111a Final Project

Professor Fajans

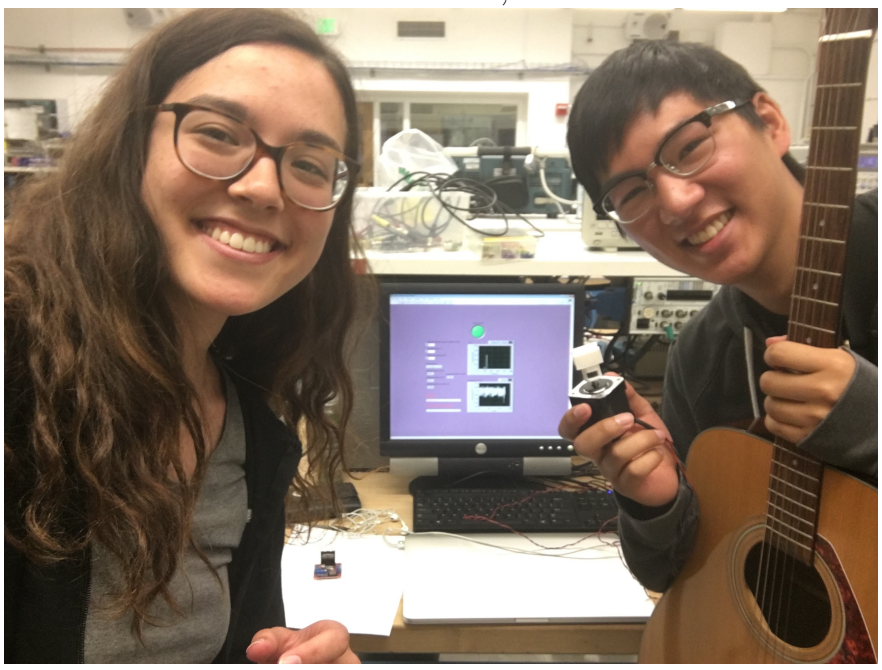
University of California, Berkeley

Automatic Guitar Tuner

Author:
Sabrina Berger

Lab Partner:
Jayke Nguyen

November 22, 2017



1 Introduction

The idea for this project arose when I wanted a faster, more mindless way to tune my guitar. Having to tune a guitar by hand can waste precious time needed in practicing the instrument.

My partner and I constructed an automatic guitar tuner that could be attached to any tuning peg on a guitar to tune the string autonomously while it is played. When the string was tuned, we indicated this with an LED lighting up and the tuner stopping adjustments to the peg. The string to be tuned was specified in a visual interface using LabVIEW (see appendix). See figure 1 for an overview of the entire tuner and where to find information about each of the components within this write up. This block diagram describes how all the components of the project described in the sections that follow connect to one another.

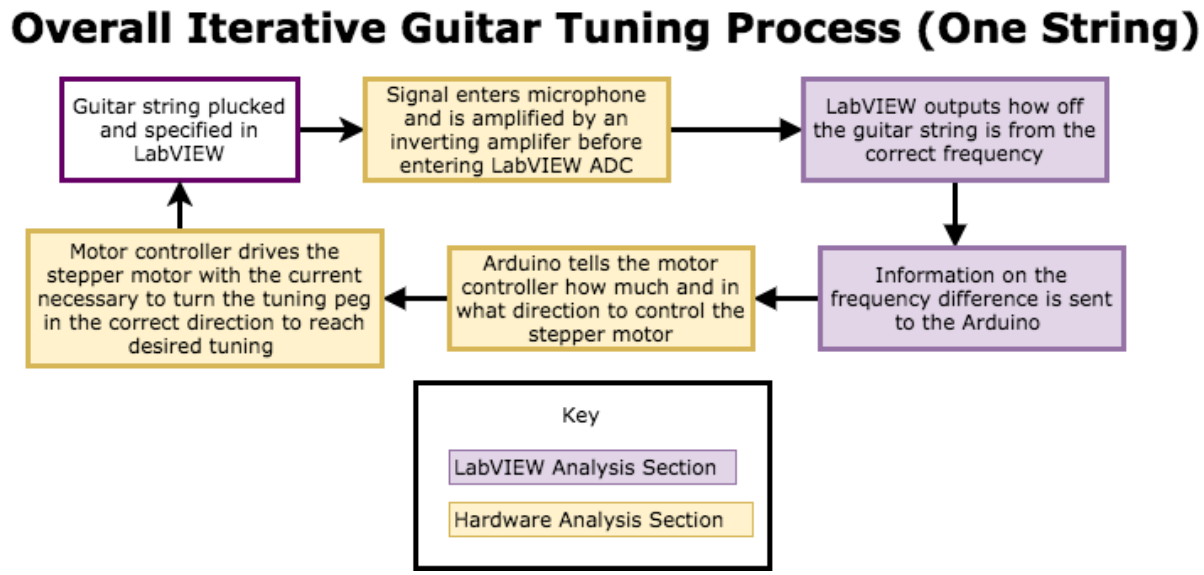


Figure 1: Overview

Most of the hardware was in the Arduino and motor. We used an Arduino Uno to control our motor. The Arduino Uno is a microcontroller board which 14 digital input and output pins, and 6 analog inputs. We employed 1 digital and analog input and 4 analog outputs on the Arduino. We had to write code in the Arduino language to correctly use the stepper motor. Luckily, we both had experience in C/C++ which the Arduino language is based off of. The Arduino language also had a built in class for controlling stepper motors that we used as a block box. We purchased a stepper motor and motor controller on Amazon to turn the tuning pegs of our guitar.

2 LabVIEW Design

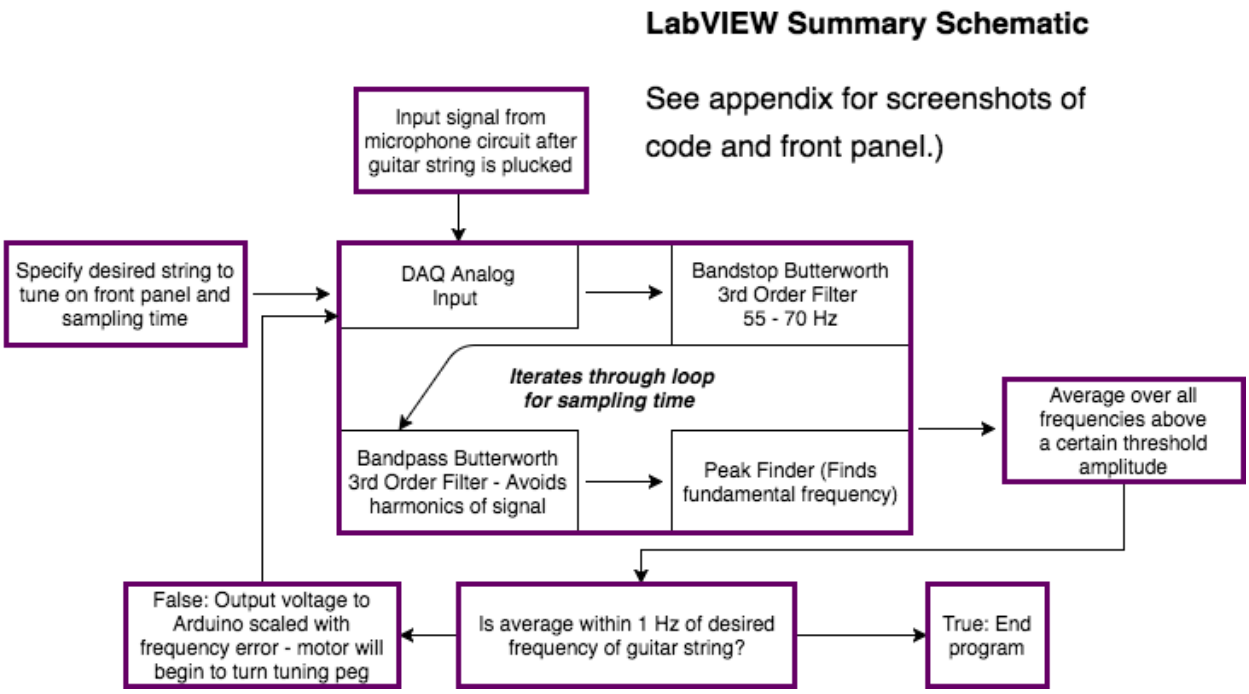


Figure 2: Summary of LabVIEW code

The majority of the signal processing for the signal from the guitar string is done within LabVIEW. See figure 2 for an overview of the LabVIEW code and see the appendix for images of the front panel and the code itself. After the guitar string was plucked, the sound is turned into the input signal via a small microphone placed near to the string. The signal is then amplified so that it can be better picked up the DAQ. The signal is converted to digital using the ADC on the DAQ.

The user selects the string to tune, the highest possible deviation from desired frequency that a string can have, the sampling time, and the minimum amplitude of the signal that's used in averaging. We found the following parameters worked the best for our tuner:

Highest Possible Frequency Difference (Hz)	100
Sampling Time (s)	7
Minimum Amplitude (V)	0.05

If the current frequency of the guitar string is outside of the possible frequency range the program allows, an alert will show that this is happening and inform the user to specify a larger range.

The signal then enters a while loop that continues collecting signals from the guitar until the sampling time is reached.

To prevent the usual power line hum of 60 Hz from entering out signal, we started out by passing the signal through a digital bandstop filter. This bandstop filter blocked 55-70 Hz frequencies. We then passed the signal through a bandpass filter to prevent harmonics from overpowering the fundamental frequency of the signal. We passed the signal through a filter that only passed signals greater than $f_{fundamental}/1.5$ and less than $1.5f_{fundamental}$. Both of these filters were third order Butterworth filters. The Butterworth filter was chosen because of the flat frequency response, frequencies passing through stay at around the same amplitude. The Butterworth filter has a slower roll-off, and for this reason, we chose a higher order filter to make the roll-off faster. We chose a third order filter to maintain efficiency of the filter. Higher order filters than this slowed down the signal processing significantly.

Next, the signal passes through a peak finder. This should find the fundamental frequency of the signal. It uses a Fast Fourier Transform (FFT) to transform the signal into frequency space and takes the highest amplitude signal in this space as the fundamental frequency. After this, the loop is restarted and another voltage signal is collected from the DAQ.

Once the sampling time is reached, the fundamental frequencies of all the signals collected in an array during the while loop are averaged. The fundamental frequency is only added to the average if the amplitude is above the threshold frequency. This is to prevent noise and other sounds around the microphone from being added to the average. We chose to average multiple fundamental frequencies from the signal to prevent such signals from incorrectly determining the difference between the target (string frequency) and the current frequency of the signal. If the average fundamental frequency of the signal was within 1 Hz of the target frequency, the program stopped. Otherwise, the program remains continuously running until this condition is met and the string is tuned.

One last detail (will be discussed more in the Arduino section of this write up) was that we also output a digital signal to the Arduino that would tell the microcontroller when to start collecting the analog voltage to determine the frequency difference.

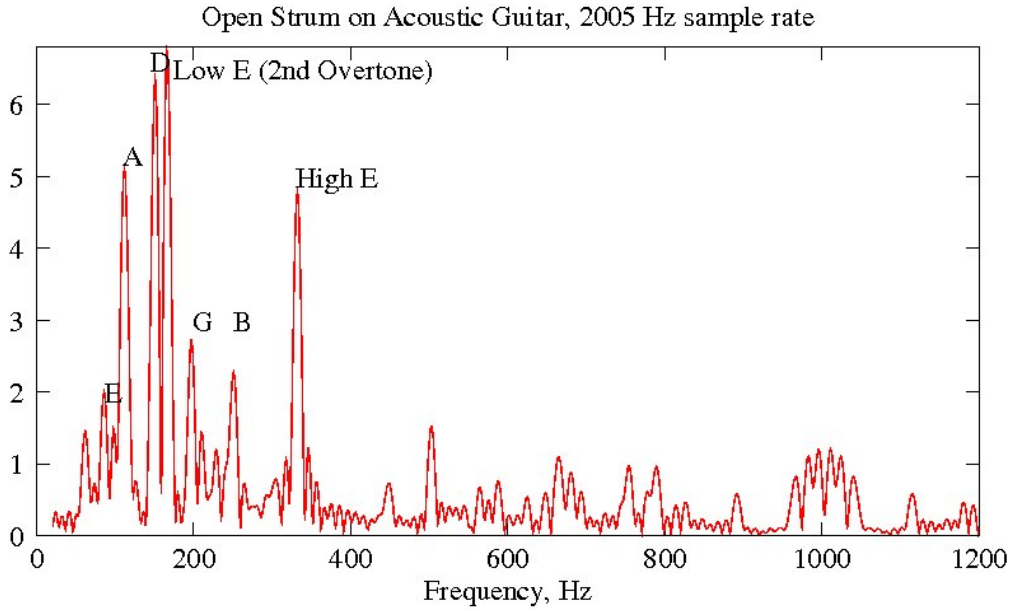


Figure 3: FFT of signal from all guitar strings showing fundamental frequency and harmonics

3 Hardware Design

3.1 Microphone Inverting Amplifier Circuit

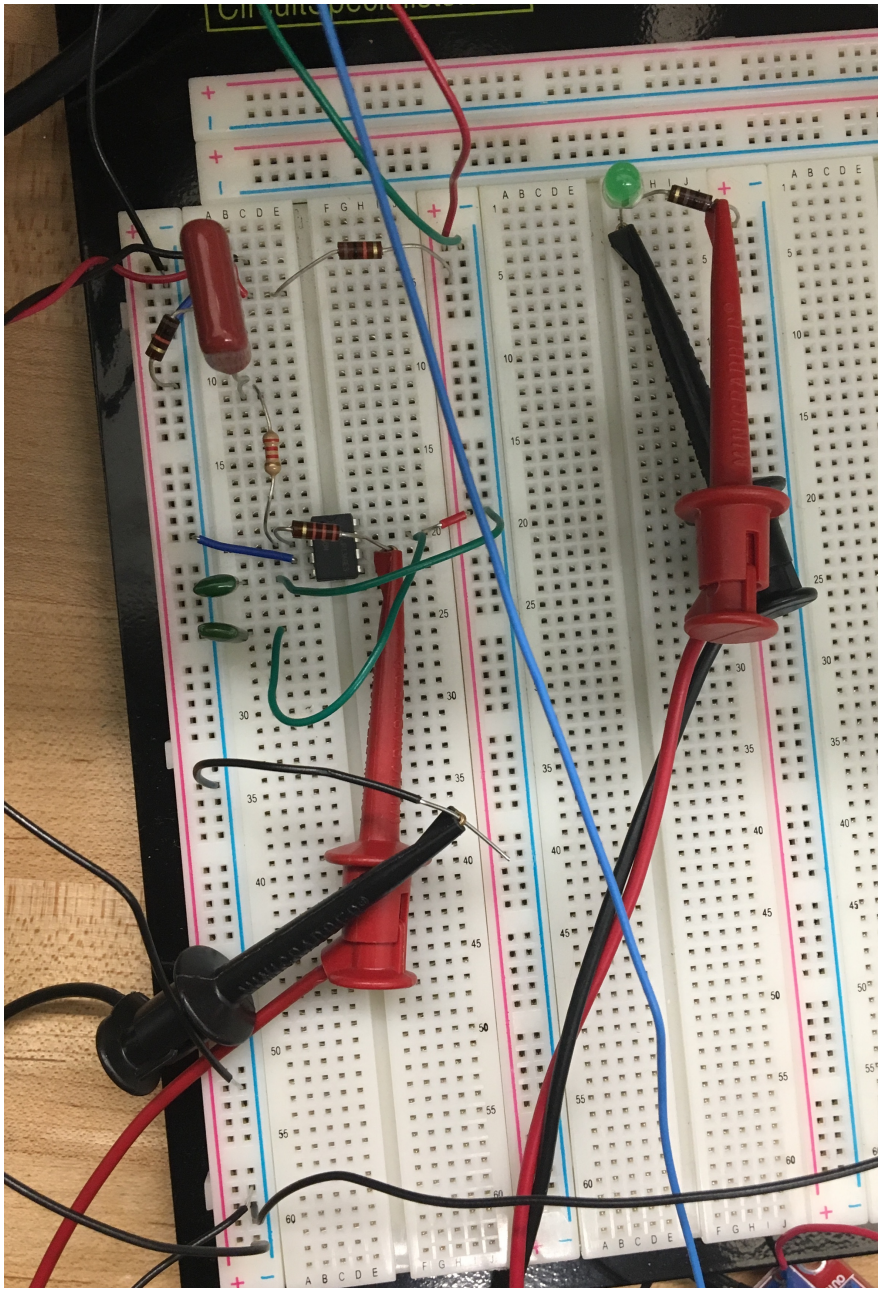


Figure 4: Actual microphone inverting amplifier circuit (op amp decoupling capacitors and microphone not pictured as we extended leads so we could hold it closer to the guitar strings)

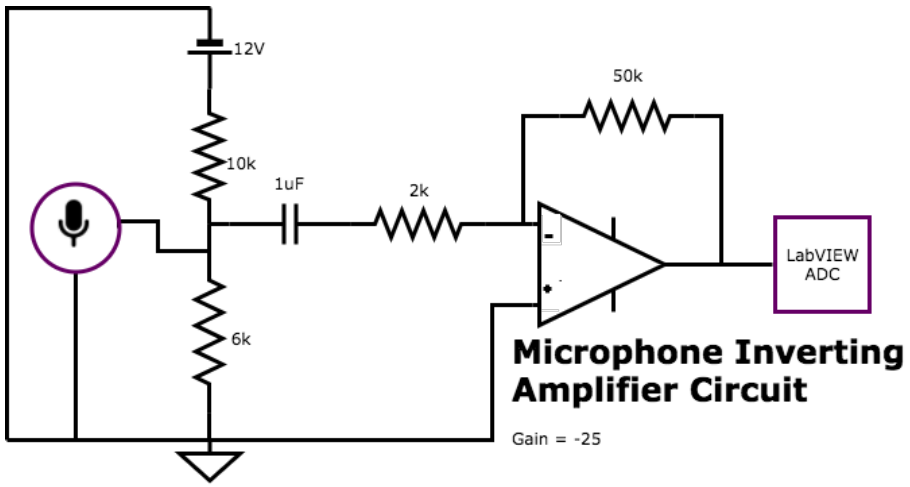


Figure 5: Microphone inverting amplifier circuit diagram

In order for the DAQ to pick up a decently high amplitude signal, we had to build a microphone amplifying circuit as seen in figure 4 and figure 5. This is because the microphone generates very low amplitude signals, around a mV or even less.

We calculate the closed loop voltage gain of the inverting amplifier as follows:

$$G = \frac{-R_2}{R_1}$$
$$G = \frac{-50k}{2k}$$
$$G = -25$$

The reason we have the 1 μF capacitor in between the voltage divider and amplifier is to reduce the DC offset of the signal from the microphone. We know the impedance of a capacitor decreases with increasing frequency as follows: $X_C = \frac{1}{j2\pi fC}$. We only want AC voltage flowing into our LabVIEW ADC so we have this capacitor in place for that reason.

We didn't want the full 12V to be used in powering the microphone so we used a voltage divider to reduce the microphone driving voltage.

$$V_{out} = V_{in}\frac{R_2}{R_1 + R_2}$$
$$V_{out} = 12V\frac{6k}{16k}$$
$$V_{out} = 4.5V$$

We powered the microphone with 4.5V.

3.2 Arduino and Motor Controller

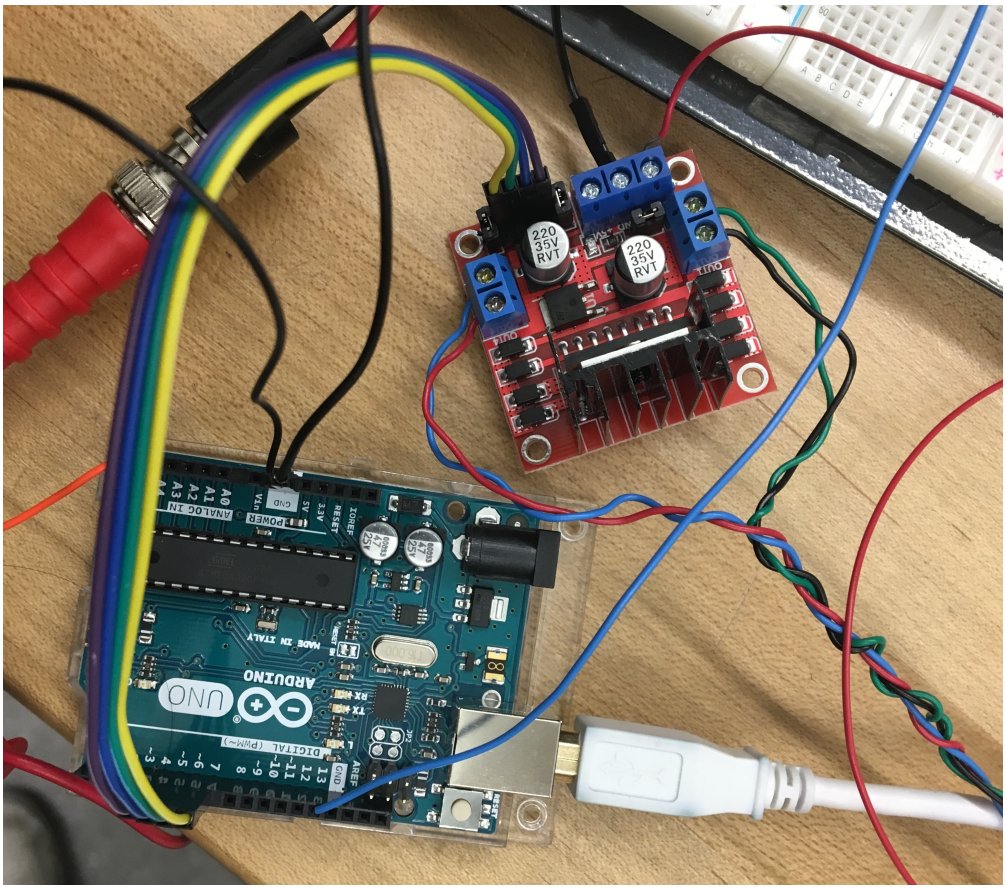


Figure 6: The Arduino and motor controller used in tuner

We employed an Arduino and motor controller in tandem to drive the stepper motor which turned the tuning pegs of the guitar. We powered the Arduino with 5V from the USB port (white cable) on our laptops. The Arduino read an analog voltage, the output from the LabVIEW code, between 0 and 5V. The ADC on the Arduino converted the analog voltage initially to a digital signal that could be interpreted by the Arduino for processing. The way the analog signal is converted into the number of steps that the stepper motor needs to turn is chronologically described below. For the actual Arduino code used, see the appendix.

1. Analog to Digital Voltage to Frequency - The LabVIEW code scaled the frequency difference (target frequency - current string frequency) between 0 and 5V. With 0 V being the string is too low in frequency by negative the highest possible frequency difference and 5V being the string is too high by the highest possible frequency difference. We'll call the highest possible frequency difference the threshold. 2.5V (right in the middle of 0 and 5V) means the string is in tune. This analog voltage is converted into bits and has to now be reverse scaled to frequency. The Arduino Uno has 6 analog inputs which each have 10 bits resolution. There are 1024 (0 corresponding to 0V and 1024 corresponding to 5V) possible values for the digital conversion on the Arduino. The digital signal is converted to frequency difference with the following relationship:

$$frequency = (channels - 512.0) \times \frac{2 * threshold}{1024.0}$$

For every 12.8571° turn on the tuning peg, the frequency is changed on average by 1 Hz (let's call this the string calibration number). This is not uniform across all strings or even one string by itself. We first took data for all the strings, turning the pegs by 90○ and noting the frequency difference. We average the values for simplicity.

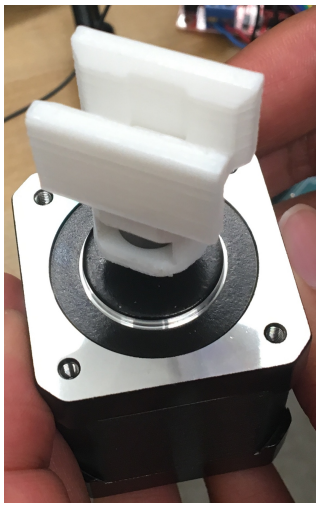
2. Frequency to Degrees - We multiply the frequency difference by the string calibration number (12.8571°/Hz) to get the number of degrees the motor should turn.

3. Degrees to Steps - Our stepper motor takes 200 steps in 360° (we know this from the spec sheet for the motor). We find the number of steps the motor should turn by multiplying the number of degrees by 200/360, the number of motor steps per degrees. This information was used in an instance of the Stepper motor class in the Arduino programming language to correctly output the correct voltage to four ports on the motor controller.

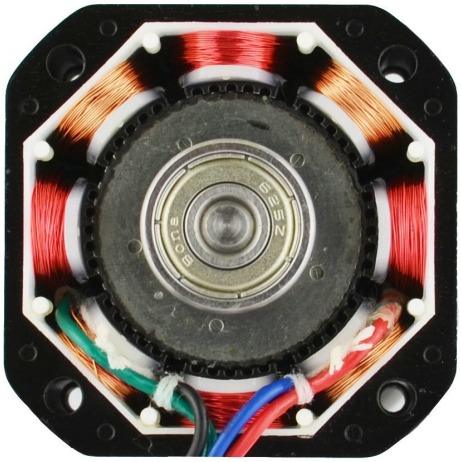
The Arduino cannot output sufficient current to make the stepper motor operate correctly. For this, we employ the motor controller (rated for 3A) to make the stepper motor (rated for 2A) turn the correct number of steps given what is determined by the Arduino. The purple, blue, green and yellow wires connected to the motor controller from the analog outputs of the Arduino. See figure6. We powered the motor controller with a 5V DC power supply plugged into an outlet. Issues with the motor are discussed in the Limitations and Possible Improvements section below (section 4).

We ran across the issue of the Arduino collecting another voltage from LabVIEW before the motor had finished all of its steps. To solve this problem, we used a simple on and off digital signal from LabVIEW to the Arduino. When this signal was on, this indicated to the Arduino when to collect another analog voltage and start tuning the guitar again.

3.3 Bipolar Stepper Motor



(a) Actual motor used with 3D printed tuning peg attachment (in white)



(b) Inside a bipolar stepper motor

Figure 7: Bipolar Stepper Motor

A stepper motor coverts electrical pulses into discrete mechanical steps. The shaft (on which the 3D printed tuning peg attachment is attached in figure 7a) moves in integer steps when electrical pulses are applied in the right order. The order in which the pulses are applied is related to the direction the motor shaft rotates. The frequency of pulses is related to the speed of the motor shaft rotating. The number of pulses applied to the motor determines the length of time the motor will rotate.

We chose a bipolar stepper motor because we needed to a high torque to turn the pegs. Stepper motors are stronger than DC motors and use an open loop for positioning control. This means that we don't need any feedback from the motor itself to continue turning it. The Arduino calculated the number of steps necessary to put the string in tune, and the motor would take that many steps.

Compared to a unipolar stepper motor, bipolar stepper motors can be more powerful because the windings are more efficiently used. It turned out that the torque of the stepper motor we purchased was still not enough to turn the pegs on all strings consistently.

The stepper motor consists of stators that are wrapped with many windings or phases (see figure 7 to see what the windings would look like if we opened our stepper motor). For a bipolar stepper motor, there is only a single winding per phase. These phases are energized with a voltage source that starts current flowing through the winding and causing polarity to to be produced on each end of the stator. A permanent magnet acts as the rotor turning the motor.

The motor was driven with current from the motor controller. Four wires go into the stepper motor from the motor controller and produce enough current for the correct amount

of steps calculated by the Arduino. The wires to the motor had to be placed in the correct arrangement on the motor controller so that they would be in sync, and counterclockwise and clockwise motions by the motor could be executed corrected.

4 Limitations and Possible Improvements

The automatic guitar tuner worked fairly well. However, there were several limitations that made the automatic tuning slightly inaccurate and difficult to execute.

Harmonics

When we observed the FFT of the signal (Amplitude vs Frequency), we noticed that there were significant harmonics being picked up by the microphone. Even with the bandpass filter in place to prevent this, we could not prevent all harmonics from reaching our analysis. There were a few techniques we tried to lower the harmonics when plucking the string before analysis. We tried plucking the string on the fretboard of the guitar rather than over the sound hole, but surprisingly, this did not make too much of a difference. The physics of standing waves on a guitar predict that changing the location we plucked the string could help decrease harmonics (because of the plucking of the string on different parts of the guitar). Unfortunately, this did not help as much as we expected.

The tuner consistently tuned the guitar strings too high when we compared it to when an actual tuner said the guitar string was in tune. This was a very tricky issue to fix and was because the fundamental frequency would not always have the highest amplitude and be picked as the peak frequency by LabVIEW. This was not a problem consistently on each string. We found higher frequency strings (i.e. the high E and B strings) had less harmonics and were more easily and accurately tuned.

These inaccuracies in frequency picked up by our signal processing might have been fixed with the perfect choice of filters or using an electric guitar we plugged in directly to the DAQ. However, harmonics are impossible to completely get rid of in the guitar tuning process.

Stepper Motor

We purchased one of the strongest bipolar stepper motors we could find online and still ran into issues with the stepper motor not being strong enough to turn the tuning pegs on the guitar. The stepper motor was rated to have a torque of 59 N cm (83.6 oz in) when running at its max rated current (2.0 A). The tuning peg also did not have a large enough area around it to increase the length of the lever arm so we couldn't use this tactic to increase the torque of the stepper motor ($\boldsymbol{T} = \boldsymbol{r} \times \boldsymbol{F}$). Unfortunately, a tuning peg did not require the same amount of force to tune it at all times. A tighter string would require more force, and as the tuner approached the right tuning for a string that was too low in pitch, the motor would have a more difficult time tuning to the correct frequency (more torque was needed).

It was also difficult to drive the stepper motor with enough current. We tried using a plug in DC power supply of 12V, but this burnt up several of our motor controllers by overheating it. Even though the motor controller was rated to produce 3.0 A of current with no issues,

we found that the spec sheet seemed inaccurate because the motor controller would burn out very quickly or not output the correct amount of current.

Arduino's ADC Inaccuracies

Lastly, we found errors in the Arduino's analog to digital conversion that caused tuning inaccuracies. The Arduino's ADC had its own intrinsic inaccuracies. It would inaccurately convert the analog voltage to a digital signal and caused the differences in frequencies calculated by LabVIEW to be inaccurately communicated to the frequency to step conversion function on the Arduino. This definitely caused the stepper motor to take the incorrect number of steps in the tuning and increased the inaccuracy of the tuning.

5 Conclusion

We were successful in creating a decently working automatic guitar tuner. We saved over $100 by building our own automatic guitar tuner as they usually sell for that or more online. The difficulties we encountered with the tuning accuracy in our tuner may have also been seen by those that bought the $100 guitar tuner online; the ratings for many of these products are not very high. Designing and constructing this tuner ended up being my favorite experience in 111a and inspired me to want to build more useful electronics in the future.

6 Acknowledgements

I would like to thank all the 111a GSIs for their tremendous help with the labs this semester and helping me understand electronics on a more fundamental level. I would also like to thank Professor Fajans for his input on this final project and tips and suggestions to improve our project's design.

111a was the only class that stopped me from changing my major to computer science last semester and only doing a physics minor. I really wanted to take this class besides what others had told me about it being so much work! Despite the incredible amount of work spent on the labs and write-ups, I am very happy to have had the opportunity to take 111a and finally apply the physics concepts I've learned about on the chalkboard to real life applications.

7 References

1. http://www.electronics-tutorials.ws/opamp/opamp_2.html
2. https://www.arduino.cc/en/main/arduinoBoardUno
3. http://www.instructables.com/id/Control-DC-and-stepper-motors/
4. http://www.mikeholt.com/instructor2/img/product/pdf/1245861713sample.pdf

5. https://www.amazon.com/Roadie-Tuner-Automatic-Guitar/dp/B00ON9WQ1A

6. https://www.amazon.com/Stepper-Bipolar-4-lead-Connector-Printer/dp/B00PNEQKC0

7. http://www.ece.ucsb.edu/Faculty/rodwell/Classes/ece2c/labs/Lab1b-2C2007.pdf

8. https://engineering.purdue.edu/ece477/Archive/2009/Spring/S09-Grp04/images/open.jpg

8 Appendix

Screenshots from LabVIEW

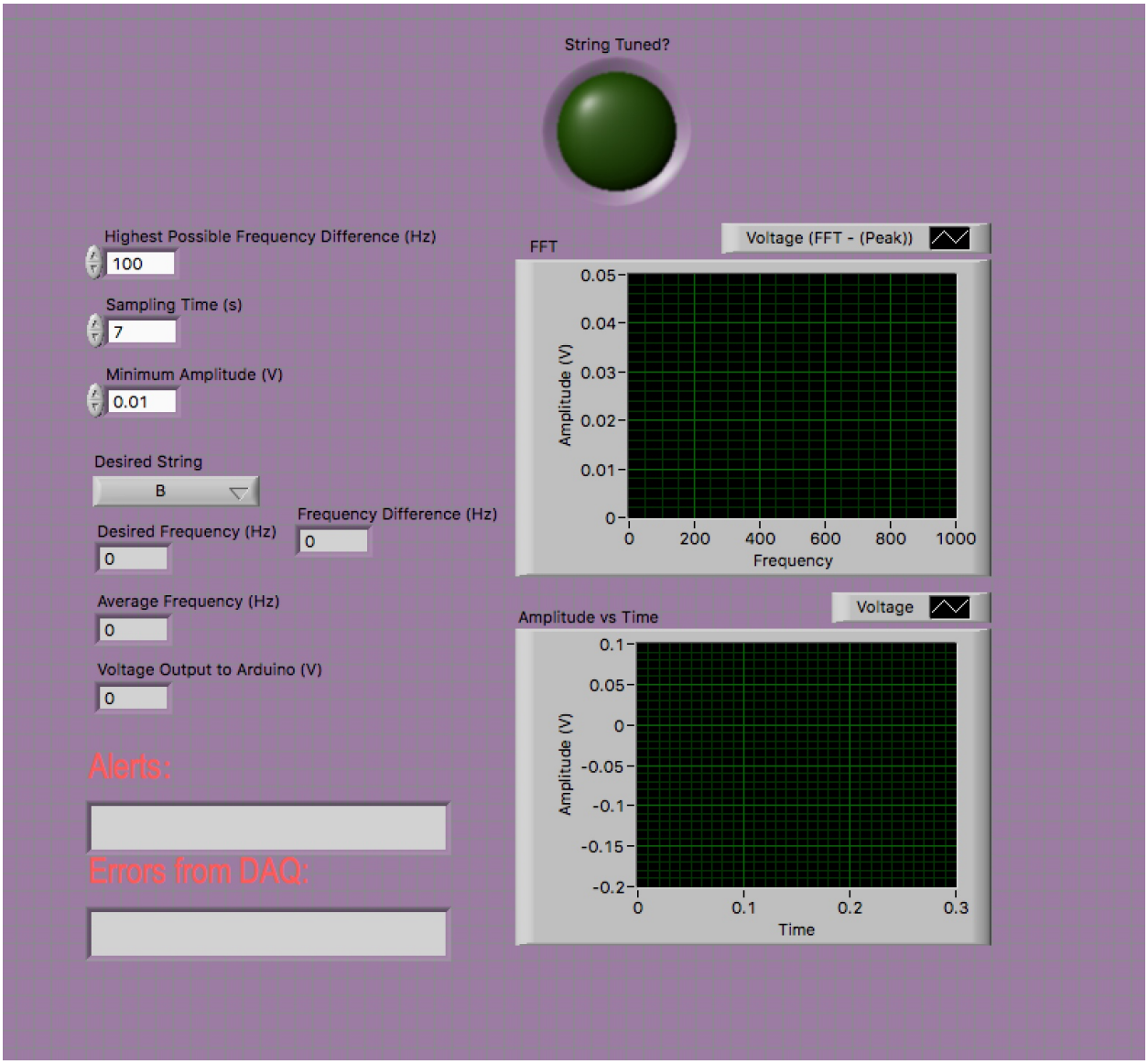


Figure 8: Front Panel

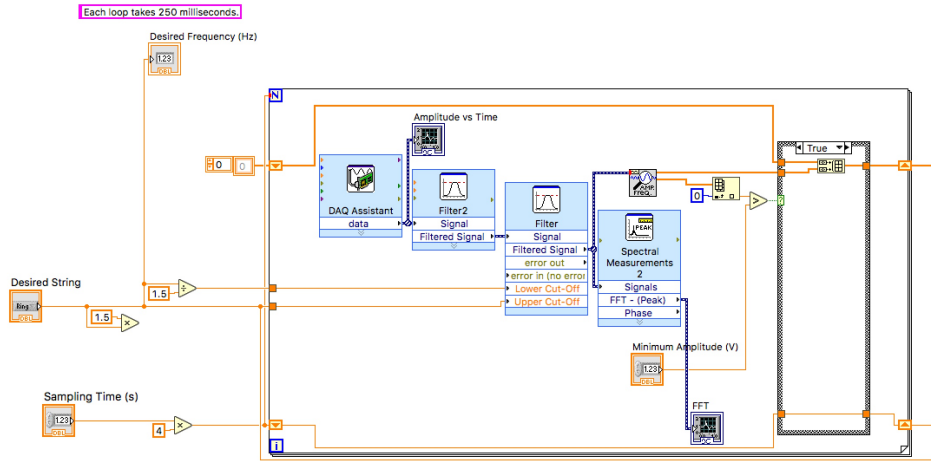


Figure 9: Block Diagram - First Half

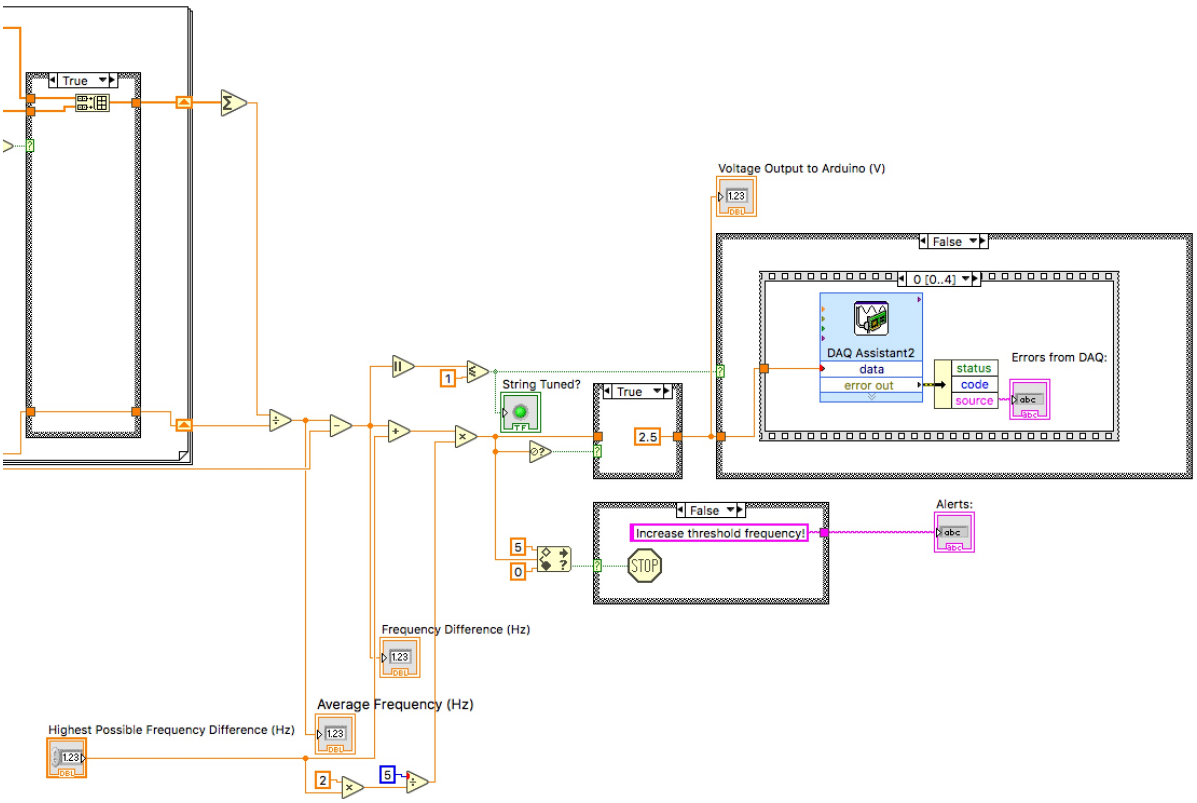


Figure 10: Block Diagram - Second Half

Standard Guitar Tunings

String (pitch)	Frequency (Hz)
E (higher)	329.63
B	246.94
G	196.00
D	146.83
A	110.00
E (lower)	82.41

Arduino Code

```
#include <Stepper.h>

#define MOTOR_STEPS 200.0
#define CHANNEL_RANGE 200.0
#define ADC_ERR 5
#define ANALOG_IN_PIN 5
#define DIGITAL_IN_PIN 13
#define SPEED 15 //rpm

Stepper stepper(MOTOR_STEPS, 4, 5, 6, 7);
const double string_calibration=12.8571;

double channel_to_freq(int channel){
  double freq;
  freq = double(channel-512.0)*(CHANNEL_RANGE/1024.0);
  Serial.println("frequency:");
  Serial.println(freq);
  return freq;
}

double freq_to_degrees(double freq){
  double deg;
  deg = freq*string_calibration;
  return deg;
}

double degrees_to_steps(double deg){
  int steps;
  steps = int(deg*(MOTOR_STEPS/360.0));
  return steps;
}
```

```
int channel_to_steps(int channel){
  int steps;
  steps = int(degrees_to_steps(freq_to_degrees(channel_to_freq(channel))));
  return steps;
}

int channel;
int steps=0;
int timing;
int d_time;

void setup(){

  Serial.begin(9600);
  pinMode(DIGITAL_IN_PIN,INPUT);

  // set the speed of the motor to 30 RPMs
  stepper.setSpeed(SPEED);
}

void loop(){

  timing = digitalRead(DIGITAL_IN_PIN);
  //Serial.println("reading");
  //Serial.println(timing);
  if (timing == 1){
    channel = analogRead(ANALOG_IN_PIN);
    steps = channel_to_steps(channel);
    Serial.println(channel);
    Serial.println(steps);
    stepper.step(steps);
    d_time=int(SPEED*(steps/200.0)*(1.0/60.0))*1000+100;
    delay(d_time);

    while (timing == 1){
      timing = digitalRead(DIGITAL_IN_PIN);
      delay(100);
      Serial.println("waiting for pin low");
    }
  }
}
```