**University of New Haven Tagliatela
College of Engineering
Department of Electrical & Computer Engineering
and Computer Science**

ELEC 3355-01
Control Systems

Automatic Guitar Tuner (Report)

Matthew Windsor
Carolina Sousa De La Cruz

Submitted To: Dr. Shayok Mukhopadhyay

May 5th, 2024

Contents

Introduction 3
Theory 3
Hardware 3
Software 5
    MATLAB 5
    Arduino 7
Results 11
Conclusion 15
Appendix 16
References 18

Introduction

The overall objective of this project was to create an automated guitar tuner by measuring the frequency of a string, comparing that frequency to a set point, and using a form of PID controller to get the input equal to the set point. Based on the comparison between the input and set point, the motor will turn in the appropriate direction until the frequency measured matches the setpoint. This was done using an Arduino Uno, a Bojack L298N motor driver, a 4.8V motor which was taken from a WARRIOR 4.8V cordless screwdriver, a Scarlett Audio Interface, MATLAB, and the Arduino software. A bang-bang controller was initially designed which was developed into a P controller.

Theory

To tune a guitar, the frequency of a note depends on the length of the string and of the string. To do this using a guitar tuner, an input signal must first be detected. To measure smaller increments of pitch, "cents" are used. This was a factor considered when developing this control system. Once the input signal is measured, the obtained frequency will be used as the process variable and must be compared to a setpoint. In the P controller, the controller does the work of calculating the error, which is the difference between the setpoint and the process variable. The controller then multiplies the error by a constant k_P, which acts as the proportional gain. This gain determines the sensitivity of the controller when it responds to changes in the error. The higher the k_p value, the higher the sensitivity of the system. The output of the controller is then taken and used to adjust the control variable. For the guitar tuner, the control variable will be the direction and speed in which the motor is spinning to best match the setpoint's frequency. To bring the process variable closer to the setpoint, the error must be reduced as much as possible. Using a P-controller is effective in reducing the steady-state errors; however, it may not completely get rid of the oscillations or overshoot in the system. To achieve optimal performance without any instabilities or sluggish response times, the proportional gain must be carefully tuned.

Hardware

The hardware items required to automate the tuning of a guitar include an audio interface, motor, a peg winder bit, an Arduino, a Bojack L298N motor driver, and a 9V wall adapter. The easiest route to turn the pegs was by using a motor with the peg winder bit. By using a WARRIOR 4.8V cordless screwdriver, the motor was easily accessible while also having the necessary chuck to hold the bit (Figure 1). To control the motor, the L298N motor driver was used due to its' easy use and availability (Figure 2). The motor driver was powered by a 9V wall adapter to keep the power input constant.

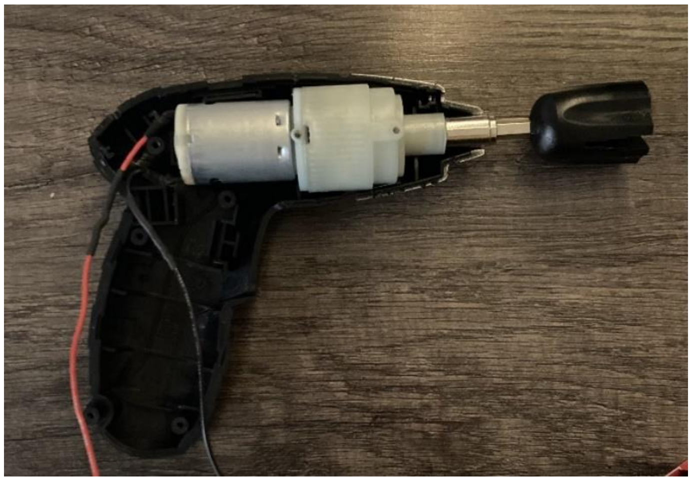


Figure 1: Motor & Peg Winder Bit

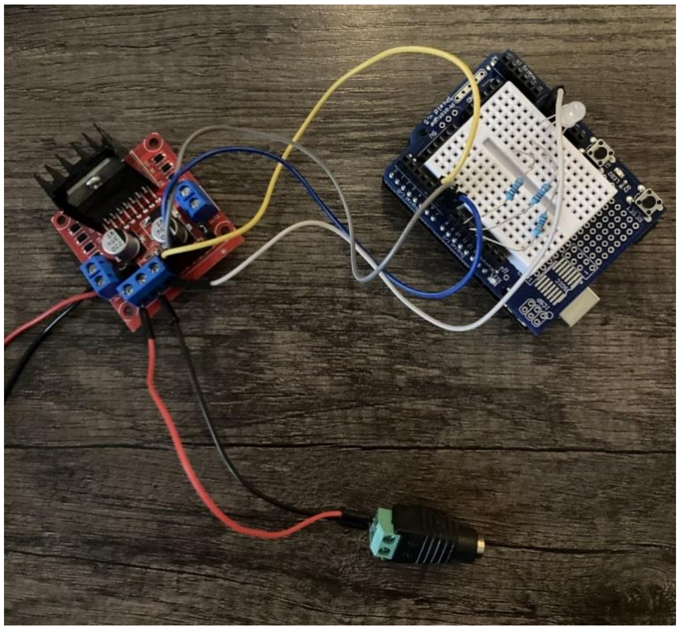


Figure 2: Motor Driver & Arduino

To capture the audio input of the guitar, a Focusrite Scarlett 2i2 3rd Gen audio interface was used (Figure 3). This allowed the program to capture only the guitar's audio at a predefined sample rate. The computer receives the signal as a microphone input which is easily captured through MATLAB.

Figure 3: Focusrite Scarlett Audio Interface

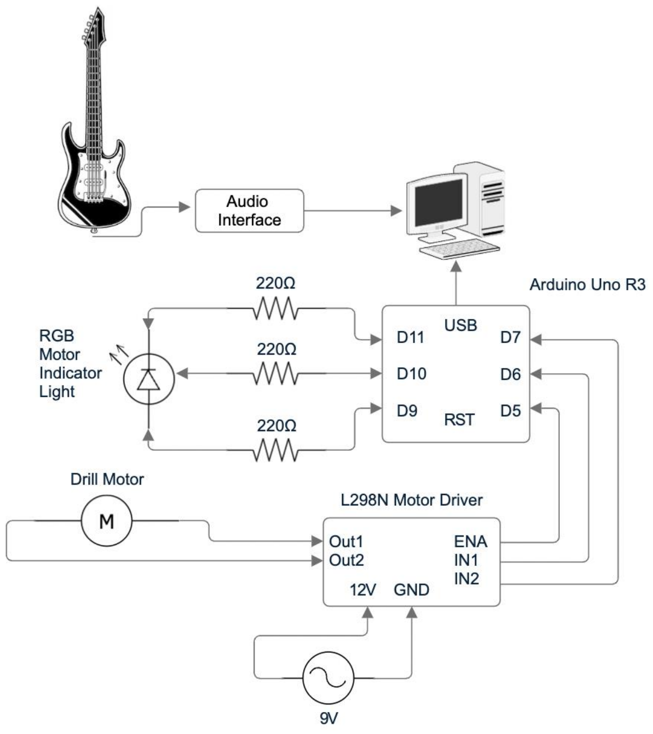
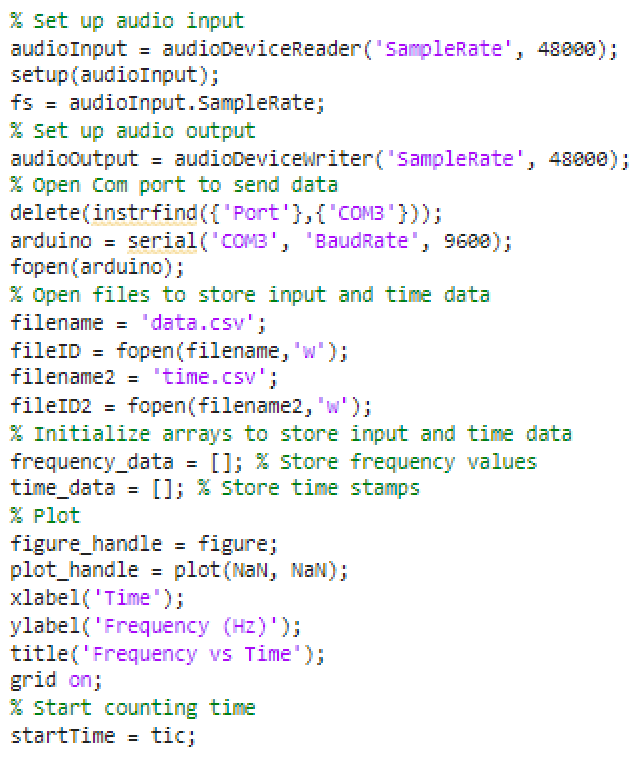


Figure 4: System Schematic Diagram

Software

MATLAB

The code to read the guitar's audio and detect the main frequency was done in MATLAB. The setup code shown in Figure 5, first shows using the function audioDeviceReader to capture the input microphone. The sample rate is 48,000 because the audio interface I am using has a sample rate of 48,000. The output is also initialized, this allows the software to record and playback the audio. To send the data to an Arduino, com port 3 is initialized with a baud rate of 9600 and the connection is opened. To save the frequency and time data, two files are initialized with corresponding names and are opened; the data that will be saved in the files are stored in the initialized arrays underneath. Finally, a plot that shows Frequency vs Time is initialized and the clock starts counting.



```
% Set up audio input
audioInput = audioDeviceReader('SampleRate', 48000);
setup(audioInput);
fs = audioInput.SampleRate;
% Set up audio output
audioOutput = audioDeviceWriter('SampleRate', 48000);
% Open Com port to send data
delete(instrfind({'Port'},{'COM3'}));
arduino = serial('COM3', 'BaudRate', 9600);
fopen(arduino);
% Open files to store input and time data
filename = 'data.csv';
fileID = fopen(filename,'w');
filename2 = 'time.csv';
fileID2 = fopen(filename2,'w');
% Initialize arrays to store input and time data
frequency_data = []; % Store frequency values
time_data = []; % Store time stamps
% Plot
figure_handle = figure;
plot_handle = plot(NaN, NaN);
xlabel('Time');
ylabel('Frequency (Hz)');
title('Frequency vs Time');
grid on;
% Start counting time
startTime = tic;
```

Figure 5: MATLAB Setup Code

Once the clock starts counting, a real time loop begins (Figure 6). The loop is a try-catch block that executes code unless there is an error. If no error occurred after start up, first the audio is captured and goes through a pitch detection algorithm. The window length and overlap length are calculated two different ways, the minimum of the two options is chosen. This is done so that the window length is not greater than the length of the audio and so that there is some overlap between windows but not more than the window itself. Using this data, the pitch function is called using the Subharmonic-to-Harmonic Ratio (SRH) method and the fundamental frequency is saved to f0. Since f0 is an array, the main frequency is extracted by taking the mode.

The main frequency is then saved into an array along with the time stamp it was taken at. The plot is then updated with the data. The main frequency and time that was just recorded is then saved to their according file and the main frequency is sent over serial to an Arduino. The code then outputs the guitar audio through headphones and the loop continues until stopped. The catch block captures any error and stops the code before any damage is done. After the loop is terminated, the files and audio connections are closed (Figure 7).

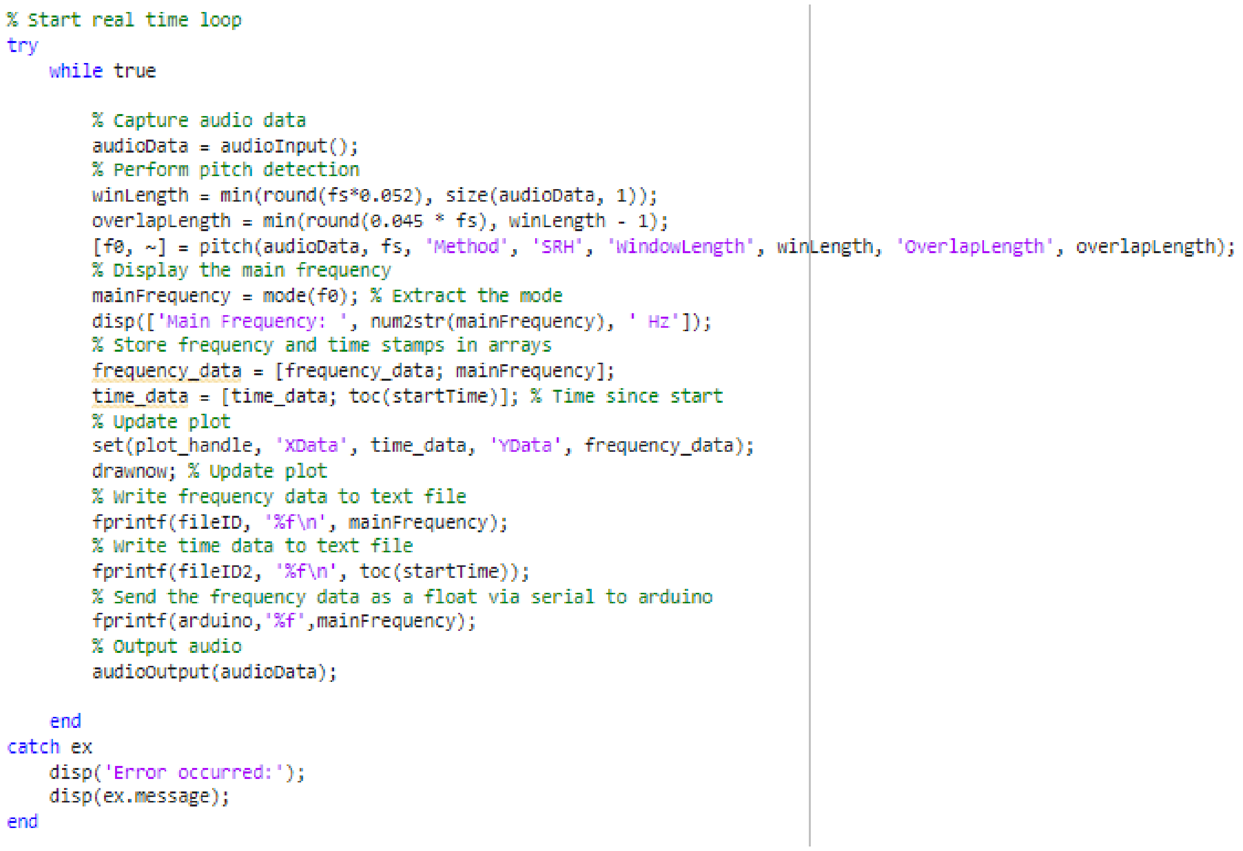

```
% Start real time loop
try
    while true

        % Capture audio data
        audioData = audioInput();
        % Perform pitch detection
        winLength = min(round(fs*0.052), size(audioData, 1));
        overlapLength = min(round(0.045 * fs), winLength - 1);
        [f0, ~] = pitch(audioData, fs, 'Method', 'SRH', 'WindowLength', winLength, 'OverlapLength', overlapLength);
        % Display the main frequency
        mainFrequency = mode(f0); % Extract the mode
        disp(['Main Frequency: ', num2str(mainFrequency), ' Hz']);
        % Store frequency and time stamps in arrays
        frequency_data = [frequency_data; mainFrequency];
        time_data = [time_data; toc(startTime)]; % Time since start
        % Update plot
        set(plot_handle, 'XData', time_data, 'YData', frequency_data);
        drawnow; % Update plot
        % Write frequency data to text file
        fprintf(fileID, '%f\n', mainFrequency);
        % Write time data to text file
        fprintf(fileID2, '%f\n', toc(startTime));
        % Send the frequency data as a float via serial to arduino
        fprintf(arduino,'%f',mainFrequency);
        % Output audio
        audioOutput(audioData);

    end
catch ex
    disp('Error occurred:');
    disp(ex.message);
end
```

Figure 6: MATLAB Main Loop Code

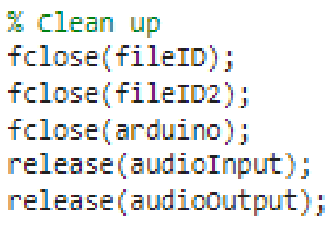

```
% Clean up
fclose(fileID);
fclose(fileID2);
fclose(arduino);
release(audioInput);
release(audioOutput);
```

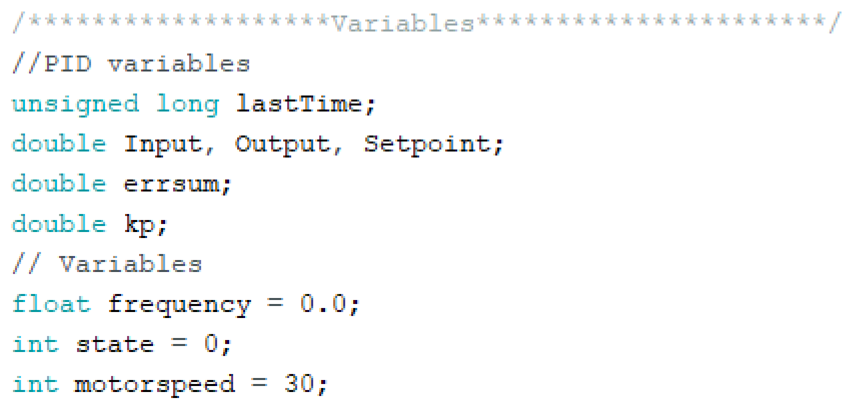
Figure 7: MATLAB Clean-Up Code

Arduino

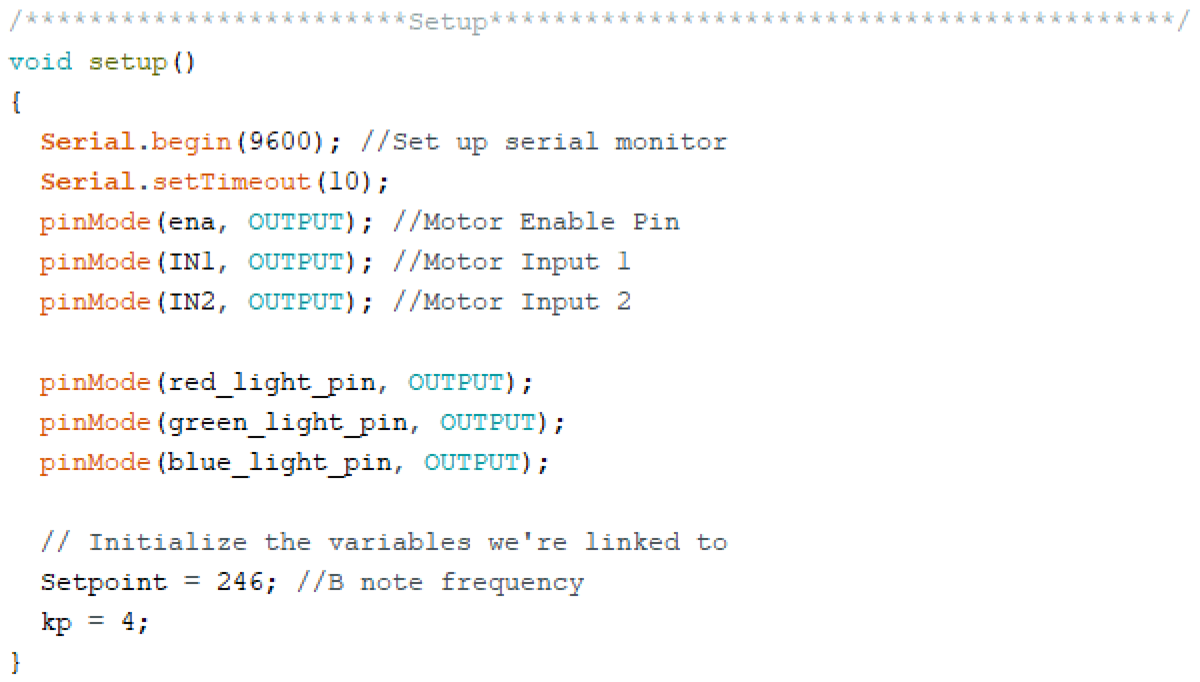
An Arduino is used for the PID controller. Based on the frequency input, the math to determine speed and direction is calculated, and a corresponding motor command is outputted. The pin initializations and functions are shown in the Appendix as those do not need explanations.

Figure 8 shows the variables and setup of the code. Some important notes here are:
1) This code only includes k_P, explained in Conclusion.

2) The motor speed is initialized as 30. This speed was an offset which is described in the Results.
3) The set point is set to 246 which is the frequency of an open B string.



```
/******************Variables**********************/
//PID variables
unsigned long lastTime;
double Input, Output, Setpoint;
double errsum;
double kp;
// Variables
float frequency = 0.0;
int state = 0;
int motorspeed = 30;
```



```
/************************Setup*******************************************/
void setup()
{
  Serial.begin(9600); //Set up serial monitor
  Serial.setTimeout(10);
  pinMode(ena, OUTPUT); //Motor Enable Pin
  pinMode(IN1, OUTPUT); //Motor Input 1
  pinMode(IN2, OUTPUT); //Motor Input 2

  pinMode(red_light_pin, OUTPUT);
  pinMode(green_light_pin, OUTPUT);
  pinMode(blue_light_pin, OUTPUT);

  // Initialize the variables we're linked to
  Setpoint = 246; //B note frequency
  kp = 4;
}
```

Figure 8: Arduino Setup Code

The main loop first starts in state 0. Figure 9 shows that in state 0, a LED changes color from red to yellow to green. This is to show the user that the code to start the controller in beginning and that the string should be plucked. After the LED is lit green, the state changes to 1 and the LED changes once more to blue, indicating the controller started.

```
/***********************Main*******************************************/
void loop() {
    // State 0: Start state
    while (state == 0){
    RGB_color(255, 0, 0); // red
    delay(1000);
    RGB_color(255, 255, 0); // yel
    delay(1000);
    RGB_color(0, 255, 0); // Green
    delay(1000);
    state = 1;
  }

  // Program begin indicator
    RGB_color(0, 0, 255); // blu
    delay(1000);
```

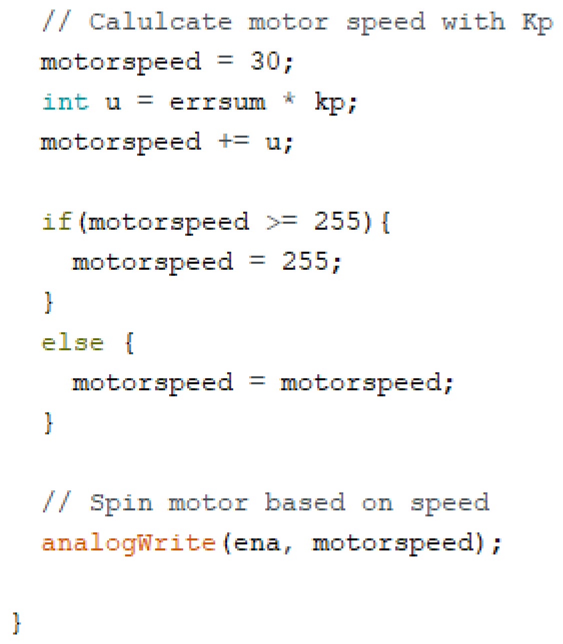
Figure 9: Arduino Main Loop (State 0)

Once in state 1 (Figure 10), the Serial Monitor checks if there is anything written there. If so, it saves the data as a string and changes it into a float. The error signal e(t) is then calculated by subtracting the input from the set point. Before changing the speed, the direction is first found. If the error signal is negative, that means the input is larger than the setpoint and the motor must turn clockwise. If the error signal is positive, that means the input is smaller than the setpoint and the motor must turn counterclockwise. Finally, if those two options are not the case, the error signal is 0 and the guitar string is tuned.

Figure 10: Arduino Main Loop (State 1)

```
// State 1: PID loop
  while (state == 1){
    if (Serial.available() > 0){
      String str = Serial.readString();
      Input = str.toFloat();
    }
    // Calculate error signal e(t)
    errsum = Setpoint - Input;
    // Find which way motor needs to spin
    if(errsum > 1){
      counterclockwise();
    }else if(errsum < -1){
      errsum * -1;
      clockwise();
    }
    else {
      finish();
    }
```

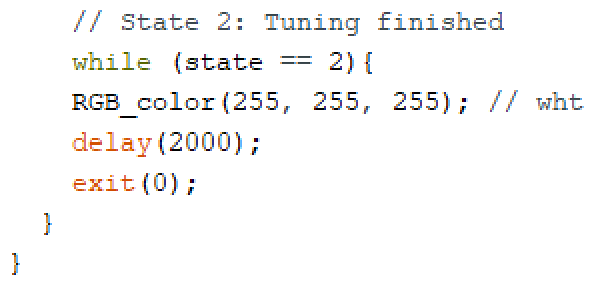
Next the motor speed is calculated (Figure 11). The offset is reinitialized so the future calculation of the motor speed is not increasing with every loop. The control signal u(t) is calculated by multiplying our error signal with k_P. The control signal is then added to the offset motor speed. Based on the final motor speed, if it is above 255, it is set to 255. Otherwise, it stays as it is and written to the motor.



```
  // Calulcate motor speed with Kp
  motorspeed = 30;
  int u = errsum * kp;
  motorspeed += u;

  if(motorspeed >= 255){
    motorspeed = 255;
  }
  else {
    motorspeed = motorspeed;
  }

  // Spin motor based on speed
  analogWrite(ena, motorspeed);

}
```

Figure 11: Arduino Main Loop (State 1)

The final state, state 2 shown in Figure 12, is called when the tuning is finished. As one can see, there is no way to reach this state. This is due to an issue that was not fixed and is described in the <u>Results</u> section. This state changes the LED to white indicating the tuning is finished and exits the program.



```
    // State 2: Tuning finished
    while (state == 2){
    RGB_color(255, 255, 255); // wht
    delay(2000);
    exit(0);
  }
}
```

Figure 12: Arduino Main Loop (State 2)

Results

First, the reason the motor speed was offset by 30 was due to a common trend that was found during the testing of the "Bang-Bang" controller. The common trend was that the motor was not receiving enough power. We found that the motor would not turn on with a speed under 30. With the motor speed starting at 0, having only k_P, k_I, or k_D, would not work as the error decreases. In the case of only a P-controller, as the error decreases say to 1, the P-gain must be 30 to move the motor. This proves difficulties when the error is larger than 8-9.

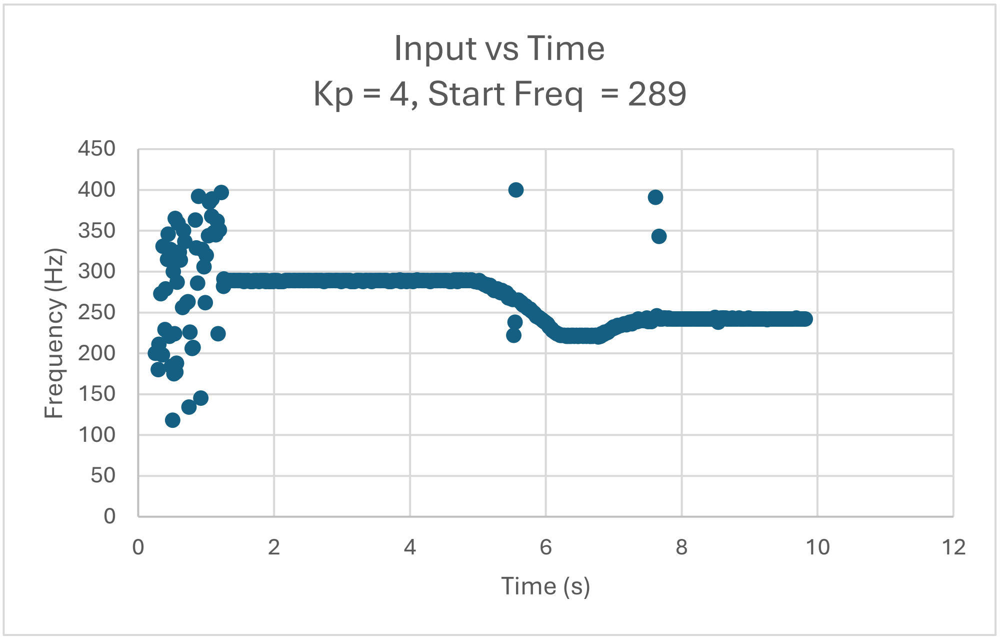
To fix this issue, the motor speed has an offset starting at 30. As the error decreases to 1, the P-gain does not need to be large to move the motor. This allows for more speeds to be achieved when the error signal increases to infinity, therefore more control.

The MATLAB code provides a graph for the Input vs Time as well as all the data points for each. The data points were copied and pasted into Excel to calculate all the error signal, control signal, and the final motor speed. The Excel file along with all the MATLAB graphs are attached in the zip file. The report will use the Excel graphs as they are easier to read.

In the beginning of each graph, the frequency takes a second or two to stabilize. In total, 4 different k_P's were tested: 4, 5, 6, and 10. Two of these cases proved good results and were tested with the input frequency high and low. The other two were extreme cases where the output did not settle or settled at the wrong frequency.

$\underline{k_P = 4}$

When $k_P = 4$, the starting frequency began at 289Hz and settles at 242Hz, leaving an error signal of 4. The max value reached 222Hz after the initial descent or an overshoot of 10.8%. This is one of the bad cases due to the system settling at the incorrect value.

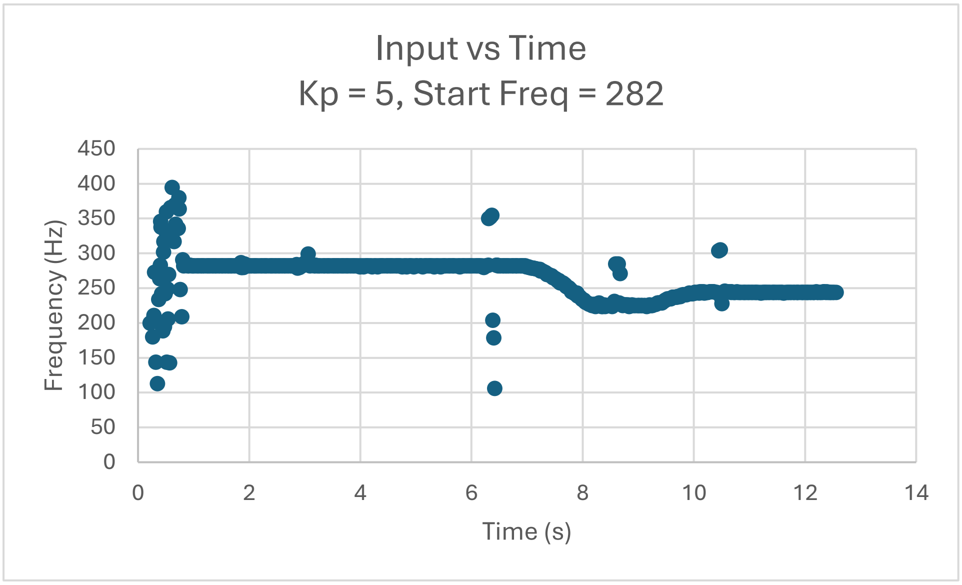


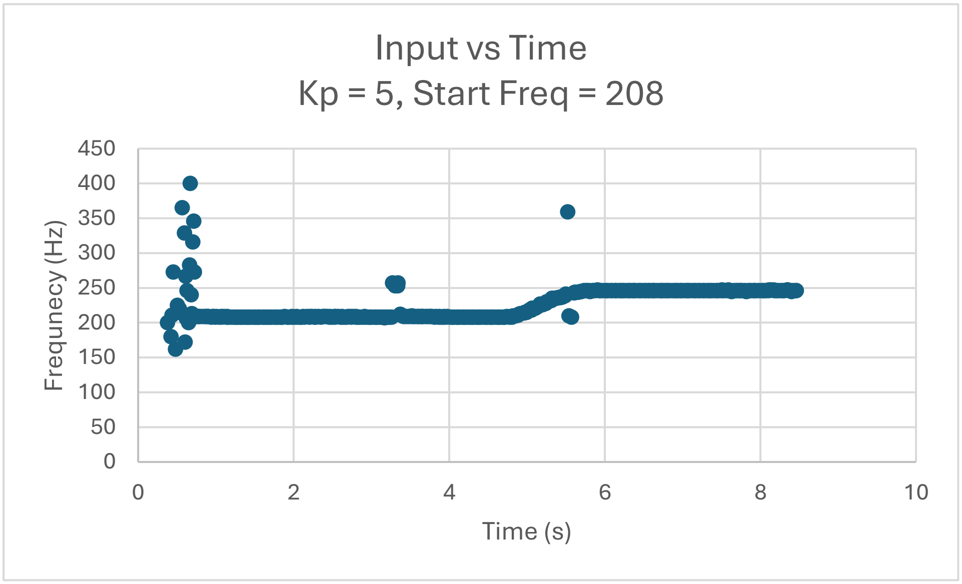
$k_P = 5$

When $k_P = 5$, the first starting frequency began at 282Hz and settles at 244Hz, leaving an error signal of 2. The max value reached 225Hz after the initial descent, or an overshoot of .93%.

The second starting frequency began at 208Hz and settles at 246Hz, leaving an error signal of 0. The max value was 246Hz which means the percent overshoot was 0%. The signal fluctuated from 246Hz to 247Hz until turned off but successfully tuned the guitar.

This is the best case recorded due to the system stabilizing at the correct setpoint with minimal overshoot.



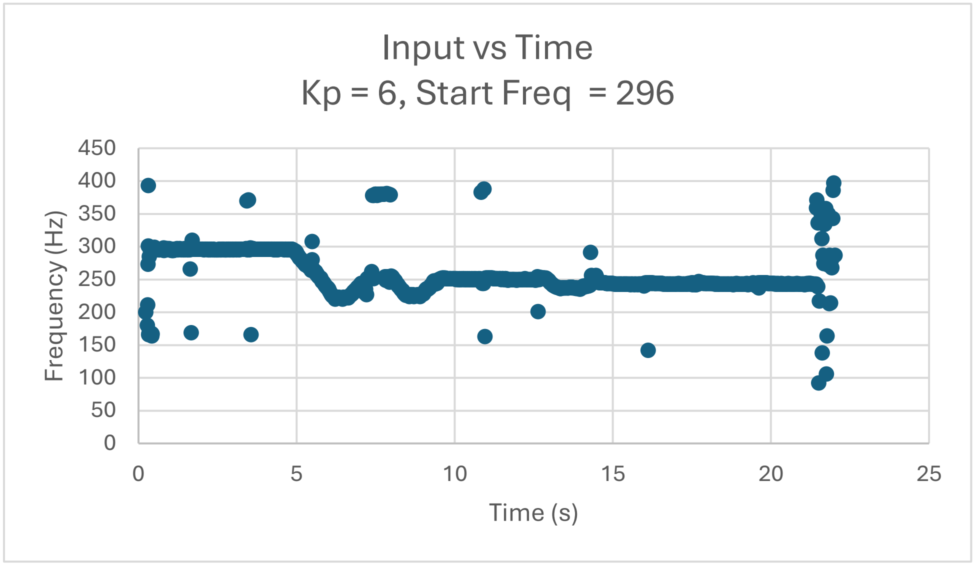


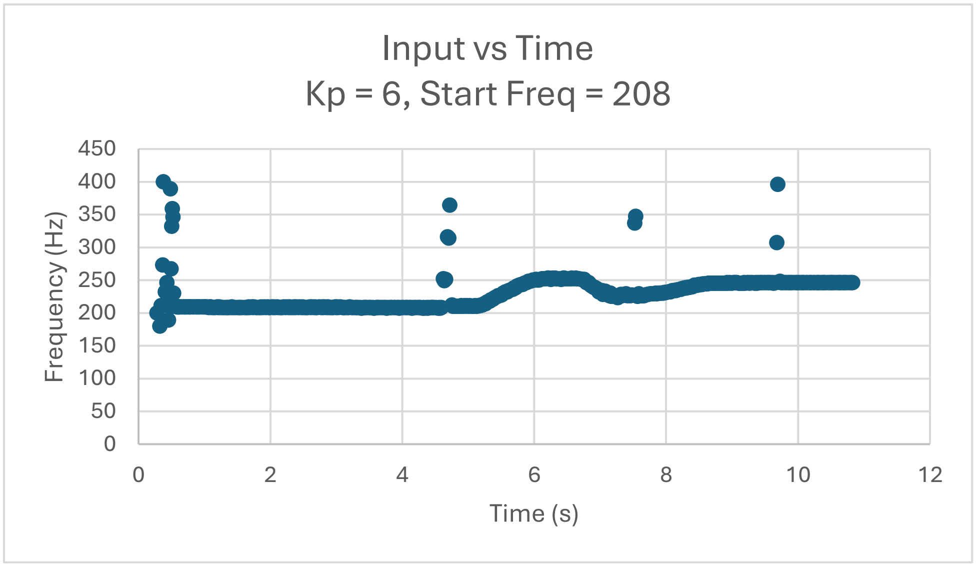
$\underline{k_P = 6}$

When k_P = 6, the first starting frequency began at 296Hz and settles at 243Hz, leaving an error signal of 3. The max value reached 220Hz after the initial descent, or an overshoot of 11.8%.

The second starting frequency began at 208Hz and settles at 246Hz, leaving an error signal of 0. The max value reached 252Hz after the initial ascent, or an overshoot of 2.4%. The signal stabilized and tuned the guitar.

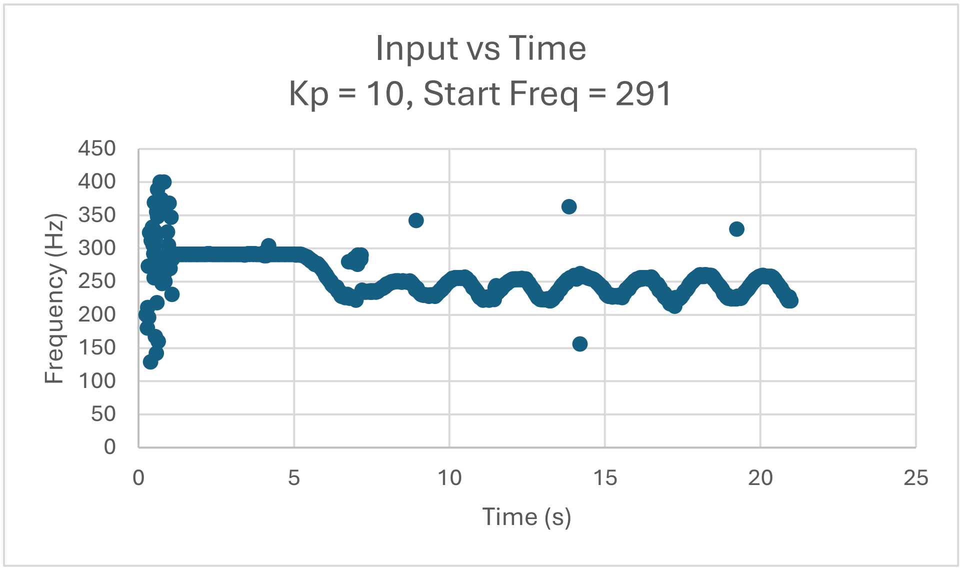
This is an acceptable case but not the best as it does settle to the setpoint or near but does it in more oscillations than when k_P = 5.





$k_P = 10$

When $k_P = 10$, the starting frequency began at 291Hz and never settled. The output oscillated at about the 246Hz frequency. Looking at the control signal and motor speeds, the motor was nearly always at max speed and did not slow down when nearing the setpoint. This is the worst case scenario.



Conclusion

Developing an automated guitar tuner through the utilization of the hardware, software, and a P-controller was a challenging, yet rewarding task. Overall, a system that measures the frequency of a guitar string, compares it to a setpoint, and adjusts the motor's direction and speed accordingly was successfully created. The transition from a basic "Bang-Bang" controller to a more refined P-controller was critical in achieving smoother and more precise tuning adjustments. The reason for not adding the I-term is time constraints. The D-term was not added for the reason of frequency spikes. Every time the string was plucked, the frequency would spike. Addressing hardware challenges, such as ensuring that there was consistent power delivered to the motor, and taking advantage of software tools such as MATLAB for audio processing and data analysis were key in the project's success. The offset motor speed strategy used to overcome initial motor power issues was effective in providing better control.

The results demonstrated the system's ability to tune the string to the desired frequency and observing how the k_P changes the system's accuracy provides a better understanding of how a P-controller operates. $k_P = 4$ showed the bottom limit of k_P, any lower and the final value will settle but will get further away from the setpoint. $k_P = 6$ showed the top limit of k_P, any higher and the final value will settle after more oscillations or never settle. $k_P = 10$ showed the case of never settling, any higher and the oscillations will continue. $k_P = 7$ to 9 were not tested as these cases would end in a result similar to $k_P = 6$ or $k_P = 10$.

Some future improvements would be addressing the delay between the communication from the hardware, Arduino, and MATLAB, as well as going from a P-controller to a PI-controller. Adding the I-term will help the system's stability, adaptability to variable load conditions, and will also help to improve transient response, overall enhancing the control performance. As for the D-term, future work would have to be done to minimize noise to ensure proper results. This project highlights the intersection between hardware, software, mathematics, and music theory while also showcasing the nature of problem-solving and optimization in real-world applications.

Attached in the Appendix as well as the zip file is a video containing the automated tuning of a guitar.

Appendix

https://youtu.be/OsgHPaL9daw

```
/***********************I/O's*************************************/
// Arduino pins for RGB LED
int red_light_pin= 11;
int green_light_pin = 10;
int blue_light_pin = 9;
//Motor Variables
int ena = 5;
int IN1 = 6;
int IN2 = 7;

/*****************Creation of the Function*************************/
// Rotate the Motor A clockwise
void clockwise(){
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
}
// Rotate the Motor A counter-clockwise
void counterclockwise(){
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
}
// Stop
void finish(){
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
}
// LED
void RGB_color(int red_light_value, int green_light_value, int blue_light_value){
  analogWrite(red_light_pin, red_light_value);
  analogWrite(green_light_pin, green_light_value);
  analogWrite(blue_light_pin, blue_light_value);
}

/***********************Function Declaration***************************/
// Functions
void clockwise();
void counterclockwise();
void finish();
void RGB_color(int red_light_value, int green_light_value, int blue_light_value);
```

References

1. https://www.sciencedirect.com/topics/engineering/piezoelectric-sensor
2. https://theacousticguitarist.com/how-guitar-tuners-work/
3. https://maxbotix.com/blogs/blog/how-ultrasonic-sensors-work
4. https://robocraze.com/blogs/post/sound-sensor-working-and-its-applications
5. https://www.arduino.cc/reference/en/libraries/pid/
6. https://www.arrow.com/en/research-and-events/articles/pid-controller-basics-and-tutorial-pid-implementation-in-arduino
7. https://github.com/br3ttb/Arduino-PID-Library/blob/master/examples/PID_AdaptiveTunings/PID_AdaptiveTunings.ino
8. https://www.instructables.com/Arduino-Frequency-Detection/
9. https://www.arrow.com/en/research-and-events/articles/pid-controller-basics-and-tutorial-pid-implementation-in-arduino
10. https://github.com/br3ttb/Arduino-PID-Library/blob/master/examples/PID_Basic/PID_Basic.ino
11. https://www.arduino.cc/reference/en/libraries/pid/
12. https://www.teachmemicro.com/arduino-pid-control-tutorial/
13. https://www.mathworks.com/help/audio/gs/real-time-audio-in-matlab.html
14. https://www.mathworks.com/help/matlab/ref/audioread.html
15. https://www.mathworks.com/help/audio/gs/real-time-audio-in-simulink.html
16. https://www.mathworks.com/help/matlab/ref/audiorecorder.html
17. https://www.mathworks.com/help/dsp/ref/dsp.audiorecorder-class.html
18. https://www.mathworks.com/help/signal/ref/spectrogram.html
19. https://www.mathworks.com/matlabcentral/answers/299334-how-to-extract-the-pitch-of-a-signal#comment_385050
20. https://www.mathworks.com/help/audio/ref/audiodevicereader-system-object.html
21. https://www.mathworks.com/help/audio/gs/audio-io-buffering-latency-and-throughput.html
22. https://www.mathworks.com/help/audio/gs/real-time-audio-in-matlab.html
23. https://www.mathworks.com/help/dsp/ref/dsp.spectrumanalyzer-system-object.html
24. https://www.youtube.com/watch?v=Ey4xoG970Go
25. https://www.arduino.cc/reference/en/language/functions/communication/serial/settimeout/
26. https://www.hackster.io/muhammad-aqib/arduino-rgb-led-tutorial-fc003e