATE	DURATION (WORK DAYS)
Control System			
Safety Features			0
Sensor Improvements			0
High-Level Software			
Intrument Profiles			0
Menus and Navigation			0
Tuning Process Visualization			0
Issue Handling			0
User Interface Improvement			
Graphical Design Template	4/1	4/9	7
Developing Framework	4/3	4/15	9
Enable Intuitive User Controls	4/8	4/16	7
Testing and Debug	4/17	5/10	18
Field Testing			
Gather a Sample Size of Target Audience			0
Record and Analyze Findings			0
Discuss Necessary Changes for Usability			0
			0
Presentation Preparation for Spring Term			
Discuss Presentation with Advisor	11/26	11/26	1
Prepare Presentation and Report	4/8	4/10	2
Rehearse Presentation	11/28	12/3	4
Revise Presentation	12/4	12/7	4



TASK NAME	START DATE	END DATE	DURATION (WORK DAYS)
Control System			
Safety Features			0
Sensor Improvements			0
High-Level Software			
Intrument Profiles			0
Menus and Navigation			0
Tuning Process Visualization			0
Issue Handling			0
User Interface Improvement			
Graphical Design Template	4/1	4/9	7
Developing Framework	4/3	4/15	9
Enable Intuitive User Controls	4/8	4/16	7
Testing and Debug	4/17	5/10	18
Field Testing			
Gather a Sample Size of Target Audience			0
Record and Analyze Findings			0
Discuss Necessary Changes for Usability			0
			0
Presentation Preparation for Spring Term			
Discuss Presentation with Advisor	11/26	11/26	1
Prepare Presentation and Report	4/8	4/10	2
Rehearse Presentation	11/28	12/3	4
Revise Presentation	12/4	12/7	4



Industrial Budget

	Expense	Cost pre Unit	Units	Total
Employee Costs	Electrical Engineer Salary (Yearly)	$65,000.00	4	$260,000.00
	Overhead (30% of Salaries)	$19,500.00	4	$78,000.00
	Employee Benefits (15% of Overhead)	$2,925.00	4	$11,700.00
	Health Insurance (/Person/Year)	$5,179.00	4	$20,716.00
	Total Cost (Year)			$370,416.00
Rent	Office Rent (/sq. ft)	$24.00	2000	$48,000.00
	Utilities (Yearly)	$4,300.00	1	$4,300.00
	Insurance (Yearly)	$1,200.00	1	$1,200.00
	Office Supplies (Yearly)	$2,500.00	1	$2,500.00
	Total Cost (Year)			$56,000.00
Hardware	Computer	$1,200.00	4	$4,800.00
	Power Supply	$500.00	1	$500.00
	Signal Generator	$400.00	1	$400.00
	Multi Meter	$400.00	1	$400.00
	Soldering Iron	$200.00	1	$200.00
	Oscilloscope	$900.00	1	$900.00
	Raspberry Pi Zero	$5.00	3	$15.00
	Teensy	$20.00	3	$60.00
	NEMA 11 Stepper Motor	$20.00	2	$40.00
	Miscellaneous (Components, 3D Printing, etc	$300.00	1	$300.00
	Total Cost (Year)			$7,615.00
Software Licenses	MATLAB	$2,150.00	2	$4,300.00
	Microsoft Office Professional	$440.00	4	$1,760.00
	Adobe Acrobat Professional	$450.00	4	$1,800.00
	OrCAD Design Suite	$10,660.00	1	$10,660.00
	Total Cost (Year)			$18,520.00
Subtotal (year)				$452,551.00
Total Budget for One Year				$520,433.65

Out-of-Pocket Budget

Item Name	Unit Price	Quantity	Total
USB to TTL Adapter 2-pack	$9.38	1	$9.38
STM32F103C8T6 2-pack	$10.99	1	$10.99
40 kg Tension Load Cell and HX711	$13.99	1	$13.99
Nema 17 Stepper Motor Bipolar	$13.99	1	$13.99
Contact Microphone Pickup 2-pack	$12.99	1	$12.99
Violin Acessory Kit	$24.37	1	$24.37
IEEE Locker Fall Term	$20.00	1	$20.00
Perf Boards	$5.30	1	$5.30
Single-row headers	$5.30	1	$5.30
Heat-shrink joiners	$5.30	1	$5.30
Retractable USB cable	$10.60	1	$10.60
DPDT Toggle Switch	$2.12	1	$2.12
IEEE Locker Winter Term	$30.00	1	$30.00
Teensy 3.5	$27.00	1	$27.00
MP6500 Stepper Motor Driver	$10.68	2	$21.36
5VDC 32-Step 1/16 Gearing	$7.94	2	$15.88
Teensy 3.5	$27.00	1	$27.00
NEMA 11 Stepper Motor 6.2v	$28.60	1	$28.60
NEMA 11 Stepper Motor 4.6v	$19.42	1	$19.42
PowerBoost 1000C	$21.95	1	$21.95
Lipo	$25.79	1	$25.79
28byj-48 Stepper Motor	$10.00	1	$10.00
Digikey Part Order	$47.25	1	$47.25
PCB Order	$12.45	1	$12.45
		Total	$421.03

Societal, Environmental and Ethical Impacts

Elementary and middle schools only have a short time slot or “period” dedicated to instrumental class (typically an hour or so). Much of the time used in class is dedicated towards tuning. One of the goals of this device is aimed to mitigate time spent tuning to allow more time for rehearsal and practice. Students can tune for themselves at the start of rehearsals without the aid of an instructor. This device will be constructed to help student musicians obtain “an ear” for tuning so violinists will gain the skill set to eventually tune independently.

The physical components that are required to build this automated tuner should not pose any significant environmental impacts. All the electronics found in the tuner are standard off the shelf components such as a small stepper motor, Arduino boards as well as wires found in most electronic devices, they are all certified by RoHS (Restriction of Hazardous Substances Directive) and the like. The housing is made out of PLA plastic, this plastic is a thermoplastic and can be easily recycled by chipping, melting, and extruding into a new product.

The tuner utilizes lithium polymer batteries to power the motor controller board as well as the NEMA motor. Due to state regulations, Li-Po batteries are classified as hazardous due to their lead content. The environmental impact associated with human toxicity is mainly due to the amount of

lead, cobalt, copper and nickel. Users are recommended to recycle the batteries when there is no charge left. That way if it is punctured, there will not be a possibility of combustion.

Understandably so, some may comment that using such a device long term will hinder the musician's ability to tune independently. While not its intended purpose, some musicians may rely on this tuner rather than utilizing their musical tuning ability. Tuners can be extremely helpful but like any other tool, it must be used correctly. This device is targeted towards very young and beginning violinists to help build the foundation to tune with perfect (or very close to) pitch. A more experienced violinist should not be using this device to tune.

Musical discouragement is a common attitude to have in one's career especially for amateur musicians. Starting out, many beginner violinists struggle to find the encouragement to continue for multiple years, especially without a teacher. Typically, this stems from many reasons but most commonly due to not producing a producing a quality sound. Naturally, this automatic violin tuner will help establish a fundamental basis for tuning independently.

Summary & Conclusion

While this device is only an early prototype, it is indicative of the kind of potential such a product could have. With a proper manufacturing process, the overall size of the device can be dramatically reduced, and the efficiency of the tuning process can be further improved through more advanced controllers. Should this project be revisited, one of the major changes would be neglecting the Raspberry Pi entirely and running everything directly on the Teensy microcontroller. Currently the violin tuner is optimized to register a pitch from bowing the string. While results have been efficient and accurate, this is not an ideal method to generate a pitch. Plucking the string does work, however the pitch does not sustain long enough for traditional closed-loop control to be effective. This is a feature that would be further explored should this design be revisited. The graphical user interface allows for easy navigation throughout various stringed instruments along with its own respective string set. Originally, the device was intended to have a touchscreen with a robust UI and useful graphics, however, even in its current state the simple interface is easy to use and intuitive for young students to easily grasp. The case that will house the exposed electronics in a production model will be light and portable for younger students to use. Overall, the goals of this project were met, and the resulting prototype is fully portable, easy to use, adaptable, and capable of robotically tuning orchestral instruments quickly and effectively.

References

[1] A. K. H. De Cheveigne, "YIN, A Fundamental Frequency Estimator for Speech," 1 May 2001. [Online]. Available: http://audition.ens.fr/adc/pdf/2002_JASA_YIN.pdf.

[2] "How a Kalman filter works, in pictures," 11 August 2015. [Online]. Available: https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/.

[3] K. H. Ang, G. Chong and Y. Li, "PID control system analysis, design, and technology," *IEEE Transactions on Control Systems Technology,* pp. 559-576, 2005.

[4] "Adafruit Powerboost 1000C | Adafruit Learning System," [Online]. Available: https://learn.adafruit.com/adafruit-powerboost-1000c-load-share-usb-charge-boost/overview.

[5] "EasyEDA - Online PCB design & circuit simulator," [Online]. Available: https://easyeda.com/.

[6] "PCB Prototype & PCB Fabrication Manufacturer - JLCPCB," [Online]. Available: https://jlcpcb.com/.

[7] P. R. Kalata, "The Tracking Index: A Generalized Parameter for α-β and α-β-γ Target Trackers," *IEEE Transactions on Aerospace and Electronic Systems,* Vols. AES-20, no. 2, pp. 174 - 182, 1984.

[8] D. T. Chmielewski, *Application of Kalman Filters (Stochastic Observers)*, Philadelphia, 2018.

[9] T. Elenbaas, "Arduino-CmdMessenger," GitHub, 2017. [Online]. Available: https://github.com/thijse/Arduino-CmdMessenger.

[10] M. Harms, "PyCmdMessenger," GitHub, 2017. [Online]. Available: https://github.com/harmsm/PyCmdMessenger.

Appendix A Concept Sketch

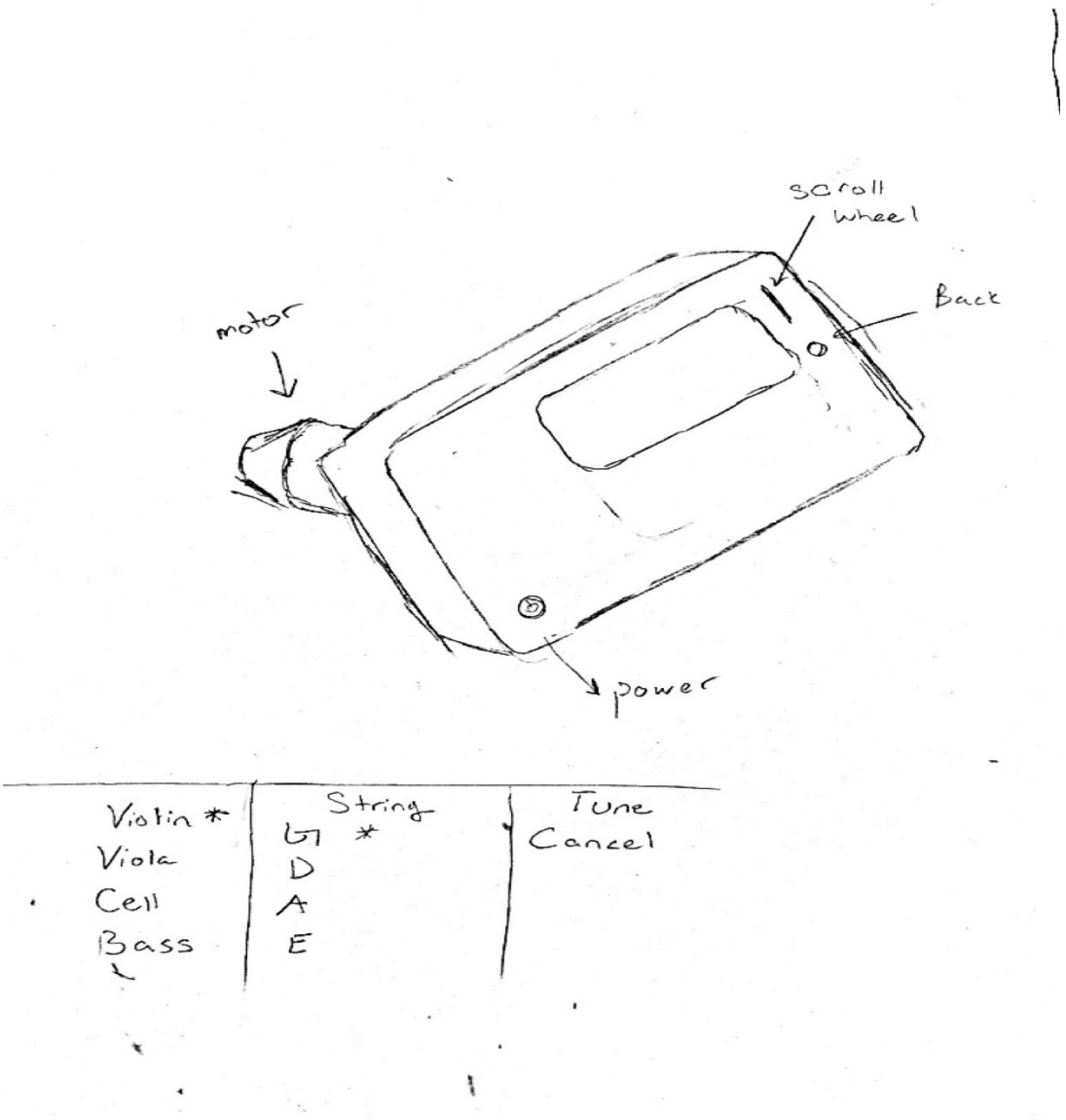


Figure 20: Sketch of User Interface and Final Project

Appendix B Teensy Firmware

```
#include <Audio.h>
#include <SPI.h>
#include <Wire.h>
#include <CmdMessenger.h>

//Hardware Serial//
//#define SERIALPORT Serial1

//USB Serial//
#define SERIALPORT Serial

CmdMessenger cmdMessenger = CmdMessenger(SERIALPORT);
enum TuningTargets{STRING_G3, STRING_D4, STRING_A4, STRING_E5};
enum StepSizes{WHOLE,HALF,QUARTER,EIGHTH};
enum SystemStates{
      NOT_READY,
      READY,
      TUNING,
      DONE
    }systemState;

#define kSerialSpeed 115200

float note;
float filtered_note;
float prob;

int Error = 0;

bool RUN = true;

void setup() {
  audioSetup();
  motorSetup();
  scheduleRoutines();
  attachCommandCallbacks();
  SERIALPORT.begin(kSerialSpeed);
  systemState = READY;
}

void loop() {
  cmdMessenger.feedinSerialData();
  performPolledRoutines();
}

//////////////////////////////////////////////////

#include <AccelStepper.h>

#define speedLimit 500

//#define enablePin 4
//#define MS1 22
```

```
//#define MS2 23

#define enablePin 39
#define MS1 38
#define MS2 37

//L298N//
//AccelStepper stepper(4, 2, 3, 4, 5);

//ULN2003//
//AccelStepper stepper(4, 2, 3, 5, 4);

//MP6500//
//AccelStepper stepper(1, 2, 3);
AccelStepper stepper(1, 36, 35);

float stepperSpeed;

void motorSetup(){
  stepper.setMaxSpeed(speedLimit);
  pinMode(enablePin, OUTPUT);
  pinMode(MS1, OUTPUT);
  pinMode(MS2, OUTPUT);
  setStepSize(QUARTER);
}

void runStepper(){
  stepper.setSpeed(stepperSpeed);
//  stepper.setSpeed(-Output);
  stepper.runSpeed();
}

void stopMotor(){
  noInterrupts();
  stepperSpeed = 0;
  interrupts();
  digitalWrite(enablePin, LOW);
}

void setStepSize(StepSizes stepSize){
  switch(stepSize){
    case WHOLE: {
      digitalWrite(MS1, LOW);
      digitalWrite(MS2, LOW);
      break;
    }
    case HALF: {
      digitalWrite(MS1, HIGH);
      digitalWrite(MS2, LOW);
      break;
    }
    case QUARTER: {
      digitalWrite(MS1, LOW);
      digitalWrite(MS2, HIGH);
      break;
    }
    case EIGHTH: {
```

```
      digitalWrite(MS1, HIGH);
      digitalWrite(MS2, HIGH);
      break;
    }
  }
}

////////////////////////////////////////////////

float R = 6.1E-3;  // Observation Noise Covariance
float Q = 7e-4;    // Process Noise Covariance
float Pc = 0.0;
float G = 0.0;     // Kalman Gain
float P = 1.0;
float Xp = 0.0;
float Zp = 0.0;
float Xe = 0.0;

float StepFilter(float measured_pitch){
  // Shift Calculated Predictions
  Xp = Xe;
  Zp = Xp;

  // Calculate PC covariance
  Pc = P + Q;

  // Calculate Gain
  G = Pc / (Pc + R);
  P = (1-G)* Pc;

  // Calculate Estimate
  Xe = G*(measured_pitch-Zp) + Xp;

  return Xe;
}

// function to print out floats in HEX
void serialFloatPrint(float f) {
  byte * b = (byte *) &f;
  SERIALPORT.print("#");
  for(int i=0; i<4; i++) {
    byte b1 = (b[i] >> 4) & 0x0f;
    byte b2 = (b[i] & 0x0f);
    char c1 = (b1 < 10) ? ('0' + b1) : 'A' + b1 - 10;
    char c2 = (b2 < 10) ? ('0' + b2) : 'A' + b2 - 10;
    SERIALPORT.print(c1);
    SERIALPORT.print(c2);
  }
}

////////////////////////////////////////////////

#include <PID_v1.h>
//#include "myEnums.h"
//#define NOTE_E5 659.3
//#define NOTE_A4 440.0
//#define NOTE_D4 293.7
```

```
//#define NOTE_G3 196.0

TuningTargets tuningTarget = STRING_D4;

double Setpoint, Input, Output;
double p_Kp=300, p_Ki=0, p_Kd=0;
PID tuningPID(&Input, &Output, &Setpoint, p_Kp, p_Ki, p_Kd, DIRECT);

float FREQ_TARGET; // Current setpoint pitch
float FREQ_BAND = 50; // Defines detection window. Target Pitch +/= FREQ_BAND
float focusBand = 3;
float validBand = 0.5;

int inRangeCounter = 0; // Counts the number of times the measured pitch is in
acceptable range
#define MaxConsecValid 20
int tuningFaultCounter = 0; // Increments if motor moves but pitch doesn't
change
#define MaxTuningFault 20

AudioInputAnalog              adc1;
AudioAnalyzeNoteFrequency     notefreq1;
AudioConnection               patchCord1(adc1,0, notefreq1,0);

long lastSampleTime = 0;
bool noteAvailable = false;
bool stream = false;

void setTuningTarget(TuningTargets target){
  switch(target){
    case STRING_G3: {
      FREQ_TARGET = 196.0;
      FREQ_BAND = 50;
      break;
    }
    case STRING_D4: {
      FREQ_TARGET = 293.7;
      FREQ_BAND = 50;
      break;
    }
    case STRING_A4: {
      FREQ_TARGET = 440.0;
      FREQ_BAND = 50;
      break;
    }
    case STRING_E5: {
      FREQ_TARGET = 659.3;
      FREQ_BAND = 80;
      break;
    }
  }
  Setpoint = FREQ_TARGET;
}

void audioSetup(){
  AudioMemory(30);
  notefreq1.begin(.15);
```

```
  tuningPID.SetMode(AUTOMATIC);
  tuningPID.SetOutputLimits(-speedLimit,speedLimit);
  setTuningTarget(STRING_E5);
}
void detectPitch(){
  if (notefreq1.available()) {
    noteAvailable = true;

    lastSampleTime = millis();
    note = notefreq1.read();


    prob = notefreq1.probability();


  }else{
    noteAvailable = false;
  }
}

void tuneString(){
  if (noteAvailable){
    if (note>FREQ_TARGET-FREQ_BAND && note <FREQ_TARGET+FREQ_BAND){
      filtered_note = StepFilter(note);
      if (stream){
        SERIALPORT.printf("%3.3f", note);
        SERIALPORT.print(" ");
        SERIALPORT.printf("%3.3f\n", filtered_note);
      }
      Input = filtered_note; // Pass measured value to PID

      // Check Process Status
      if                (filtered_note>FREQ_TARGET-validBand                &&
filtered_note<FREQ_TARGET+validBand){
          inRangeCounter = inRangeCounter + 1;
      }else{
          inRangeCounter = 0;
      }
      noInterrupts();
      stepperSpeed      =      map(filtered_note,      FREQ_TARGET-focusBand,
FREQ_TARGET+focusBand, -speedLimit, speedLimit);
//      stepperSpeed = -Output; // Use PID Controller
//      Serial.printf("%3.3f\n", -Output);
      interrupts();
    }
  }
  if ((millis()-lastSampleTime) > 100){
    noInterrupts();
    stepperSpeed = 0;
    interrupts();
  }
}

void runStateMachine(){
  switch(systemState){
    case NOT_READY: {
      break;
```

```
    }
    case READY:{
      break;
    }
    case TUNING:{
      tuneString();
      if (inRangeCounter > MaxConsecValid){
        systemState = DONE;
        inRangeCounter = 0;
        stopMotor();
//        SERIALPORT.println("Done");
//        cmdMessenger.sendCmd(kAcknowledge, "Done");
      }
      break;
    }
    case DONE:{
      break;
    }
  }
}

////////////////////////////////////////////////////

#include <Metro.h>
#define MOTOR_INTERVAL 10 //Defined in Microseconds

IntervalTimer myTimer;
Metro ledMetro = Metro(500);

void scheduleRoutines(){
  myTimer.begin(runStepper, 10);
  pinMode(13,OUTPUT);
  //myTimer.priority(1);
}

void performPolledRoutines(){
  tuningPID.Compute();
  detectPitch();
  runStateMachine();

  //blink LED
  if (ledMetro.check() == 1){
    digitalWrite(13, !digitalRead(13));
  }
}

////////////////////////////////////////////////////

enum
{
  kGetStatus,          // 0
  kGetStatusAck,       // 1
  kGetPitchData,       // 2
  kGetPitchDataAck,    // 3
  kSetTarget,          // 4
  kSetTargetAck,       // 5
  kBeginTuning,        // 6
```

```
  kBeginTuningAck,    // 7
  kStopTuning,        // 8
  kStopTuningAck,     // 9
  kCalibrate,         // 10
  kCalibrateAck,      // 11
  kToggleRawStream,    // 12
  kToggleRawStreamAck // 13
};

void sendHeader(){
  cmdMessenger.sendCmdBinArg<int>(systemState);
  cmdMessenger.sendCmdBinArg(Error);
}
void attachCommandCallbacks()
{
  cmdMessenger.attach(kGetStatus, OnGetStatus);
  cmdMessenger.attach(kGetPitchData, OnGetPitchData);
  cmdMessenger.attach(kSetTarget, OnSetTarget2);
  cmdMessenger.attach(kBeginTuning, OnBeginTuning);
  cmdMessenger.attach(kStopTuning, OnStopTuning);
  cmdMessenger.attach(kCalibrate, OnCalibrate);
  cmdMessenger.attach(kToggleRawStream, toggleRawStream);
  cmdMessenger.printLfCr();
}

void OnGetStatus(){
  // Returns the System State and Error
  cmdMessenger.sendCmdStart(kGetStatusAck);
  sendHeader();
  cmdMessenger.sendCmdEnd();
}

void OnGetPitchData(){
  // Returns the setpoint pitch and the measured pitch
  cmdMessenger.sendCmdStart(kGetPitchDataAck);
  sendHeader();
  cmdMessenger.sendCmdBinArg<float>(FREQ_TARGET);
  cmdMessenger.sendCmdBinArg<float>(filtered_note);
  cmdMessenger.sendCmdEnd();
}


//void OnGetPitchData(){
//  // GARUNTEED DONE TUNING AFTER 1 CYCLE
//  cmdMessenger.sendCmdStart(kGetPitchDataAck);
//  cmdMessenger.sendCmdBinArg<int>(systemState);
//  cmdMessenger.sendCmdBinArg(Error);
//  cmdMessenger.sendCmdBinArg<float>(FREQ_TARGET);
//  cmdMessenger.sendCmdBinArg<float>(filtered_note);
//  cmdMessenger.sendCmdEnd();
//  systemState = DONE;
//}


void toggleRawStream(){
  // Toggles telemetry stream on and off
  stream = !stream;
```

```
  if (stream){
    cmdMessenger.sendCmdStart(kToggleRawStreamAck);
    sendHeader();
    cmdMessenger.sendCmdArg("Streaming On");
    cmdMessenger.sendCmdEnd();
  }else{
    cmdMessenger.sendCmdStart(kToggleRawStreamAck);
    sendHeader();
    cmdMessenger.sendCmdArg("Streaming Off");
    cmdMessenger.sendCmdEnd();
  }
}

void OnBeginTuning(){
  // Activate motor and start live tuning process
  systemState = TUNING;
  digitalWrite(enablePin, HIGH);
  cmdMessenger.sendCmdStart(kBeginTuningAck);
  sendHeader();
  cmdMessenger.sendCmdArg("Tuning Start");
  cmdMessenger.sendCmdEnd();
}

void OnStopTuning(){
  // Stops the tuning process. Halts motor and returns system to READY state
  cmdMessenger.sendCmdStart(kStopTuningAck);
  sendHeader();
  cmdMessenger.sendCmdArg("Stopped Tuning");
  cmdMessenger.sendCmdEnd();
  systemState = READY;
  stopMotor();
}

void OnCalibrate(){
  // Calibrate Stuff
}

void OnSetTarget()
  // Set the tuning profile via the letter of the note
{
  String target = cmdMessenger.readStringArg();

  if (target == "E"){
    cmdMessenger.sendCmdStart(kSetTargetAck);
    sendHeader();
    cmdMessenger.sendCmdArg("Target: E String");
    cmdMessenger.sendCmdEnd();
    setTuningTarget(STRING_E5);
  }
  else if(target == "A"){
    cmdMessenger.sendCmdStart(kSetTargetAck);
    sendHeader();
    cmdMessenger.sendCmdArg("Target: A String");
    cmdMessenger.sendCmdEnd();
    setTuningTarget(STRING_A4);
  }
  else if(target == "D"){
```

```
    cmdMessenger.sendCmdStart(kSetTargetAck);
    sendHeader();
    cmdMessenger.sendCmdArg("Target: D String");
    cmdMessenger.sendCmdEnd();
    setTuningTarget(STRING_D4);
  }
  else if(target == "G"){
    cmdMessenger.sendCmdStart(kSetTargetAck);
    sendHeader();
    cmdMessenger.sendCmdArg("Target: G String");
    cmdMessenger.sendCmdEnd();
    setTuningTarget(STRING_G3);
  }else{
    cmdMessenger.sendCmdStart(kSetTargetAck);
    sendHeader();
    cmdMessenger.sendCmdArg("Unknown String");
    cmdMessenger.sendCmdEnd();
  }
}

void OnSetTarget2(){
  // Set the tuning profile via explicit pitch and band values
  FREQ_TARGET = cmdMessenger.readBinArg<float>();
  FREQ_BAND = cmdMessenger.readBinArg<float>();
  Setpoint = FREQ_TARGET;

  cmdMessenger.sendCmdStart(kSetTargetAck);
  sendHeader();
  cmdMessenger.sendCmdBinArg<float>(FREQ_TARGET);
  cmdMessenger.sendCmdBinArg<float>(FREQ_BAND);
  cmdMessenger.sendCmdEnd();
}
```

Appendix C Raspberry Pi Software

```
# -*- coding: utf-8- -*-
from subprocess import Popen, PIPE
from time import sleep
from datetime import datetime
import board
import digitalio
import adafruit_character_lcd.character_lcd as characterlcd
import threading
import PyCmdMessenger
import queue
import RPi.GPIO as GPIO

lcd_columns = 16
lcd_rows = 2

#lcd_rs = digitalio.DigitalInOut(board.D22)
#lcd_en = digitalio.DigitalInOut(board.D17)
#lcd_d4 = digitalio.DigitalInOut(board.D25)
#lcd_d5 = digitalio.DigitalInOut(board.D24)
#lcd_d6 = digitalio.DigitalInOut(board.D23)
#lcd_d7 = digitalio.DigitalInOut(board.D18)

lcd_rs = digitalio.DigitalInOut(board.D20)
lcd_en = digitalio.DigitalInOut(board.D16)
lcd_d4 = digitalio.DigitalInOut(board.D7)
lcd_d5 = digitalio.DigitalInOut(board.D8)
lcd_d6 = digitalio.DigitalInOut(board.D25)
lcd_d7 = digitalio.DigitalInOut(board.D24)


lcd = characterlcd.Character_LCD_Mono(lcd_rs, lcd_en, lcd_d4, lcd_d5, lcd_d6,
lcd_d7, lcd_columns, lcd_rows)

lcd.clear()

lcd_line_1 = "Initializing "
lcd_line_2 = "Program"

lcd.message = lcd_line_1 + "\n" + lcd_line_2

GPIO.setmode(GPIO.BCM)
#GPIO.setup(4, GPIO.IN, pull_up_down=GPIO.PUD_UP)
#GPIO.setup(5, GPIO.IN, pull_up_down=GPIO.PUD_UP)
#GPIO.setup(6, GPIO.IN, pull_up_down=GPIO.PUD_UP)
#GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)

GPIO.setup(19, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(13, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(6, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(5, GPIO.IN, pull_up_down=GPIO.PUD_UP)

Up_E = threading.Event()
Down_E = threading.Event()
```

```
Select_E = threading.Event()
Back_E = threading.Event()

Sync = threading.Event()
Tune = threading.Event()
Stahp = threading.Event()
Done = threading.Event()

#arduino                                                                    =
PyCmdMessenger.ArduinoBoard("/dev/ttyAMA0",baud_rate=9600,int_bytes=4)
arduino                                                                     =
PyCmdMessenger.ArduinoBoard("/dev/ttyACM0",baud_rate=115200,int_bytes=4)

commands = [["GetStat",""],
            ["RcvStat","ii"],
            ["GetPitch",""],
            ["RcvPitch","iiff"],
            ["SetTarget","ff"],
            ["AckTarget","iiff"],
            ["InitTune",""],
            ["AckTune","iis"],
            ["StopTune",""],
            ["AckStop","iis"],
            ["Calibrate","ii"],
            ["AckCal","iiii"],
            ["Stream",""],
            ["AckStream","iis"]]

c = PyCmdMessenger.CmdMessenger(arduino, commands)

class Update(threading.Thread):
    def __init__(self, Sync, Tune, Freq, Params, Back, Stahp, Done):
        threading.Thread.__init__(self)

        self.tune = Tune
        self.sync = Sync
        self.freq = Freq
        self.Params = Params
        self.params = []
        self.back = Back
        self.Stahp = Stahp
        self.Done = Done
        self.msg = []


    def run(self):
        #print("sending")
        c.send("GetStat")
        while True:
            self.msg = c.receive()

            if self.tune.is_set():
                if self.Stahp.is_set():
                    self.tune.clear()
                    self.Stahp.clear()
                    c.send("StopTune")
                    #print("sent StopTune Cancelled")
```

```
                else:
                    if ((self.msg[1])[0]) == 1:
                        #print("tune raised")
                        c.send("InitTune")
                        #print("sent InitTune")
                    elif ((self.msg[1])[0]) == 2:
                        if self.msg[0] == "RcvPitch":
                            if not self.freq.qsize():
                                self.freq.put(((self.msg[1])[3]))
                            #print((self.msg[1])[2])
                        c.send("GetPitch")
                        #print("sent GetPitch")
                    elif ((self.msg[1])[0]) == 3:
                        self.tune.clear()
                        self.Done.set()
                        #print("done")
                        c.send("StopTune")
                        #print("sent StopTune Done")

            elif self.sync.is_set():
                if self.msg[0] == "AckTarget":
                    self.sync.clear()
                    self.tune.set()
                    c.send("GetStat")
                    #print("sent GetStat")
                else:
                    #print("sync raised")
                    self.params = self.Params.get()
                    #print(self.params)
                    #print("sent  SetTarget  " + str(self.params[0]) + ",  " +
str(self.params[1]))
                    c.send("SetTarget", self.params[0], self.params[1])
#                    print("sent SetTarget " + self.params)

            else:
                c.send("GetStat")
                #print("sent GetStat")

class LCD(threading.Thread):
    def __init__(self, Freq, Params, Sync, Up, Down, Select, Back, Stahp, Done):
        threading.Thread.__init__(self)

        self.Sync = Sync
        self.Up = Up
        self.Down = Down
        self.Select = Select
        self.Back = Back
        self.Stahp = Stahp
        self.Done = Done

        self.MenuNum = 0
        self.Menu1IDX = 0
        self.Menu2IDX = 0
        self.idx = 0
        self.idx2 = 0
        self.Menu =  [("Violin", [("E", 659.3, 80.0), ("A", 440.0, 50.0), ("D",
293.7, 50.0), ("G", 196.0, 50.0)]),
```

```
                    ("Viola", [("A", 440.0, 50.0), ("D", 293.7, 50.0), ("G",
196.0, 50.0), ("C", 130.8, 50.0)]),
                    ("Cello", [("A", 220.0, 50.0), ("D", 146.8, 50.0), ("G",
98.0, 50.0), ("C", 65.41, 50.0)]),
                    ("Upright Bass", [("E", 41.2, 50.0), ("A", 55.0, 50.0),
("D", 73.4, 50.0), ("G", 98.0, 50.0)])]
        self.UpdateScreen = True
        self.Target = 440.0
        self.Tolerance = 0.1
        self.freq = Freq
        self.Params = Params

    def run(self):
        lcd_line_1 = "Starting"
        lcd_line_2 = "Threads"
        lcd.message = lcd_line_1.ljust(16)  + "\n" + lcd_line_2.ljust(16)
#         sleep(1)
        while True:
            if self.MenuNum == 2:
                while (not self.Back.is_set()) and (not self.Done.is_set()):
                    if not self.freq.empty():
                        lcd_line_1 = "Tuning: " + str(self.Target)
                        lcd_line_2 = "{:.2f} Hz".format(self.freq.get())
                        lcd.message   =   lcd_line_1.ljust(16)     +   "\n"   +
lcd_line_2.ljust(16)
                        #print("Update")
                if self.Back.is_set():
                    self.Stahp.set()
                    self.Back.clear()
                    lcd_line_1 = "Cancelled"
                    lcd_line_2 = ""
                    lcd.message    =    lcd_line_1.ljust(16)       +    "\n"    +
lcd_line_2.ljust(16)
                    sleep(1)
                else:
                    self.Done.clear()
                    lcd_line_1 = "Successfully"
                    lcd_line_2 = "Tuned :)"
                    lcd.message    =    lcd_line_1.ljust(16)       +    "\n"    +
lcd_line_2.ljust(16)
                    sleep(1)

                self.MenuNum = 1
                self.UpdateScreen = True

            else:
                if self.Up.is_set():
                    if self.MenuNum == 0:
                        self.Menu1IDX -= 1
                    else:
                        self.Menu2IDX -= 1
                    self.UpdateScreen = True
                    self.Up.clear()

                if self.Down.is_set():
                    if self.MenuNum == 0:
                        self.Menu1IDX += 1
```

```
                    else:
                        self.Menu2IDX += 1
                    self.UpdateScreen = True
                    self.Down.clear()

                if self.Select.is_set():
                    if self.MenuNum == 0:
                        self.MenuNum = 1
                    else:
                        self.MenuNum = 2
                    self.UpdateScreen = True
                    self.Select.clear()

                if self.Back.is_set():
                    if self.MenuNum == 1:
                        self.MenuNum = 0
                    self.UpdateScreen = True
                    self.Back.clear()

                if self.UpdateScreen:
                    if self.MenuNum == 0:
                        lcd_line_1      =      (self.Menu[(self.Menu1IDX      %
len(self.Menu))])[0] + "*"
                        lcd_line_2   =   (self.Menu[((self.Menu1IDX   +   1)   %
len(self.Menu))])[0]

                    elif self.MenuNum == 1:
                        self.idx = (self.Menu1IDX % len(self.Menu))
                        lcd_line_1                                              =
(((self.Menu[self.idx]))[1][(self.Menu2IDX                                      %
len((self.Menu[self.idx])[1]))])[0] + "*"
                        lcd_line_2                                              =
(((self.Menu[self.idx]))[1][((self.Menu2IDX          +          1)          %
len((self.Menu[self.idx])[1]))])[0]
#                        print(self.Menu[self.idx])

                    else:
                        self.idx2          =          (self.Menu2IDX          %
len((self.Menu[self.idx])[1]))
                        self.Target                                             =
(((self.Menu[self.idx]))[1][self.idx2])[1]
                        self.Range                                              =
(((self.Menu[self.idx]))[1][self.idx2])[2]
#                        self.Range = 80.0
                        self.Params.put([self.Target, self.Range])
                        lcd_line_1 = "Tuning..."
                        lcd_line_2 = "Target: " + str(self.Target)
                        self.Sync.set()

#                    print(self.idx)
#                    print(self.idx2)
                    #print(lcd_line_1)
                    #print(self.MenuNum)
                    lcd.message    =    lcd_line_1.ljust(16)    +    "\n"    +
lcd_line_2.ljust(16)
                    self.UpdateScreen = False
```

```
def Up(channel):
    Up_E.set()
    print("UP SET")

def Down(channel):
    Down_E.set()
    print("DOWN SET")

def Select(channel):
    Select_E.set()
    print("SELECT SET")

def Back(channel):
    Back_E.set()
    print("BACK SET")


def main():

    Freq = queue.Queue()
    Params = queue.Queue()

    GPIO.add_event_detect(13, GPIO.FALLING, callback=Up, bouncetime=300)
    GPIO.add_event_detect(19, GPIO.FALLING, callback=Down, bouncetime=300)
    GPIO.add_event_detect(6, GPIO.FALLING, callback=Select, bouncetime=300)
    GPIO.add_event_detect(5, GPIO.FALLING, callback=Back, bouncetime=300)

    Operate = Update(Sync, Tune, Freq, Params, Back, Stahp, Done)
    GUI = LCD(Freq, Params, Sync, Up_E, Down_E, Select_E, Back_E, Stahp, Done)
    Operate.start()
    GUI.start()
    Operate.join()
    GUI.join()

if __name__ == '__main__':
    main()
```

Appendix D Enclosure Drawings

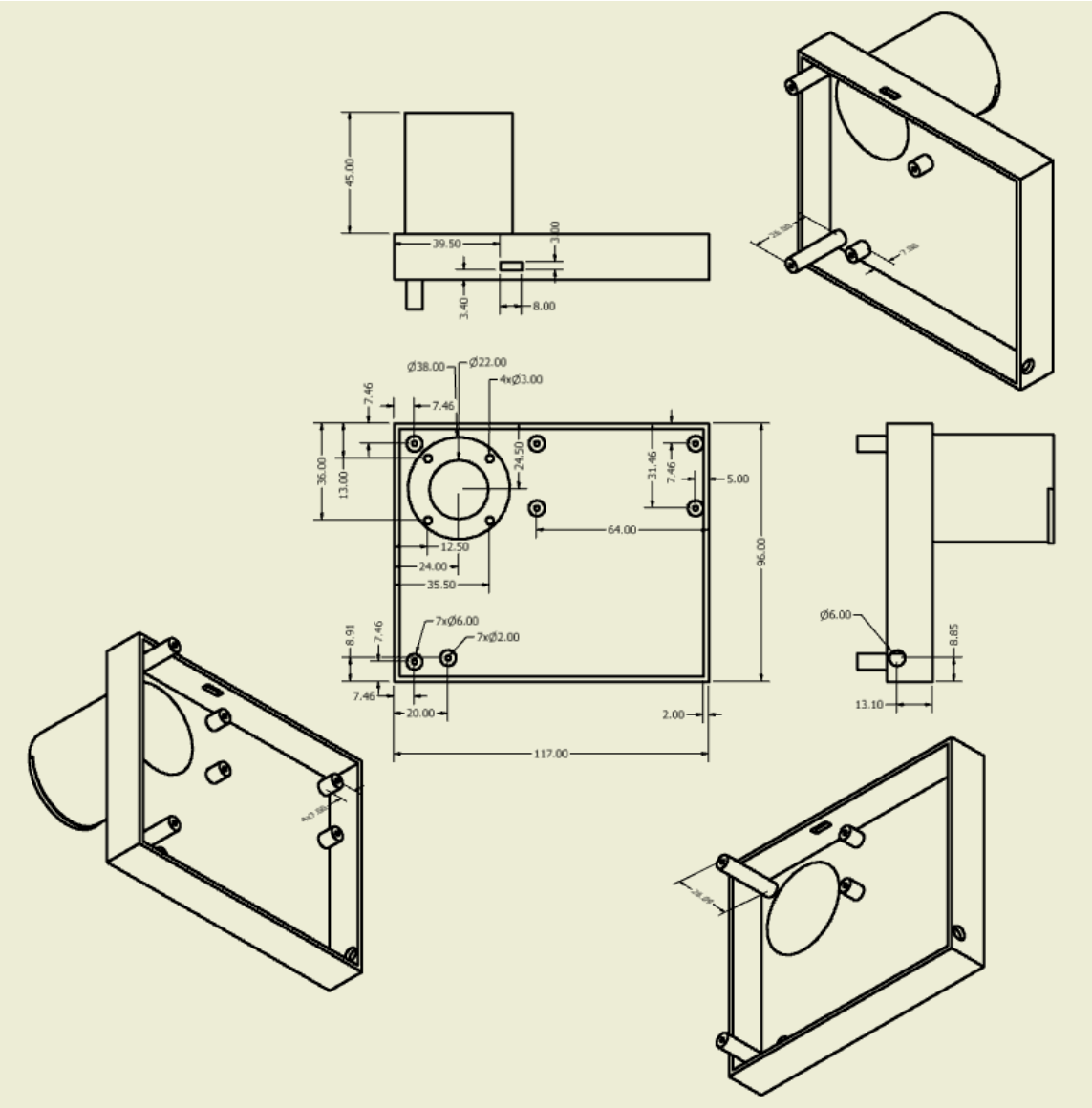


Figure 21 Enclosure Bottom Drawing

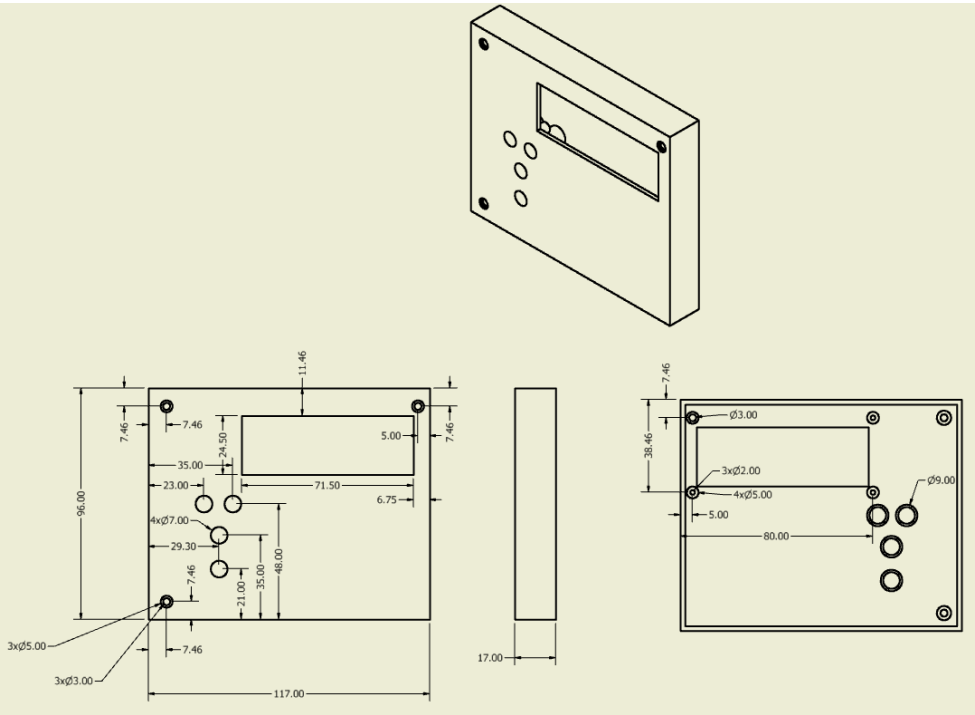


Figure 22 Enclosure Top Drawing

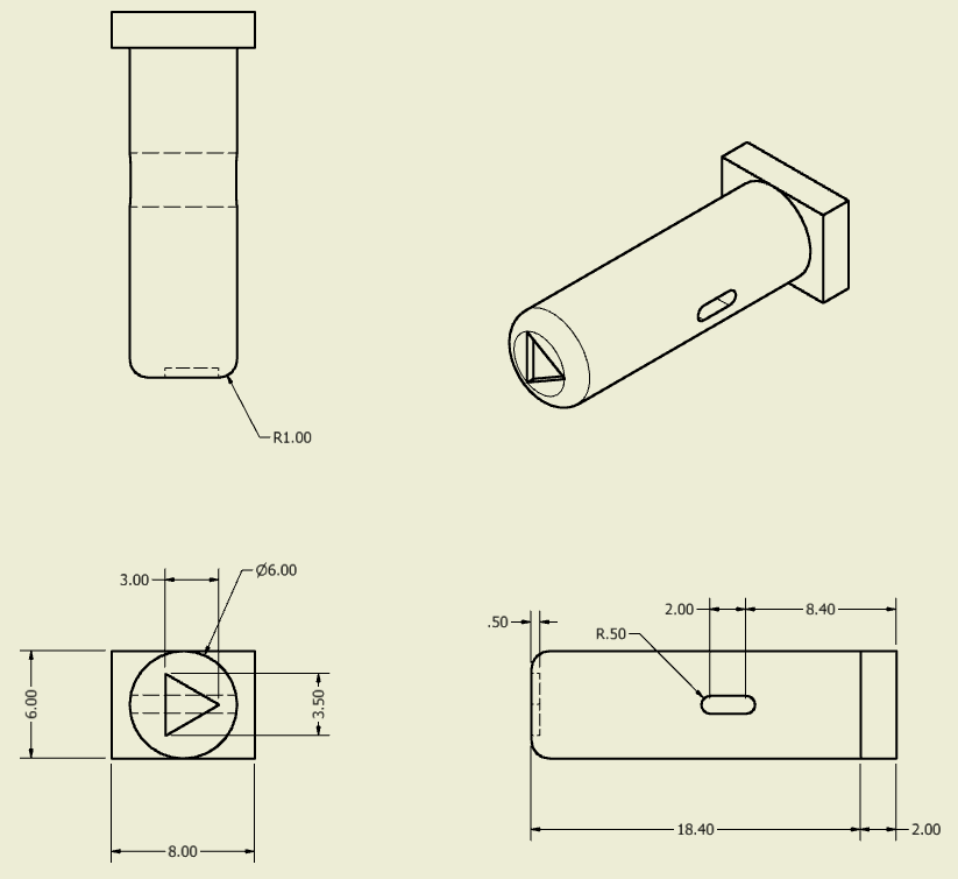


Figure 23 Button Up/Down Drawing

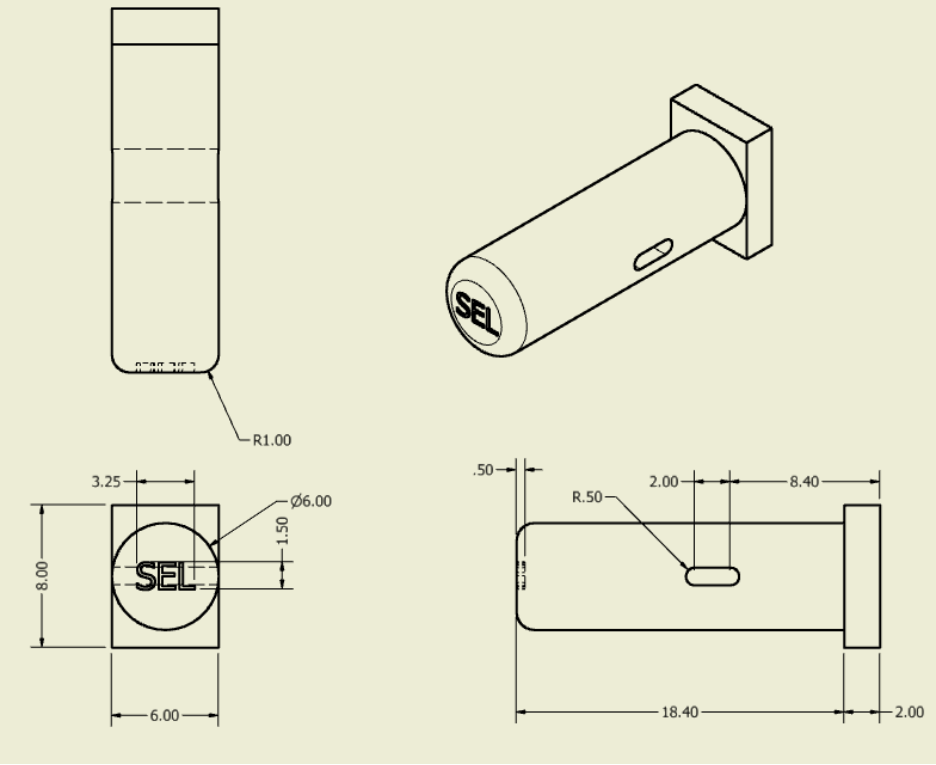


Figure 24 Button Select Drawing

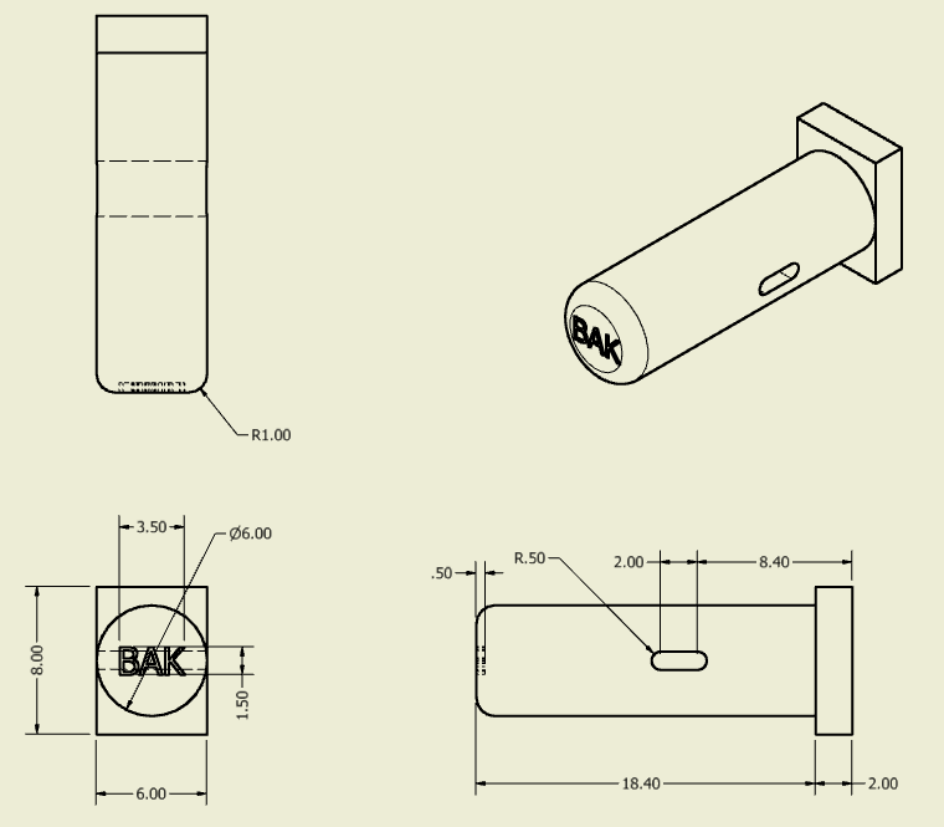


Figure 25 Button Back Drawing