Robotic Electric Guitar Tuner

Elgitarrstämmande Robot

MARTIN GYLLING

RUBEN SVENSSON
Robotic electric guitar tuner

Bachelor’s thesis in mechatronics

MARTIN GYLLING
RUBEN SVENSSON

Bachelor’s Thesis at ITM
Supervisor: Damir Nesic
Examiner: Nihad Subasic

MMK 2017:24 MDAB 642
Abstract

This report presents the process of building a system that tunes a guitar with the help of an Arduino board, writing algorithms for processing signals from a microphone and controlling the tension of the strings according to the desired fundamental frequency. The research question for this project is how different sample rates affects the accuracy of the tuning, different sample rates were examined in order to find a sample rate that yields the best precision in tuning across all strings of the guitar.
Referat

Elgitarrsstämmande Robot

Denna rapport går igenom byggandet av ett system som har som funktion att stämma en gitarr med hjälp av ett Arduino kort, samt implementering av algoritmer för behandling av mikrofon signaler och styrning av strängarnas spänning enligt den önskade grundtonen. Forskningsfrågan för detta projekt är hur olika samplingstakter påverkar stämningens noggrannhet. Olika samplingstakter undersöktes för att finna den som ger den bästa noggranheten för alla strängar.
Acknowledgements

We would like to thank our supervisor Damir Nesic for guiding us all the way. We would like to thank the Lab assistants for saving us when we felt like we were in over our heads. We would like to thank Nihad Subasic for great lectures and giving us the opportunity to pursue our crazy ideas.
Nomenclature

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACF</td>
<td>Autocorrelation Function</td>
</tr>
<tr>
<td>AD</td>
<td>Analog to digital</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog-digital converter.</td>
</tr>
<tr>
<td>DC</td>
<td>Direct current.</td>
</tr>
<tr>
<td>JND</td>
<td>Just noticable difference.</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse width modulation</td>
</tr>
</tbody>
</table>

Symbols

- \( \tau \): The lag of the Autocorrelation function.
- \( W \): Window size of a sampled signal.
# Contents

1 Introduction .................................................. 1  
  1.1 Background .................................................. 1  
  1.2 Purpose ....................................................... 1  
  1.3 Scope ......................................................... 1  
  1.4 Method ....................................................... 2  

2 Theory .......................................................... 3  
  2.1 Sound Perception and Musical Scales ......................... 3  
  2.2 YIN Autocorrelation ........................................... 4  
  2.3 PID-Control .................................................. 7  

3 Demonstrator .................................................. 9  
  3.1 Problem formulation .......................................... 9  
  3.2 Hardware and Electronics .................................... 12  
    3.2.1 Amplifier ................................................ 12  
    3.2.2 H-bridge driver and Motor .............................. 13  
  3.3 Software ..................................................... 14  
    3.3.1 Signal measurement ..................................... 14  
    3.3.2 Frequency processing ................................... 15  
    3.3.3 PID-Control and driver ................................ 15  
  3.4 Results ...................................................... 16  

4 Discussion and conclusions ................................... 21  
  4.1 Discussion .................................................. 21  
  4.2 Conclusions ................................................ 22  

5 Recommendations and Future work .............................. 23  
  5.1 Recommendations ........................................... 23  
  5.2 Future work ................................................ 23  

Bibliography ...................................................... 25  

Appendices ......................................................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Nozzle blueprint</td>
<td>28</td>
</tr>
<tr>
<td>B Results of the experiment</td>
<td>29</td>
</tr>
<tr>
<td>C Code</td>
<td>31</td>
</tr>
</tbody>
</table>
List of Figures

2.1 (a) Displays two sine-waves with similar but not equal frequencies. (b) Displays the resulting beating. The figure was made with MathWorks MATLAB 2016b ...................................................... 3

2.2 (a) Example of waveform containing recording of the A-string of an acoustic guitar. (b) Autocorrelation function (ACF) of the guitar signal calculated using (2.3). The figure was made with MathWorks MATLAB 2016b ........................................................................ 5

2.3 (a) Difference function calculated using the example waveform from Fig. 2.3(a). (b) Cumulative mean difference function of the same example waveform. Here the function starts at 1 and remains high until the dip corresponding to the period of the signal. The figure was made with MathWorks MATLAB 2016b ...................................................... 7

3.1 This picture shows how the system looked when it was completely assembled. ................................................................. 9

3.2 This chart represents the system layout of the robot tuner. The figure was made with Apple Keynote ........................................... 10

3.3 Flow-chart of the software implemented. The figure was made with draw.io ................................................................. 11

3.4 The circuit shows the amplifier as well as the low-pass filter of the input signal. This figure was made in Adobe Illustrator CC 2015 12

3.5 This picture shows the amplifier component of the finished assembly. ................................................................. 13

3.6 This picture shows the copper plated board for the H-bridge. ........................................................................................ 14

3.7 The 3D printed nozzle that was designed to fit the tuning pegs of a guitar and the shaft of the DC motor. ................................. 14

3.8 A plot of the average values of all errors for each sample rate. Plotted using MathWorks MATLAB 2016b ............................... 18

3.9 A plot of the average error for each string for all the sample rates. Plotted using MathWorks MATLAB 2016b .............................. 19
List of Tables

2.1 Musical intervals relating to A using just intonation [12] . . . . . . . . . 4
2.2 The frequencies corresponding to the strings on the guitar based on A
440Hz ISO-standard [11]. Notice that the small e is two-octaves above E. 5
3.1 The target and initial frequencies used in the experiment. . . . . . . . . . 17
3.2 The average values of the errors for each sample rate. . . . . . . . . . . 17
Chapter 1

Introduction

1.1 Background

The guitar is an old traditional instrument. Historically it has been tuned by listening and comparing the frequency of the string to that of a tuning fork or similar reference with fixed frequency. With the advancements of electrical engineering, electronic tuners of different kinds have become widely available to consumers. Now there exist tuners in many variants. But most still rely on the user for the actual tuning and only display a reference if the string should be tightened or loosened. Some automatic solutions have surfaced but these are mostly mounted on the guitar like described in [9]. Instead this project explores development of a standalone solution that could be used by a guitar-tech or music teacher who needs to tune multiple guitars quickly and reliably.

1.2 Purpose

The purpose of this project is to research the possibility of implementing a pitch detection algorithm in conjunction with a motor and to explore the accuracy of different sampling rates to achieve the most reliable tuning of the different strings on a guitar.

1.3 Scope

This project covers a Bachelor’s thesis in Mechatronics at KTH. Since the work period will only be a semester, the main focus will be on frequency detection and motor control. The project was done using the hobby electronics microprocessor Arduino Uno [1]. The Arduino Uno is equipped with a ATmega328P microcontroller that doesn’t possess a digital signal processing periphery and therefore poses some constraints on the resolution of the input signal as well as the computational speed per cycle.
1.4 Method

An Arduino Uno was used for processing the signals and controlling the motor that turned the tuning pegs of the guitar. The signal from the guitar microphones was amplified with a DC voltage offset with the help of two op amps, a low-pass filter was also implemented to remove any background noise and aliasing of the signal.

Since the tuning pegs had to be able to turn in both directions, the motor needed an H-bridge driver so that the Arduino could control the direction of the current through it.

The design consist of two parts, a box that contains the Arduino, amplifier and driver. The other part is a drill handle for the motor that is connected to the box with a cable. The box was cutout with a laser cutter and the handle was 3D printed. A nozzle that was snapped onto the tuning pegs was 3D printed and attached to the rod of the motor.

PID-control was used for controlling the speed of the DC motor. The system was not modeled in order to find the PID-Parameters that gave the quickest and most stable step-response, instead the parameters were picked by trial and error in order to find a step-response that was ”good enough”.

YIN Autocorrelation method was used for detecting the frequency of the signal from the guitar microphones. Different sample rates with this method was examined and plots where accuracy was a function of the sample rate was made in order to find the sample rate that yields the most accurate tuning across all strings of the guitar.
Chapter 2

Theory

2.1 Sound Perception and Musical Scales

The way humans perceive sound and the systems used for instrument tuning are deeply related. When two frequencies are close to each other, but not equal, something called beating is experienced [6]. This is produced by the two frequencies amplifying each other at some points in time, and attenuating each other at other points, which can be seen in Figure 2.1. The frequency of the beats can be calculated using

\[ f_b = f_1 - f_2 \]  \hspace{1cm} (2.1)

where \( f_b \) is the beating frequency and \( f_1 \) and \( f_2 \) is the two close frequencies. Beating can occur in music when playing two different notes simultaneously as is often done.

Figure 2.1. (a) Displays two sine-waves with similar but not equal frequencies. (b) Displays the resulting beating. The figure was made with MathWorks MATLAB 2016b
CHAPTER 2. THEORY

<table>
<thead>
<tr>
<th>Interval</th>
<th>Note</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unison</td>
<td>A</td>
<td>1:1</td>
</tr>
<tr>
<td>Minor second</td>
<td>Bb</td>
<td>25:24</td>
</tr>
<tr>
<td>Major second</td>
<td>B</td>
<td>9:8</td>
</tr>
<tr>
<td>Minor third</td>
<td>C</td>
<td>6:5</td>
</tr>
<tr>
<td>Major third</td>
<td>C#</td>
<td>5:4</td>
</tr>
<tr>
<td>Perfect fourth</td>
<td>D</td>
<td>4:3</td>
</tr>
<tr>
<td>Augmented fourth</td>
<td>D#</td>
<td>45:32</td>
</tr>
<tr>
<td>Perfect fifth</td>
<td>E</td>
<td>3:2</td>
</tr>
<tr>
<td>Minor Sixth</td>
<td>F</td>
<td>8:5</td>
</tr>
<tr>
<td>Major Sixth</td>
<td>F#</td>
<td>5:3</td>
</tr>
<tr>
<td>Minor seventh</td>
<td>G</td>
<td>9:5</td>
</tr>
<tr>
<td>Major seventh</td>
<td>G#</td>
<td>15:8</td>
</tr>
<tr>
<td>Octave</td>
<td>A</td>
<td>2:1</td>
</tr>
</tbody>
</table>

Table 2.1. Musical intervals relating to A using just intonation [12]

Because of this the notes that sound the most pleasant are those where the fundamental frequencies of the notes are related by a rational ratio. This tuning system is called *just intonation* [6]. The musical interval an octave is defined by the ratio 2:1. Notes that are related by the interval an octave have the same name. For example both 110 Hz and 220 Hz is called an A. The notes in between are divided in to twelve intervals as can be seen in Table 2.1 However tuning an instrument with fixed notes, like a piano or guitar, using these ratios proves difficult. If notes are defined by a rational ratio back to a single frequency or key-center, the same notes will have slightly different frequencies when related to another frequency. The solution was to divide the octave into equal parts, making the distance from one interval to the next constant, one *semi-tone* [6]. Instead of using rational ratios the frequency $f_2$ of a note $n$ semi-tones away from another frequency $f_1$ is found with

$$f_2 = f_1 2^{n/12}$$

(2.2)

This method of tuning is called *equal temperament tuning* and is the method used on most contemporary instruments [6]. The frequency values corresponding to the pitches of the open strings on the guitar can be found in Table 2.2.

If the semi-tone is divided by 100 this is called a cent. Experiments have shown that the JND (Just noticeable difference) between a perceived note and the actual note of a melody, is 6 cents [6].

### 2.2 YIN Autocorrelation

There are several ways to acquire the frequency of a signal. It can be done either in the frequency-domain or in the time-domain. A common algorithm in the time-
2.2. YIN AUTOCORRELATION

<table>
<thead>
<tr>
<th>String</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>329.63Hz</td>
</tr>
<tr>
<td>B</td>
<td>246.94Hz</td>
</tr>
<tr>
<td>G</td>
<td>196.00Hz</td>
</tr>
<tr>
<td>D</td>
<td>146.83Hz</td>
</tr>
<tr>
<td>A</td>
<td>110.00Hz</td>
</tr>
<tr>
<td>E</td>
<td>82.41Hz</td>
</tr>
</tbody>
</table>

Table 2.2. The frequencies corresponding to the strings on the guitar based on A 440Hz ISO-standard [11]. Notice that the small e is two-octaves above E.

![Waveform and Autocorrelation](image)

Figure 2.2. (a) Example of waveform containing recording of the A-string of an acoustic guitar. (b) Autocorrelation function (ACF) of the guitar signal calculated using (2.3). The figure was made with MathWorks MATLAB 2016b.

The domain is the **AutoCorrelation Function** (ACF). It works by multiplying the signal with a time-delayed version of itself. For a discrete signal $x_t$ it can be defined as

$$r_1(\tau) = \sum_{j=t+1}^{t+W} x_j x_{j+\tau}$$  \hspace{1cm} (2.3)

where $r_1(\tau)$ is the ACF of lag $\tau$ at time index $t$, and the integration window size $W$. If $x_t$ is the signal shown in Figure 2.2(a) the resulting ACF is shown in Figure 2.2(b). This function has local maxima where the time-delay or lag $\tau$ is the period of the signal. If the period is found the frequency can be easily calculated if the sample frequency of the signal is known using
CHAPTER 2. THEORY

\[ frequency = \frac{\text{sample frequency}}{\text{period}} \quad (2.4) \]

A collection of steps to improve on the result of the normal ACF in musical applications was introduced in [5] under the name YIN Autocorrelation. If the first step of YIN Autocorrealtion is the normal ACF the second step is the introduction of a difference function

\[ d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2 \quad (2.5) \]

Here the period of the signal \( x_t \) is found by finding at what lag \( \tau \) \( d_t(\tau) \) is 0. The resulting function can be seen in Figure 2.3(a). The difference function may be expressed in terms of the ACF by expanding the squared sum into:

\[ d_t(\tau) = r_t(0) + r_{t+\tau}(0) - 2r_t(\tau) \quad (2.6) \]

The first two terms are energy terms. When they are constant, the difference function \( d_t(\tau) \) varies as the opposite of the ACF \( r_t(\tau) \) meaning searching for the minimum of the difference function would yield the same result as searching for the maximum of the ACF. However, since the second energy term varies with \( \tau \), the maxima of \( r_t(\tau) \) and the minima of \( d_t(\tau) \) may sometimes not coincide.

Since the difference function Figure 2.3(a) is zero when \( \tau \) is zero and often nonzero at the period due to imperfect periodicity, the algorithm chooses \( \tau \) at the zero-lag local minimum or dip, instead of the period dip, resulting in division by zero in (2.4) and the method fails. The solution proposed as step three in [5] is using the cumulative difference function in place of the normal difference function.

\[ d'_t(\tau) = \begin{cases} 1, & \text{if } \tau = 0, \\ d_t(\tau)/\left[ (1/\tau) \sum_{j=1}^{\tau} d_t(j) \right], & \text{otherwise.} \end{cases} \quad (2.7) \]

This function was obtained by dividing each value of the previous difference function by its average over shorter-lag values. The difference from \( d(\tau) \) is that it starts at 1 rather than 0, remains large at low lags, only dropping below 1 where \( d(\tau) \) falls below average Fig. 2.2(b). An additional benefit of this step is normalization of the function for the next step of the method.

The fourth step of [5] is to apply an absolute threshold to the to dip-selection process. It easily happens that the higher order dips of \( d'_t(\tau) \) Figure 2.3(b) is deeper than the period dip. If a lower dip is within the search range this is selected as the period and the calculated frequency is too low. By applying an absolute threshold we pick the smallest value of \( \tau \) that gives \( d'_t(\tau) \) a minimum below the threshold. If non is found then \( \tau \) corresponding to the smallest local minimum within the search range is chosen.
2.3. PID-CONTROL

Figure 2.3. (a) Difference function calculated using the example waveform from Fig. 2.3(a), (b) Cumulative mean difference function of the same example waveform. Here the function starts at 1 and remains high until the dip corresponding to the period of the signal. The figure was made with MathWorks MATLAB 2016b

Step five is using parabolic interpolation. The earlier steps only work as mentioned if the period is a multiple of the sampling period. Otherwise, the estimate of the period of the signal may be off by up to half the sampling period. This can be avoided by fitting a parabola over the minimum of \( d'_t(\tau) \) and its local neighbors. Then using the ordinate of the interpolated parabola in the dip-selection process. The abscissa of the selected minimum gets used as the period. The paper [5] finds that this result is slightly biased. To avoid this the abscissa of the corresponding minimum of the raw difference function \( d_t(\tau) \) is used.

2.3 PID-Control

If a system is supposed to compare an input signal to a desired reference value, the system will need to calculate the error between the input signal and reference value. The function of the error then becomes

\[
e(t) = r(t) - y(t)
\]  

(2.8)

where \( y(t) \) is the signal that the system is supposed to control and \( r(t) \) the desired value of that signal. The purpose of the controller is to minimize the error over time by constantly making small adjustments to the system, A PID-Controller is one of many options that can be applied to this problem.
According to [10] a PID-Controller is made up by three parts, a proportional (P) part, an integrating (I) part and a derivative (D) part. The P-controller is the simplest form of controller because it proportionally affects gain of the output signal. The function for the P-controller is

$$u(t) = K_P e(t) + u_0 \quad (2.9)$$

where $u(t)$ is the output signal of the controller, $K_P$ is the proportional gain and $u_0$ the output signal of the controller if the error is zero. An increase in proportional gain will reduce the rise time of the system but only to a certain degree. Once this threshold is reached an increase in gain will only increase the overshoot, overshoot is when the output signal exceeds the steady state value of the system.

To counter overshoot an integrating part can be added to the P-controller to get a PI-controller. The function of the PI-controller is

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau) d\tau + u_0 \quad (2.10)$$

where $K_I$ is the integral gain of the controller. With a PI-Controller the overshoot is reduced and any disturbance to the system as well as steady state error will be eliminated, this will however make the system more unstable if the proportional and integral gains are increased beyond a certain degree.

To counter instability there are two options, lower the gain of the P and I parts or adding a derivative part to the PI-controller and get a PID-controller. The resulting function for the controller is now

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau) d\tau + K_D \frac{d}{dt} e(t) \quad (2.11)$$

where $K_D$ is the derivative gain.
Chapter 3

Demonstrator

3.1 Problem formulation

The Robot tuner can be divided into two different parts; one part that amplifies the input signal and detects frequencies from the strings of the guitar and another part that controls the tension of the guitar strings. Figure 3.1 shows how the Robot looked when all the components had been assembled.

Figure 3.1. This picture shows how the system looked when it was completely assembled.

Figure 3.2 represents the system. The frequency is detected using the YIN...
Autocorrelation method that later can be processed by the PID-Control that will send a PWM signal to the driver which in turn will actuate the DC motor in order to adjust the tension of the guitar string.

Below are some of the criteria that the system had to fulfill in order to tune a guitar and answer the research question.

The criteria of the amplifier are:

- The amplifier needs to have at least a gain of 15 in order for the Arduino to reliably read the signals from the guitar microphones.

- The amplifier needs to add an 2.5 V DC-offset to the input signal so that the Arduino can read the complete waveform of the signal.

- A filter is needed in order to reduce background noise and remove any errors from aliasing.

The criteria for the motor are:

- The motor needs a gearbox with a gear ratio that is high enough to supply the shaft with enough torque to turn the tuning pegs of the guitar.

- The motor combined with the gearbox needs to be small enough to fit comfortably into the user’s hand.

The criteria of the frequency detection are:

- The frequency detection must produce a result within the JND of the frequency of the current string.

- The result must be produced fast enough to avoid excessive oscillation with the PID-controller.
3.1. PROBLEM FORMULATION

Figure 3.3. Flow-chart of the software implemented. The figure was made with draw.io
CHAPTER 3. DEMONSTRATOR

3.2 Hardware and Electronics

3.2.1 Amplifier

Because of the low voltage output of the microphones of the guitar and the arduino’s AD converters reading values between 0-5 V, the signal had to be amplified and DC-offset using a custom amplifier circuit. To remove any errors from aliasing of the signal and background noise a low-pass filter set to 1 kHz was implemented. The amp and filter circuit shown in figure 3.4 was provided by the lab assistant Philip Pulli.

Below is the list of components that were used for the amplifier in figure 3.4.

- 1 x OP amp INA126PA
- 1 x OP amp CA3160E
- 2 x 1 kΩ passive resistors
- 1 x 10 kΩ potentiometer
- 1 x 37 µF capacitor
- 1 x 10 kΩ passive resistor

The resistors used for the CA3160E were the two passive 1 kΩ resistors and for the $R_G$ the 10 kΩ potentiometer was used. The components that was used for the low-pass filter was the 37µF capacitor and the 10 kΩ passive resistor.
3.2. HARDWARE AND ELECTRONICS

Figure 3.5. This picture shows the amplifier component of the finished assembly.

The OP amp CA3160E together with the two 1 kΩ resistors were used to give the INA126PA an 2.5 V offset to the amplified signal. The potentiometer was used for controlling the gain of the output signal. All of the components for the amplifier soldered into a circuit can be seen in Figure 3.5.

3.2.2 H-bridge driver and Motor

To control the DC motor an H-bridge driver was needed. An H-bridge is a component that consist of 4 transistors that can direct the current trough the motor. This enables the Arduino to control the direction of motor rotation by applying a PWM signal to the inputs of the H-bridge. A copper plated board that was designed by the lab assistants Marcus Olsson and Fredrika Kringberg was used for the soldering of all the components of the H-bridge (see Figure 3.6).

Their design was an H-bridge that could simultaneously run two DC motors, but for this project only one of the motor ports was required. The H-bridge was supplied with 12 V and the required pins of the H-bridge was connected to the Arduino’s digital out ports. Different motors were tested in order to find one that had enough torque. A demo sample gearbox and DC motor from Maxon motor supplied by the lab assistants was the one that was picked for the project.

In order for the motor to be able turn the tuning pegs a nozzle was designed in Solid Edge ST7 [2] to fit over the tuning pegs and the shaft of the DC motor (see Figure 3.7). It was 3D printed using the Ultimaker 2 [14]. The blueprint with all the measurements for the nozzle can be found in Appendix A.
CHAPTER 3. DEMONSTRATOR

3.3 Software

3.3.1 Signal measurement

The signal was captured using the Arduinos built in AD converter. Using the Arduino IDE function `AnalogRead()` [4], the arduino captures a 10-bit value of the input signal. But the sample rate using this function is dependent on what other function calls there are in the code. Instead the audio was captured using interrupts, to ensure a fixed sample rate. This halts the process in the main loop to process the code in the interrupt function and then returns to where it stopped in the main loop. Two kinds of interrupts were used. First the sample rate of the ADC was set to the maximum value possible 38.5 kHz according to [7]. When the ADC finishes encoding the input signal from the guitar the current value is stored in the
3.3. SOFTWARE

variable `incomingAudio` which can be seen under `ISR(ADC_vect)` in Figure 3.3. The other interrupt used is a timer interrupt using the timer clock of the arduino [8]. This stores the value currently in `incomingAudio` at the current index of the byte array `rawData` with size 512 as can be seen under `ISR(TIMER0_COMPA_vect)` in Figure 3.3. Allowing for control of the resulting sample rate independently from the ADC. The method of storing the values in an array or buffer using a counter was inspired by [3]. However the implementation used for this project was done using interrupts instead of the main loop of the program to achieve more control of the sample rate as mentioned above.

3.3.2 Frequency processing

When the byte array `rawData` is full it is passed to the function `FreqCalc()`. Here the samples are first processed using the cumulative mean difference function mentioned in (2.6). This was implemented using nested for-loops as can be seen in Appendix C. The result is a value stored in the floating-point variable `dpt`, whose waveform is similar to the one in Fig. 2.3(b) for each iteration of the for-loop. The value of `dpt` is then passed to a peak-detection state-machine. Here the first step is comparing the current value to the previous. If the current value is less then the previous the derivate is negative and the values of the cumulative difference function are decreasing. If the current value is less than the absolute threshold mentioned in step 4 of section 2.2 the state-machine searches for a current value greater than the previous. If such a values is found the derivative is now positive and thus the abscissa of the previous value is corresponding to the period. A parabola is fit to the abscissa and it’s immediate neighbors and a new value of the period is calculated using

\[
\tau'_t = \frac{1}{2} \frac{\tau_t - (\tau_t - (\tau_{t-1}))^2(d(\tau_t) - d(\tau_{t+1})) - (\tau_t - (\tau_{t+1}))^2(d(\tau_t) - d(\tau_{t-1}))}{(\tau_t - (\tau_{t-1}))^2(d(\tau_t) - d(\tau_{t+1})) - (\tau_t - (\tau_{t+1}))(d(\tau_t) - d(\tau_{t-1}))}
\]

where \(\tau_t\) is the period lag found by the peak-detection and and \(\tau'_t\) is the period-lag after the parabolic interpolation. Note that the function value used is \(d(\tau_t)\) instead of \(d'(\tau_t)\) due to slight bias as explained in [5]. Finally using the interpolated period the fundamental frequency \(f_0\) of the signal is calculated using \(f_0 = \frac{\text{sample rate}}{\tau'_t}\). The result of the calculation is then passed to the PID-controller which can be seen under **Main program** in Figure 3.3.

3.3.3 PID-Control and driver

The function `PID_Control()` for the PID-Controller was implemented on the Arduino according to equation 2.11. The values of the proportional, integrating and derivative gains were set by trial and error rather than creating a model for the whole system and computing the most optimal values for the parameters.

The function takes the reference frequency corresponding to the string that is being tuned and compares it to the frequency that the microphones picks up
when the string is struck. The input frequency error is calculated by equation 2.8 and the output signal of the controller is calculated as earlier mentioned by equation 2.11. The function then maps the output signal of the PID-controller to a corresponding PWM value between 0-100 % using the built in function `map()` from the software package for the Arduino. The PWM value is then sent to a function called `motor_speed()` and if the PWM value is above 50 % it will map the PWM value to an 8 bit value and send it to a function called `clockwise()` which tells the motor to go clockwise at a speed given by the PWM signal.

The same procedure is made by the `motor_speed` function if the input PWM signal is below 50 % but will instead map and send the PWM signal to a function called `counter_clockwise()` which will turn the motor in a clockwise rotation with a given speed as can be seen in 3.3.

All these steps are repeated until the input frequency falls within the range of JND, then the rotation of the motor stops.

### 3.4 Results

The experiment that was made in order to figure out which sample rate that gives the most accurate tuning across all strings of the guitar was setup in the following way:

- A time limit of 60 seconds was set for the robot to tune one string.
- If the robot was idle for 5 seconds while tuning, the tuning was deemed complete.
- Each string was tuned down two semi-tones from its target frequency before each attempt.

The reason why a time constraint of 60 seconds was set was because if the robot had trouble adjusting the tension of a string to reach the target frequency it would have more time to get the tuning closer to JND. If the time ran out or if the motor was idle for more than 5 seconds, the frequency would be measured with an InfiniiVision DSO-X 2012A Oscilloscope [13] and the value would be put into notes.

The guitar was detuned in multiple ways and the system could handle them as well, but doing the experiment for different initial conditions as well would take too much time, something that was not within the scope of this project.

Tuning down each string two semitones beneath the target frequency ensured that each test was made with the same initial conditions. By having the same initial conditions for each attempt any deviations in the resulting data could be ruled out from being caused by different initial conditions for each experiment.

The reason why the strings were tuned down and not above the target frequency was to avoid putting the strings under too much stress during the experiment. It was desired that the system would turn the tuning pegs a few revolutions before being in tune, fewer that two semitones was thus not desirable.
3.4. RESULTS

<table>
<thead>
<tr>
<th>String</th>
<th>Target frequency</th>
<th>Initial frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>329.63Hz</td>
<td>293.67 Hz</td>
</tr>
<tr>
<td>B</td>
<td>246.94Hz</td>
<td>220.00 Hz</td>
</tr>
<tr>
<td>G</td>
<td>196.00Hz</td>
<td>174.62 Hz</td>
</tr>
<tr>
<td>D</td>
<td>146.83Hz</td>
<td>130.81 Hz</td>
</tr>
<tr>
<td>A</td>
<td>110.00Hz</td>
<td>98.00 Hz</td>
</tr>
<tr>
<td>E</td>
<td>82.41Hz</td>
<td>73.42 Hz</td>
</tr>
</tbody>
</table>

Table 3.1. The target and initial frequencies used in the experiment.

The initial frequencies were calculated using equation 2.2 and can be found in Table 3.1.

For the experiment six different sample rates between 10 kHz - 35 kHz were tested and for each sample rate all the strings would be tuned and measured 5 times to get an average frequency on each string for each sample rate.

The difference between a frequency $f_1$ and frequency $f_2$ in cents was calculated by substituting $c = n/100$ in (2.2) and solving for $c$.

$$c = 1200\log_2 \frac{f_2}{f_1} \quad (3.2)$$

An average of all of the errors measured in cents were then calculated in order to find the sample rate that gave the most accurate tuning.

The six sample-frequencies that were tested were 10 kHz, 12.5 kHz, 15.625 kHz, 19.231 kHz , 25 kHz and 31.25 kHz. The average values of the errors for each sample rate from the experiment can be seen in Table 3.2. Figure 3.8 shows a plot of the results from Table 3.2, values beneath the dashed line fall within the JND and the graph seems to have an oscillate around the JND. The average values for each string for all of the sample rates can be seen in Figure 3.9, most of the errors for the strings follow the same pattern as the plot in Figure 3.8. All of the average values for each sample rate were calculated from the measured data that can be found in Appendix B.

<table>
<thead>
<tr>
<th>Sample-frequency[Hz]</th>
<th>Average error[cents]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>4.17</td>
</tr>
<tr>
<td>12500</td>
<td>7.67</td>
</tr>
<tr>
<td>15625</td>
<td>3.80</td>
</tr>
<tr>
<td>19231</td>
<td>4.04</td>
</tr>
<tr>
<td>25000</td>
<td>9.07</td>
</tr>
<tr>
<td>31250</td>
<td>41.89</td>
</tr>
</tbody>
</table>

Table 3.2. The average values of the errors for each sample rate.
Figure 3.8. A plot of the average values of all errors for each sample rate. Plotted using MathWorks MATLAB 2016b
3.4. RESULTS

Figure 3.9. A plot of the average error for each string for all the sample rates. Plotted using MathWorks MATLAB 2016b.
Chapter 4

Discussion and conclusions

4.1 Discussion

Initially the results from the experiment show that, from the six sample rates there are three that make the total average tuning error fall within the JND. 10, 15,625 and 19,231 kHz. 15,625 kHz provides the smallest average tuning error as can be seen in Figure 3.8. However the difference from the other acceptable results is small. It’s worth noting that the result shows that the tuning error of at least one of the strings is above the JND limit for every sample rate as can be seen in Figure 3.9. Hence an optimal sample rate does not only depend on the total average being less then the JND but also that most used strings should be below the JND as well.

The results show that for lower sample rates the low-strings E-D have better precision than the higher strings G-e. This is most likely due to, lack of frequency resolution with the lower sample rate. The parabolic interpolation in step five in section 2.2 improves the result a little by providing an estimate of the period. But since the estimate is not perfect, the error in the PID could be nonzero even though the actual frequency of the string is correct, resulting in oscillation of the control signal.

When the sample rate is increased the higher strings get slightly more accurate. The biggest difference however, can be seen for the E and A strings. At 31,25 kHz the average error for these strings are larger than for the other strings, especially the E-string which had a average error at 222.6 cents. As can be seen in Appendix B the algorithm failed to tune the E-string. This can be explained by two reasons. Since a fixed buffer size of 512 samples was used, when the sample rate increases the resolution in time increases but the total size of the time window decreases. If this window is not big enough to contain the period information needed for YIN autocorrelation, the algorithm will provide a wrong frequency and the process will fail. Secondly, with the low frequency being described by more samples, more calculations had to be made to find the period of the signal. This results in a slow update of the PID error which in turn leads to the motor overshooting the target frequency before algorithm can find the current frequency. The system becomes
unstable and thus the method fails.

### 4.2 Conclusions

Finding a single optimal sample rate for an automated guitar tuner is a challenging task. Even though the sample rate of the processor plays a big role in the frequency calculation, for a reliable tune other facts must be considered and modeled, for example parameters of the PID, the responsiveness of the motor and the fit of the nozzle plays a big part in the resulting frequency of the strings. From the experiment performed, the sample rate 15,625 kHz was shown to provide the best result. The average total error is the smallest and when looking at the average error of the individual strings, only one is above the JND.
Chapter 5

Recommendations and Future work

5.1 Recommendations

To get as good results as possible, it should be ensured that the nozzle fits tightly to the tuning peg to avoid lag induced by lack of immediate transfer of movement. It should also be made sure that the motor has adequate torque to turn the tuning pegs.

5.2 Future work

The parameters of the PID-parameters needs to be optimized in order to get a more responsive controller and less oscillation in the control signal. More work could also be done to find the optimal PID parameters for each individual string. The sampling capabilities of the Arduino is limited, if possible it is recommended to use a ADC with higher bit-depth and sample rate in conjunction with more memory to provide a more accurate signal representation. A processor capable of fast floating point operations could also increase the accuracy of the PID control by getting more frequent values of the current frequency. If a faster motor that still provides the precision and torque needed could be found, the usage could be extended to application of new strings as well. An automatic audio leveling circuit could also be implemented to have consistent signal level for a longer period of time, avoiding estimation errors due to the transient of the guitar pick.
Bibliography


Appendix A

Nozzle blueprint

Snitt A-A

Detalj B

<table>
<thead>
<tr>
<th>Detaljnr</th>
<th>Antal</th>
<th>Benämning</th>
<th>Material</th>
<th>Antec/Dimension</th>
<th>Anmärkning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Snitt A-A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MF133X</td>
<td>Nozzle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Skapa av: Martin Gylling
Datum: 2017-04-27
Appendix B

Results of the experiment

Sample rate: 10000 Hz

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>A</th>
<th>D</th>
<th>G</th>
<th>B</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>82.71</td>
<td>110.77</td>
<td>145.58</td>
<td>196.4</td>
<td>245.94</td>
<td>327.61</td>
</tr>
<tr>
<td>Mean</td>
<td>85.1</td>
<td>110.01</td>
<td>146.28</td>
<td>195.97</td>
<td>245.85</td>
<td>338.71</td>
</tr>
<tr>
<td>Mean</td>
<td>82.82</td>
<td>109.49</td>
<td>145.55</td>
<td>198.18</td>
<td>248.66</td>
<td>329.75</td>
</tr>
<tr>
<td>Mean</td>
<td>82.61</td>
<td>110.17</td>
<td>147.6</td>
<td>196.27</td>
<td>247.65</td>
<td>335.12</td>
</tr>
<tr>
<td>Mean</td>
<td>82.47</td>
<td>109.66</td>
<td>146.89</td>
<td>196</td>
<td>245.06</td>
<td>326.75</td>
</tr>
<tr>
<td>Mean</td>
<td>82.82</td>
<td>110.02</td>
<td>145.86</td>
<td>196.56</td>
<td>247.072</td>
<td>331.584</td>
</tr>
<tr>
<td>Reference</td>
<td>82.41</td>
<td>110</td>
<td>145.83</td>
<td>196</td>
<td>245.94</td>
<td>329.63</td>
</tr>
<tr>
<td>Error</td>
<td>8,215445</td>
<td>0.514741</td>
<td>0.355686</td>
<td>4974556</td>
<td>0.925171</td>
<td>10.23221</td>
</tr>
<tr>
<td>Average error</td>
<td>4,169503</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sample rate: 12500 Hz

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>A</th>
<th>D</th>
<th>G</th>
<th>B</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>82.18</td>
<td>105.41</td>
<td>145.69</td>
<td>199.2</td>
<td>247.46</td>
<td>324.6</td>
</tr>
<tr>
<td>Mean</td>
<td>85.1</td>
<td>110.45</td>
<td>146.76</td>
<td>195.79</td>
<td>250.15</td>
<td>327.16</td>
</tr>
<tr>
<td>Mean</td>
<td>82.5</td>
<td>110.84</td>
<td>145.68</td>
<td>196.12</td>
<td>245.65</td>
<td>323.42</td>
</tr>
<tr>
<td>Mean</td>
<td>82.47</td>
<td>108.55</td>
<td>147.6</td>
<td>199.48</td>
<td>248.45</td>
<td>326.55</td>
</tr>
<tr>
<td>Mean</td>
<td>82.38</td>
<td>110.23</td>
<td>146.67</td>
<td>194.85</td>
<td>250.13</td>
<td>330.75</td>
</tr>
<tr>
<td>Mean</td>
<td>82.52</td>
<td>110.18</td>
<td>147.2</td>
<td>198.99</td>
<td>248.564</td>
<td>326.452</td>
</tr>
<tr>
<td>Reference</td>
<td>82.41</td>
<td>110</td>
<td>148.83</td>
<td>195</td>
<td>245.94</td>
<td>329.63</td>
</tr>
<tr>
<td>Error</td>
<td>2,302922</td>
<td>2,484897</td>
<td>4,357086</td>
<td>8,712488</td>
<td>11,34818</td>
<td>16,772</td>
</tr>
<tr>
<td>Average error</td>
<td>7,655552</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sample rate: 15265 Hz

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>A</th>
<th>D</th>
<th>G</th>
<th>B</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>82.67</td>
<td>110.61</td>
<td>145.46</td>
<td>195.28</td>
<td>245.27</td>
<td>328.56</td>
</tr>
<tr>
<td>Mean</td>
<td>85.44</td>
<td>108.55</td>
<td>145.24</td>
<td>197.88</td>
<td>247.35</td>
<td>328.56</td>
</tr>
<tr>
<td>Mean</td>
<td>81.67</td>
<td>108.84</td>
<td>145.5</td>
<td>195.9</td>
<td>247.4</td>
<td>337.7</td>
</tr>
<tr>
<td>Mean</td>
<td>82.66</td>
<td>110.01</td>
<td>147.71</td>
<td>196.22</td>
<td>248.9</td>
<td>338.5</td>
</tr>
<tr>
<td>Mean</td>
<td>81.88</td>
<td>110.77</td>
<td>145.78</td>
<td>195.63</td>
<td>246.84</td>
<td>330.91</td>
</tr>
<tr>
<td>Mean</td>
<td>82.234</td>
<td>110.176</td>
<td>147.558</td>
<td>196.182</td>
<td>247.232</td>
<td>328.846</td>
</tr>
<tr>
<td>Reference</td>
<td>82.41</td>
<td>110</td>
<td>145.63</td>
<td>196</td>
<td>245.94</td>
<td>329.63</td>
</tr>
<tr>
<td>Error</td>
<td>3,911825</td>
<td>2,767761</td>
<td>8,327783</td>
<td>1,606829</td>
<td>2,045928</td>
<td>4,122314</td>
</tr>
<tr>
<td>Average error</td>
<td>5,797106</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## APPENDIX B. RESULTS OF THE EXPERIMENT

<table>
<thead>
<tr>
<th>Sample rate: 19231 Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E</strong></td>
</tr>
<tr>
<td>Attempt1</td>
</tr>
<tr>
<td>Attempt2</td>
</tr>
<tr>
<td>Attempt3</td>
</tr>
<tr>
<td>Attempt4</td>
</tr>
<tr>
<td>Attempt5</td>
</tr>
<tr>
<td><strong>Average</strong></td>
</tr>
<tr>
<td><strong>Reference</strong></td>
</tr>
<tr>
<td><strong>Error</strong></td>
</tr>
<tr>
<td><strong>Average error</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample rate: 25000 Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E</strong></td>
</tr>
<tr>
<td>Attempt1</td>
</tr>
<tr>
<td>Attempt2</td>
</tr>
<tr>
<td>Attempt3</td>
</tr>
<tr>
<td>Attempt4</td>
</tr>
<tr>
<td>Attempt5</td>
</tr>
<tr>
<td><strong>Average</strong></td>
</tr>
<tr>
<td><strong>Reference</strong></td>
</tr>
<tr>
<td><strong>Average error</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample rate: 31250 Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E</strong></td>
</tr>
<tr>
<td>Attempt1</td>
</tr>
<tr>
<td>Attempt2</td>
</tr>
<tr>
<td>Attempt3</td>
</tr>
<tr>
<td>Attempt4</td>
</tr>
<tr>
<td>Attempt5</td>
</tr>
<tr>
<td><strong>Average</strong></td>
</tr>
<tr>
<td><strong>Reference</strong></td>
</tr>
<tr>
<td><strong>Error</strong></td>
</tr>
<tr>
<td><strong>Average error</strong></td>
</tr>
</tbody>
</table>
Appendix C

Code

Listing C.1. Main program

```c
/*
 * School: KTH Royal Institute of Technology
 * Course: MF133X Degree Project in Mechatronics, First Cycle
 * Made by: Martin Gylling and Ruben Svensson
 * finalized: 2017-05-19
 * TRITA: MMK 2017:24 MDAB 642
 * This code is used to capture an audio signal from an electric guitar,
 * store it in a buffer,
 * calculate the period and the frequency of the signal,
 * compare the frequency to a reference frequency of the corresponding
 * string,
 * actuate a DC-motor to tune the guitar string.
 */

#define LENGTH 512

// Sample Frequency in kHz
const float sample_freq = 15625;

//Guitar Tuning table
const float stringFreqs[6] = {82.41, 110, 146.83, 196, 246.94, 329.63};

//The largest expected period of a string
int stringMaxPeriod[6];

//index of current string
byte currentString = 0;

//Variables for the frequency detection part
byte incomingAudio; //Holds the current value from the ADC
byte clipLight = 13; //Pin connected to the clip light
byte clipping = 0; //1 if clipping, else 0
byte rawData[LENGTH]; //Array storing the data from the ADC

int count; //Counter variable for the sampling process
int len = sizeof(rawData); //The length of the audioBuffer

int tau, j; //Variables for the sums and finding the lag
```
APPENDIX C. CODE

```c
long r, rold, rt, rtau, dt, dtold, dtold2, dj; //Different function variables
int thresh = 0; //Dynamic threshold of when to output frequency
float freq_per, freq_old, freq_old2, filtered_freq, dpt, dold; //Floats to store frequency and sum data
byte pd_state = 0; //Peak-detection state-machine variable

//PID Parameters and Variables
float pid_output = 0;
float old_output;
float error, error_sum;
float Kp = 1;
float Ki = 10;
float Kd = 0;
//JND in frequency for the different strings
float tol[6] = {0.29, 0.38, 0.51, 0.68, 0.8573, 1.1444};
float pid_range[6] = {25, 30, 30, 40, 60, 80};

float pwm_output;

//Motor Pins and Variables
int enA = 10;
int in1 = 9;
int in2 = 8;

//Variables for master state-machine
int button;
static const byte switchPin = B00001000;
byte masterState, wasPressed;

void setup() { //Initialize pins and variables
  analogRead(A0);
  Serial.begin(9600);
  pinMode(switchPin, INPUT);
  pinMode(enA, OUTPUT);
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(clipLight, OUTPUT);
  for (int i = 0; i < 8; i++) {
    pinMode(i, OUTPUT);
  }
  masterState = 0;
  wasPressed = 0;
  count = 0;

  //Calculate the largest expected period for each string
  for (int i = 0; i < 6; i++) {
    stringMaxPeriod[i] = (int) sample_freq / (stringFreqs[i] / 2);
  }
  cli(); //disable interrupts
  }
//set up continuous sampling of analog pin 0

//clear ADCSRA and ADCSRB registers
ADCSRA = 0;
ADCSRB = 0;

ADMUX |= (1 << REFS0); //set reference voltage
ADMUX |= (1 << ADLAR); //left align the ADC value- so we can read
                      //highest 8 bits from ADCH register only

ADCSRA |= (1 << ADPS2) | (1 << ADPS0); //set ADC
       //clock with 32 prescaler- 16mHz/32=500kHz
ADCSRA |= (1 << ADATE); //enable auto trigger
ADCSRA |= (1 << ADIE); //enable interrupts when measurement complete
ADCSRA |= (1 << ADEN); //enable ADC
ADCSRA |= (1 << ADSC); //start ADC measurements

//set timer0 interrupt at 2kHz
TCCR0A = 0; //set entire TCCR0A register to 0
TCCR0B = 0; //same for TCCR0B
TCNT0 = 0; //initialize counter value to 0
//set compare match register for 10khz increments
OCR0A = 15; //= (16*10^6) / (10000*64) - 1 (must be <256)
//turn on CTC mode
TCCR0A |= (1 << WGM01);
//Set CS01 and CS00 bits for 64 prescaler
TCCR0B |= (1 << CS01) | (1 << CS00);
//enable timer compare interrupt
TIMSK0 |= (1 << OCIE0A);
sei(); //enable interrupts

ISR(ADC_vect) {//when new ADC value ready
incomingAudio = ADCH; //store current ADC value
//If signal is 5V or 0V set clip light else turn of clip light
if(incomingAudio >= 255 || incomingAudio <= 0){
  clipping = 1;
}
else if (incomingAudio > 10 && incomingAudio < 245){
  clipping = 0;
}
}

ISR(TIMER0_COMPA_vect) {//at timer interval
//While rawData is not full, put value of incomingAudio and increase
//the index
if (count < LENGTH) {
  rawData[count] = incomingAudio;
  count++;
}
//Check if the switch was pressed
button = PINB;
button = button & switchPin;

switchPressed(switchPin);
}

// If switchPin was pressed and released switch to next masterState
void switchPressed(byte switchPin){
    if (button == switchPin){
        wasPressed = 1;
    }

    if (wasPressed == 1 && button == 0){
        wasPressed = 0;
        if(masterState < 7){
            masterState++;
        }
        else{
            masterState = 1;
        }
    }
}

// Estimates the abscissa using the estimated period value and its immediate neighbours
float ParaIntrp(int c, float fa, float fb, float fc){
    int a = c-2;
    int b = c-1;
    float x;

    x = (float)((b - (b-a)*(fb-fc)-(b-c)*(b-c)*(fb-fa))/(2*(b-a)*(b-a) *(fb-fc)-(b-c)*(fb-fa)));
    return x;
}

// Calculates the frequency of the input signal with YIN Autocorrelation and peak-detection state-machine
void FreqCalc(){
    if (count >= LENGTH) {
        dt = 0;
        dtold = 0;
        dtold2 = 0;
        dj = 0;
        dpt = 0;
        r = 0;
        rt = 0;
        rtau = 0;
        pd_state = 0;
        float period = 0;
        int period_old = 0;
        float current_lowest = 100;

        for(tau=0; tau < len; tau++){
            // YIN-Autocorrelation
dtold2 = dtold;
dtold = dt;

            // YIN-Autocorrelation
            //...
dold = dpt;
rold = 0;
r = 0;
dt = 0;
dpt = 0;
//Serial.println(rawData[tau]);
for(j=0; j < len/2; j++){
    if(j+tau >= len){
        r = 0;
        rt = (rawData[j]-128)*(rawData[j]-128)/256;
        rtau = 0;
    } else {
        r = (rawData[j]-128)*(rawData[j+tau]-128)/256;
        rt = (rawData[j]-128)*(rawData[j]-128)/256;
        rtau = (rawData[j+tau]-128)*(rawData[j+tau]-128)/256;
    }
    dt += rt + rtau -2*r;
    rold += r;
}
dj += dt;
if(tau == 0){
    dpt = 1;
} else {
    dpt = (float)(dt*tau)/dj;
}
// Peak Detect State Machine
if (pd_state == 2 && dold < dpt && dpt < 0.18)
{
    period_old = tau-1;
    period = ParaIntrp(tau, dt, dtold, dtold2);
    pd_state = 3;
    break;
}
else if (pd_state == 2 && dold < dpt && dold < current_lowest - 0.1)
{
    current_lowest = dold;
    period_old = tau-1;
    period = ParaIntrp(tau, dt, dtold, dtold2);
    pd_state = 1;
}
if (pd_state == 1 && dold > dpt){
    pd_state = 2;
}
if(tau > stringMaxPeriod[currentString]){ 
    break;
}
if (!tau) {
    thresh = rold * 0.5;
}
pd_state = 1;
}

// Frequency identified in Hz
if (thresh > 2) {
  if (period != 0) {
    freq_old2 = freq_old;
    freq_old = filtered_freq;
    freq_per = sample_freq/period;
    filtered_freq = 0.8*freq_per+0.1*freq_old+0.1*freq_old2;
    if (filtered_freq > sample_freq/(stringMaxPeriod[currentString]/4)) {
      filtered_freq = freq_old;
      freq_per = freq_old;
    }
  }
  PID_Control(stringFreqs[currentString], freq_per);
}

count = 0;


void PID_Control(float stringFreq, float inputFreq) {
  old_output = error;
  error = stringFreq - inputFreq;
  error_sum += error/sample_freq;
  if (error < tol[currentString] && error > -tol[currentString]) {
    error = 0;
  }

  pid_output = Kp*error + Ki*error_sum + Kd * (error-old_output) * sample_freq/1000;
  if (pid_output < -pid_range[currentString]) {
    pid_output = -pid_range[currentString];
  } else if (pid_output > pid_range[currentString]) {
    pid_output = pid_range[currentString];
  }
  pwm_output = map(pid_output, -pid_range[currentString], pid_range[currentString], 0, 100);

  motor_speed(pwm_output);
}

void motor_speed(float pwm) {
  if (pwm <= 50) {
    counter_clockwise(map(pwm, 0, 50, 0, 255));
  } else if (pwm > 50) {
    counter_clockwise(map(pwm, 50, 100, 0, 255));
  }
void counter_clockwise(int PWM) {
    digitalWrite(in2, LOW);
    digitalWrite(in1, HIGH);
    analogWrite(enA, (255 - PWM));
}

void clockwise(int PWM) {
    digitalWrite(in2, HIGH);
    digitalWrite(in1, LOW);
    analogWrite(enA, PWM);
}

void loop() {
    if (clipping) {
        digitalWrite(clipLight, HIGH);
    } else {
        digitalWrite(clipLight, LOW);
    }
    if (masterState > 1) {
        FreqCalc();
    }
    switch (masterState) {
    case 0:
        // initial state
        pwm_output = 50;
        PORTD = B11111100;
        delay(5 * sample_freq);
        masterState = 1;
        break;
    case 1:
        // idle state
        pwm_output = 50;
        motor_speed(pwm_output);
        error_sum = 0;
        PORTD = B00000000;
        break;
    case 2:
        // Low E-string
        currentString = 0;
        PORTD = B10000000;
        break;
    case 3:
        // A-string
        currentString = 1;
        PORTD = B01000000;
        break;
    case 4:
        // D-string
        currentString = 2;
        break;
    default:
        break;
    }
}
APPENDIX C. CODE

```c
PORTD = B00100000;
break;
case 5:
    //G-string
    currentString = 3;
    PORTD = B00010000;
    break;
case 6:
    //B-string
    currentString = 4;
    PORTD = B00001000;
    break;
case 7:
    //High E-string
    currentString = 5;
    PORTD = B00000100;
    break;
```