



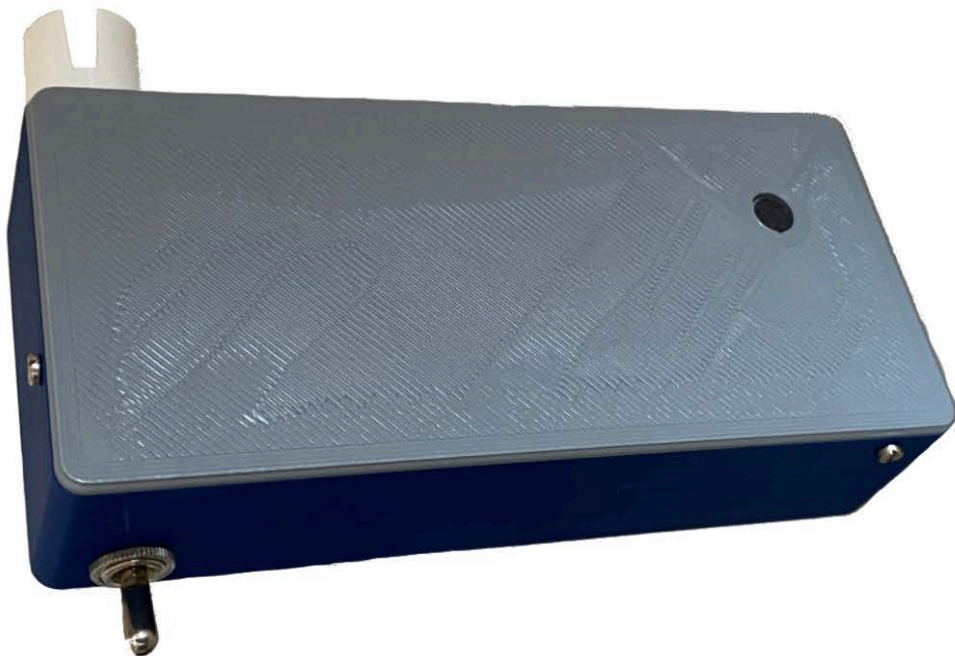
DEGREE PROJECT IN MECHANICAL ENGINEERING,  
FIRST CYCLE, 15 CREDITS  
*STOCKHOLM, SWEDEN 2021*

# Pitcher

An automatic guitar tuner

**HANNES ANDERSSON**

**JOHN SJÖBERG**



**KTH ROYAL INSTITUTE OF TECHNOLOGY  
SCHOOL OF INDUSTRIAL ENGINEERING AND MANAGEMENT**





# Pitcher

An automatic guitar tuner

HANNES ANDERSSON  
JOHN SJÖBERG

Bachelor's Thesis at ITM  
Supervisor: Nihad Subasic  
Examiner: Nihad Subasic

TRITA-ITM-EX 2021:26



# Abstract

Pitcher is a prototype which makes it easier for inexperienced guitar players to tune their guitars without any prior knowledge required. This thesis will explore how the construction varies between the usage of DC and a stepper motor, how reliable the tuner is and how long it takes to tune the guitar.

The tuner will capture sound with a microphone and calculate the current frequency of the string with YIN autocorrelation. Based on the frequency a control system regulator is used to determine the speed and direction of a motor which turns the tuning peg, this is repeated until the string is in tune. 30 tests were conducted from different starting frequencies, and the time it took for the tuner to find the right pitch and the string's corresponding frequency was measured. Some of the measurements were a couple of  $Hz$  off pitch, and only about half of the frequencies measured belonged to the interval where there is no noticeable difference of the pitch, therefore the tuner could not be considered reliable. The time it takes to tune the guitar is dependent on how far off pitch the string is and the difference in time does not depend linearly with the starting frequency, it increases faster the further off pitch the string is.

The tuner is portable and to apply the tuner to the guitar it is held and placed on the tuning peg with one hand as the other hand is plucking the string.

**Key Words:** Arduino, mechatronics, autocorrelation, guitar tuner, pitch detection algorithm

# Referat

## Automatisk gitarrstämmare

Den automatiska gitarrstämman, Pitcher, är en prototyp som möjliggör för oerfarna gitarranvändare utan förkunskaper att stämma en gitarr. Den här avhandlingen kommer att undersöka hur konstruktionen skiljer sig åt vid användning utav en stegmotor respektive en likströmsmotor, hur lång tid det tar att stämma gitarren samt hur tillförlitlig prototypen är.

Stämman avläser ljudsignaler med en mikrofon och beräknar sedan frekvensen av strängen med hjälp av YIN autokorrelation. Den beräknade frekvensen behandlas i en regulator som avgör vilken hastighet och i vilken riktning motorn ska rotera stämskruven. Detta repeteras tills korrekt frekvens erhålls. 30 test gjordes då gitarren stämdes från olika startfrekvenser där tiden att stämma strängen respektive dess frekvens mättes. Några mätningar hade en frekvens som avvek flera  $Hz$  från korrekt frekvens, och cirka hälften av frekvenserna från alla mätningar tillhörde frekvensintervallet där ingen skillnad kan höras på tonen, därför kan gitarrstämman ej anses vara tillförlitlig. Tiden det tar att stämma en sträng är beroende på hur ostämmd den är och skillnaden i tid beror inte linjärt av startfrekvens, utan den ökar snabbare desto mer ostämmd gitarren är.

Stämman är portabel och för att applicera den på gitarren placeras munstycket på stämskruven medan den andra handen slår an strängen.

**Nyckelord:** Arduino, mekatronik, autokorrelation, gitarrstämman, tonavläsning

# Acknowledgements

We would like to thank and acknowledge:

Amir Avdic and Staffan Qvarnström for helping us along the way by supplying us with components, advice and answers to our many questions.

Janne from Industrial Productions for advice and access to their workshop.

Nihad Subasic for knowledge from lectures and guidance.

Our fellow course participants, especially Albin Boestad and Fabian Rudberg, for inspiration, tips and feedback.

*Hannes Andersson, John Sjöberg  
Stockholm, May 2021*





# Nomenclature

This thesis consists of code segments where variables from the code will be displayed in another style, as `variable_name`. Text displayed as `function_name()`, ending with two parenthesis, represents the function `function_name`. The specific function `main()` refers to the code's function `loop()`.

## List of Abbreviations

<b>AC</b>	Alternating Current
<b>ACF</b>	Autocorrelation Function.
<b>DC</b>	Direct Current
<b>IDE</b>	Integrated Development Environment
<b>JND</b>	Just Noticeable Difference
<b>OP-AMP</b>	Operational Amplifier
<b>PDA</b>	Pitch Detection Algorithm
<b>PID</b>	Proportional-Integral-Derivative
<b>PLA</b>	Polylactic Acid
<b>PWM</b>	Pulse With Modulation
<b>RPM</b>	Revolutions Per Minute
<b>USB</b>	Universal Serial Bus

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Purpose . . . . .	1
1.3	Scope . . . . .	2
1.4	Method . . . . .	2
<b>2</b>	<b>Theory</b>	<b>3</b>
2.1	Tone and Guitar Tuning . . . . .	3
2.2	Pitch Detection Algorithm . . . . .	4
2.3	Microphone . . . . .	6
2.4	Arduino Uno Microcontroller . . . . .	6
2.5	H-bridge . . . . .	7
2.6	DC Motor . . . . .	8
2.7	Stepper Motor . . . . .	8
2.8	PID Controller . . . . .	9
<b>3</b>	<b>Demonstrator</b>	<b>11</b>
3.1	Mechanical Design . . . . .	11
3.1.1	Body Design . . . . .	11
3.1.2	Nozzle Design . . . . .	12
3.1.3	Electret Microphone . . . . .	13
3.1.4	Motor and Driver . . . . .	13
3.2	Electrical Design . . . . .	14
3.2.1	Signal Processor . . . . .	14
3.3	Programming . . . . .	15
3.3.1	Stepper Motor . . . . .	16
3.3.2	DC Motor . . . . .	16
<b>4</b>	<b>Result</b>	<b>19</b>
<b>5</b>	<b>Discussion</b>	<b>21</b>
5.1	Reliability . . . . .	21
5.2	Tuning Time . . . . .	21
5.3	Motor Type . . . . .	22

5.4	Conclusion . . . . .	23
5.5	Future Developments . . . . .	23
	<b>Bibliography</b>	<b>25</b>
	<b>Appendices</b>	<b>26</b>
<b>A</b>	<b>Technical Drawings</b>	<b>A1</b>
A.1	This is the technical drawing of the Nozzle . . . . .	A1
A.2	This is the technical drawing of the case . . . . .	A2
A.3	This is the technical drawing of the lid . . . . .	A2
<b>B</b>	<b>Data Sheet for DC Motor</b>	<b>B1</b>
<b>C</b>	<b>Arduino code</b>	<b>C1</b>
C.1	This code is used for the DC motor . . . . .	C1
C.2	This code is used for the stepper motor . . . . .	C7
<b>D</b>	<b>Acumen Code</b>	<b>D1</b>

# List of Figures

2.1	Example of speech waveform and its calculated ACF . . . . .	5
2.2	Difference function and mean normalized difference function . . . . .	5
2.3	Arduino Uno . . . . .	7
2.4	Schematic figure of an H-bridge . . . . .	7
2.5	Embryo of a DC motor . . . . .	8
2.6	Inside view of a stepper motor . . . . .	9
3.1	Finished construction of the tuner . . . . .	11
3.2	Construction of case and lid . . . . .	12
3.3	Construction of the nozzle . . . . .	12
3.4	Microphone . . . . .	13
3.5	Soldered circuit . . . . .	14
3.6	Schematic diagram of the signal processor . . . . .	15
3.7	Flowchart of interrupt functions . . . . .	17
3.8	Flowchart of main functions . . . . .	18
4.1	Guitar tuner . . . . .	19
4.2	Graph of tuning time . . . . .	20

# List of Tables

2.1	Standard tuning of a six-stringed guitar . . . . .	4
4.1	Testing data from different starting frequencies . . . . .	20



# Chapter 1

## Introduction

### 1.1 Background

One of the most important requirements for a musical instrument like a guitar is being able to perform the right note when needed. Tuning makes this possible but it is far from easy to tune a guitar for someone who does not have the experience. A guitar is tuned by screwing on the tuning pegs on the head of the guitar which tightens or loosens the strings until the desired tone is reached. There are various kinds of guitar tuners, for example electric which tells whether the tone is too high or low when a string is being played but there are also tuning forks which only give the right tone to tune after. The latter of those require a lot of experience from the player, and the electrical one still requires the player to make manual adjustments which can lead to unintended deviation from the desired pitch. An automatic guitar tuner which sets the desired tone when a string is being played would simplify this process, and it would also make it possible for inexperienced players to tune their guitars without effort.

### 1.2 Purpose

This project is about making a prototype of a portable automatic guitar tuner that can tune a guitar according to the user's preferences with acceptable reliability. The purpose is to make it possible for everyone to tune a guitar despite experience level. This report will examine these following questions,

- *How reliable is the tuner?*
- *How long does it take for the tuner to get the right pitch?*
- *What are the differences in using a stepper motor or a DC motor?*

### 1.3 Scope

The goal of this report is to create a device that with the help of a microphone, can read the frequency of a string, and from that data, rotate a connected motor until the frequency of the string matches the desired frequency. The prototype will only be able to tune one predetermined string as a proof of concept, and it will be constructed so that the person using it must hold the device on their own.

### 1.4 Method

The concept and methods used for the guitar tuner was influenced by a bachelor's thesis written by *Svensson* and *Gylling* who built a guitar tuner for an electric guitar [1]. The guitar tuner in this project consist of two systems, and an Arduino Uno was used as microcontroller to control these two systems. The first system converts the incoming analog signal to digital and from there calculates the frequency of the signal. The signal was detected by an electret microphone which passed its value to the Arduino where the frequency was calculated with a method called YIN autocorrelation, this is the same method that *Svensson* and *Gylling* used in their bachelor's thesis to calculate the frequency of an incoming analog signal.

The second system is the motor control, it consists of an electric motor which rotates the tuning peg based on the calculated frequency, and a H-bridge since the motor has to be able to rotate in both directions depending on whether the string is too tight or too loose. The motor controlling system will have an implemented PID-controller to shorten the time for the tuner to find the correct frequency.

The work will be divided into milestones, where the first one is to get the signal processor to functionally work as desired, the second one will be to add a motor to the circuit. The third and last one is to design and create a case which can fit all the required components.



## Chapter 2

# Theory

### 2.1 Tone and Guitar Tuning

Sound is perceived in the ear as vibrations, and a repeating vibration is what we call a pitch. For example, a common pitch is A440 which vibrates 440 times every second, in other words, it has the frequency of 440 Hz [2].

The fundamental parts of a standard guitar generally consist of six strings, of different thickness, that are being held over a fretboard between a bridge in one end, and over a nut and connected to different tuning pegs in the other. When a string is plucked it vibrates with a certain frequency which mainly depends on three things: the mass, the length and the tension on the string. Heavier strings move more slowly which in turn lowers the frequency, hence different strings on the guitar have different characteristics. The length of the string is manipulated by the player from pressing down on the fretboard, instead of being the distance between the nut and the bridge the string now has the length of the distance between the next fret and the bridge instead, and a shorter string generates a higher frequency. Adjusting the tension of the string is done by turning the tuning pegs, higher tension gives a higher frequency, and this is generally how a guitar is tuned [3]. When tuning a guitar, there must be some interval of frequencies that determine whether the string is tuned or not since it is hard to find the exact frequency. To determine this, the JND can be used as tolerance which is the interval of a frequency where there is no difference between the perceived note and the actual note. The JND varies for different notes and can be expressed in frequencies [1].

On fretted string instruments, such as the guitar, equal tempered scale is used. It is built on the idea that every octave consists of 12 equally large intervals, which allows the instrument to play all fixed tones in every key at the cost of making every interval fall out of perfect tune [4].

The standard tuning for a six string guitar is based on A440, the ISO standard for the tuning of  $A_4$ , this is presented in Table 2.1 [5].

**Table 2.1.** Standard tuning of a six-stringed guitar where "e" represents the higher e-string.

String	Frequency
e	392,63 Hz
B	246,94 Hz
G	196,00 Hz
D	146,84 Hz
A	110,00 Hz
E	82,41 Hz

## 2.2 Pitch Detection Algorithm

Pitch detection algorithms are used when calculating the pitch from a digital signal. The pitch of a digital signal is either determined in the time-domain or the frequency-domain. The autocorrelation function (ACF) is a commonly used algorithm in the time-domain. The ACF of a discrete signal  $x_t$  is defined as

$$r_t(\tau) = \sum_{j=t+1}^{t+W} x_j x_{j+\tau} \quad (2.1)$$

where  $r_t$  is the autocorrelation function of lag  $\tau$  at time index  $t$ , and the window size of the integration  $W$ . This function for the signal in Figure 2.1.(a) is illustrated in Figure 2.1(b). The local maximum in Figure 2.1(b) is where the lag  $\tau$  is the period of the signal.

A method for pitch detection in musical applications is introduced in [6] and is called YIN autocorrelation, it consists of several steps that improve the ACF. The first step in the YIN-method is the ACF, the second step is to introduce a difference function

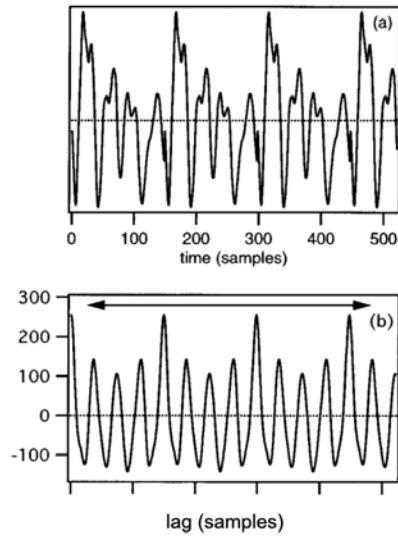
$$d_t(\tau) = \sum_{j=1}^W (x_j - x_{j+\tau})^2 \quad (2.2)$$

where the period of the signal  $x_t$  is determined by searching for which values of  $\tau$  the function is equal to zero. There are infinite sets of such values due to the periodicity of the function as can be seen in Figure 2(a). The squared sum in Equation 2.2 can be expressed in the terms of the ACF as

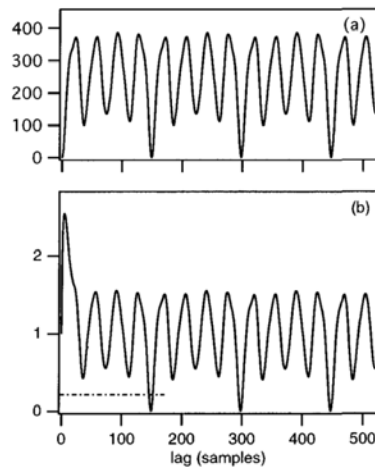
$$d_t(\tau) = r_t(0) + r_{t+\tau}(0) - 2r_t(\tau) \quad (2.3)$$

where the two first terms are energy terms. As those terms are constant, the difference function  $d_t(\tau)$  varies as the opposite of  $r_t(\tau)$  when searching for a maximum of one or the minimum of the other one the result would be the same. The second energy term also varies with  $\tau$  leading to that the minima of  $d_t(\tau)$  and the maxima of  $r_t(\tau)$  may not always coincide. The third step of [6] is to introduce a cumulative

## 2.2. PITCH DETECTION ALGORITHM



**Figure 2.1.** (a) Example of a speech waveform. (b) The ACF calculated from the waveform in (a) [6].



**Figure 2.2.** (a) Difference function calculated for the signal in Figure 2.1.(a). (b) Cumulative mean normalized difference function to the same signal as in (a) [6].

mean normalized difference function. The difference function of Figure 2.2.(a) is zero when the lag is zero and often it is nonzero at the function period whereas the periodicity is imperfect. The algorithm will choose the zero-dip lag instead of the period lag if the search range does not have a lower boundary, this will fail the algorithm and therefore the following *cumulative mean normalized difference*

function is introduced

$$d'_t(\tau) = \begin{cases} 1, & \text{if } \tau = 0, \\ d_t(\tau) / \left[ (1/\tau) \sum_{j=1}^{\tau} d_t(j) \right] & \text{otherwise.} \end{cases} \quad (2.4)$$

The function  $d'_t(\tau)$  is obtained by dividing each value of the old difference equation  $d_t(\tau)$  with its average over shorter-lag values. Seen in Figure 2.2.(b) the function starts in 1 comparing to  $d_t(\tau)$  which starts in 0. Other differences is that  $d'_t(\tau)$  tends to remain large at low lags, and drops below 1 only where  $d_t(\tau)$  falls below the average. According to [6] this reduces “too high” errors and to reduce the “too low” errors, an absolute threshold is introduced in step 4.

When one of the higher-order dips of  $d'_t(\tau)$  is deeper than the period dip a subharmonic error occurs called “octave error” and the ACF may choose a high-order peak. If an absolute threshold is set then the smallest value of  $\tau$  is chosen such that the minimum of  $d'_t(\tau)$  is deeper than that threshold. If no such minimum is found, then the global minimum is chosen instead.

Previous steps are based on the fact that the period is a multiple of the sampling period. If that is not the case, the earlier estimations may be inaccurate by up to half the sampling period. This can be solved in step 5 in [6] by using a parabolic interpolation on the function  $d'_t(\tau)$  over the minimum and its nearest neighbours. The ordinate of the interpolated minimum is used in the selection process for the dip and the abscissa of the minimum is used as a period estimate. To avoid that the abscissa is biased the corresponding minimum of  $d_t(\tau)$  is being used.

## 2.3 Microphone

In general a microphone is a converter between acoustic energy and electrical energy. One of the main uses for a microphone is as a measuring instrument where acoustic signals are converted into electrical currents which can be processed and interpreted. Inside an electret microphone one of the key components is a diaphragm. When sound waves affect the diaphragm the distance between two plates of a capacitor inside the microphone gets altered, which in turn affects the voltage. One advantage with an electret microphone is that the microphone itself isn't dependent on an external source of power [7].

## 2.4 Arduino Uno Microcontroller

Arduino Uno, which is shown in Figure 2.3, is an open-source microcontroller produced by the company Arduino. The microcontroller is equipped with 14 digital input/output pins and six analog input pins, as well as a USB-connector and a 32 kilobyte flash memory. The operating voltage is 5 V, and the recommended input to power the Arduino Uno is 7 – 12 V. Programming the microcontroller is done in a C-based language in the Arduino IDE software from where it is transferred to the Arduino Uno via the USB.

## 2.5. H-BRIDGE

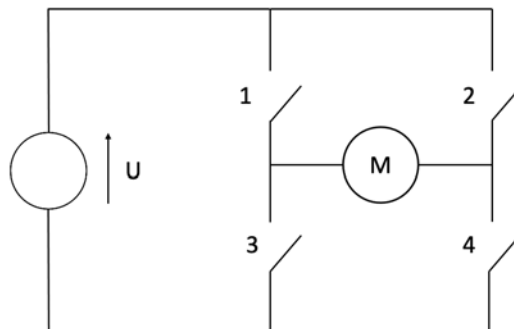
The Arduino can read analog values between  $0-5\text{ V}$  with the function `analogRead()` but it can not output analog values between that voltage range. To output voltages between  $0-5\text{ V}$  the Arduino uses the function `digitalWrite()` which outputs a  $5\text{ V}$  signal that is turned on and off in different time intervals, this generates a steady analog output value. The time between the on and off switch determines the analog value and by modulating it, the output value can be changed. This kind of signal is called PWM signal [8].



**Figure 2.3.** Picture of an Arduino Uno [8].

## 2.5 H-bridge

A common way of controlling electric driven motors is by using an H-bridge, these allow the motors to rotate both forwards and backwards while still using direct current. This is done by the design of having four switches which can change between being turned on and off, for example, if switch 2 and 3 in Figure 2.4 is on and 1 and 4 are turned off the current will flow from right to left through the motor in the figure. Doing the opposite will cause the current to flow from left to right which makes the motor rotate in the opposite direction. A regular safety feature on H-bridges is that they don't allow all four switches to be on at the same time since this will cause the voltage source to short circuit [9].



**Figure 2.4.** A simple schematic figure of an H-bridge made in *Microsoft PowerPoint*.

## 2.6 DC Motor

A DC motor is an electrical motor which transforms energy from a direct current into mechanical energy. In principle, the stator of the motor creates a magnetic field inside, and by running the current  $I$  through a loop inside the rotor according to Figure 2.5, a force pair occurs which cause a rotation. The loop itself is connected to a commutator that allows the current through the loop to change direction every half turn, this allows the force pair to have the same direction for every full turn. For an actual DC motor, the number of loops is increased. The direction that the motor turns depend on the direction of the current through the circuit, this means that the direction can be changed by the use of an H-bridge. The speed of the rotor can be adjusted by changing the voltage of the power source [9]. Another way of reducing the speed of the motor is by fitting it with a gearbox which reduces the speed of the shaft while increasing the torque of the motor.

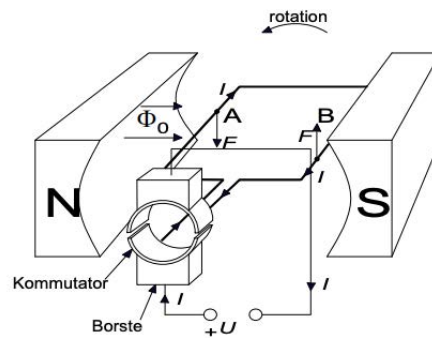


Figure 2.5. Embryo of a DC motor [9].

## 2.7 Stepper Motor

A stepper motor is very similar to a DC motor except that it rotates in small distinct steps. The rotation is caused by several coils that are distributed evenly around the rotor which can be seen in Figure 2.6, these coils are grouped up in what is known as phases. The coils in each phase are energized simultaneously, and by sequentially activating each phase, the motor can rotate in precise steps. They also have their highest amount of torque at lower speeds. For these reasons, stepping motors are commonly used where precision in both speed and positioning is needed. The resolution of each step is determined by the amount of steps per revolution, which usually ranges between 4-400. In order to drive what is known as a bipolar stepper motor a driver with two full H-bridges is required to alternate the polarity of the different phases [10].

## 2.8. PID CONTROLLER

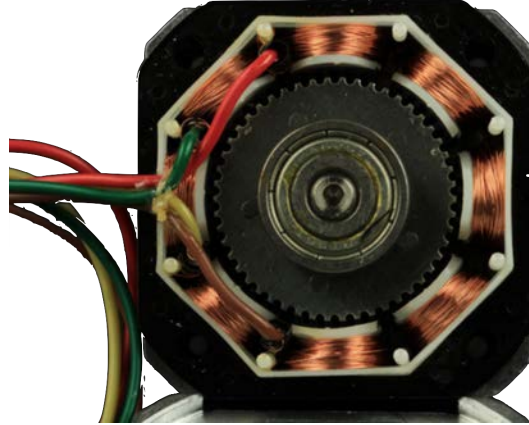


Figure 2.6. Inside view of a stepper motor [10].

## 2.8 PID Controller

The PID controller is a commonly used loop mechanism used in industrial control systems. The controller consists of three terms, the proportional, integral and derivative, which together creates the PID. The PID controller continuously calculates the error value  $e(t)$  as

$$e(t) = r(t) - y(t) \quad (2.5)$$

where  $r(t)$  is the desired value of the signal and  $y(t)$  is the measured value of the signal. The controller strives to minimize the error  $e(t)$  by calculating an input value  $u(t)$  to the system as

$$u(t) = K_p e(t) + K_i \int_0^t e(t') dt' + K_d \frac{d}{dt} e(t) \quad (2.6)$$

where  $K_p$ ,  $K_i$  and  $K_d$  respectively denotes the gain coefficients for the proportional, integral and derivative term [11].





## Chapter 3

# Demonstrator

The design of the guitar tuner can be broken down into three subgroups. The mechanical, the electrical and the programming. This chapter will in detail present the design of these three parts.



**Figure 3.1.** The finished construction with all components mounted.

### 3.1 Mechanical Design

#### 3.1.1 Body Design

The body of the prototype works as a case for the components. Since the concept of the tuner demands the player to pluck the string as the guitar is being tuned, it was

important that it can be held with one hand. This resulted in the semi-compact prototype seen in Figure 3.2 which was big enough to hold the necessary components without being too bulky. The bottom part of the case holds the majority of the components such as the Arduino Uno, the circuit, the motor, the batteries and a power switch. The holes on the side of the case allowed both the motor and the power switch to be screwed on to their respective location. The motor is held by two M3 screws and the power switch was threaded which allowed it to be fitted using a nut. The rest of the components were fitted with adhesive tape to hold them in place. The microphone was attached to the lid of the case to have it facing the body of the guitar while the tuner is being used. The case and the lid are held together by M3 screws, and both parts were designed in *Solid edge ST10* and later 3D printed in PLA plastic, technical drawings are presented in Appendix A.2, A.3.

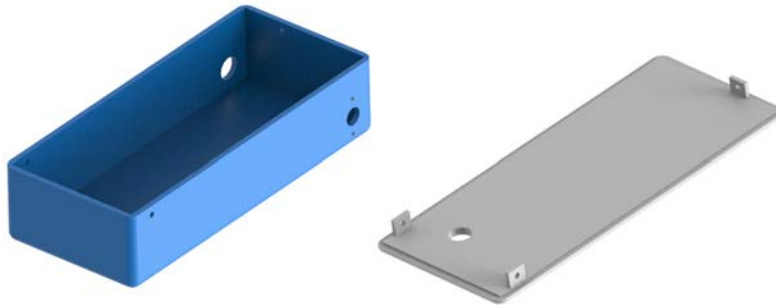


Figure 3.2. Image of the case and lid rendered in *Keyshot 10*

### 3.1.2 Nozzle Design

The nozzle consists of two crossed cuts where the tuning pegs can be placed. The cuts are tapered so that tuning pegs of different sizes will be able to fit. In order to mount it on the motor shaft, a semicircle shaped hole that matched the cross section of the shaft was made. The flat surface of the semicircle transfers the torque from the motor to the nozzle. The design was made in *Solid edge ST10* and then 3D-printed in PLA plastic, a technical drawing is presented in Appendix A.1.



Figure 3.3. Image of the nozzle rendered in *Keyshot 10*

### 3.1. MECHANICAL DESIGN

#### 3.1.3 Electret Microphone

The microphone used was an electret microphone with an integrated amplifier from the company Adafruit of the model MAX4466. The microphone itself has the capability to interpret frequencies ranging between  $20 - 20000 \text{ Hz}$  and the amplifier has an adjustable gain which can increase the original signal with a factor of 25 to 125. When using the microphone together with an Arduino it was recommended that the  $3,3 \text{ V}$  output was used in order to power the amplification since it had the lowest impact on the signal [12].



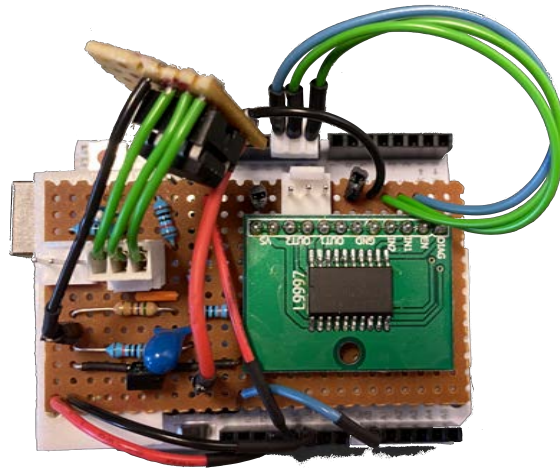
**Figure 3.4.** Picture of Adafruit MAX4466 [12].

#### 3.1.4 Motor and Driver

The two motors used were a two-phase stepper motor from Tamagawa Seiki and a DC motor L149.6.392 from Micromotors which were provided by the lab assistant Amir Avdic. The stepper motor has a resolution of 200 steps per revolution. It was used due to its high torque at low speed since the pegs will rotate at low speed, and therefore the motor had to have enough torque. The stepper motor also provide precise positioning which can be an advantage when adjusting the pegs for minimal error. The driver for the stepper motor that was used was a DRV8825 stepper motor controller by Texas Instruments. It can operate between  $8,2 - 45 \text{ V}$  and handle upwards to  $2,5 \text{ A}$  of continuous current with proper heat sinking. The DC motor has a supply voltage of  $6 \text{ V}$ , a torque of  $0,2 \text{ Nm}$  and it has a rotational speed of  $5 \text{ rpm}$ , more information of the motor can be found in Appendix B. This low rpm is important when turning the pegs with a DC motor so that the pegs will not turn too far between the samples. The high torque is also important as mentioned earlier since the tuning pegs can be tough. To control the direction of the DC motor a L9997 motor driver is used. To run the DC motor, a PWM signal is sent to the motor and for different values of the PWM signal, the motor will rotate at different rpm's. The direction is determined by sending high or low signals to specified input ports on the driver. The torque of the DC motor will decrease when lowering the value of the PWM signal.

## 3.2 Electrical Design

The electrical design consists of a signal processor and a motor driver. The circuit was first developed on a breadboard until the circuit was fully working, later the finished circuit was soldered on to a solder plate. The signal processor had the same design for both DC and stepper motor which the driver later was connected to. The electrical design is powered by one 9 V battery and four AA 1,5 V batteries.



**Figure 3.5.** The soldered DC circuit on a copper board mounted on the Arduino.

### 3.2.1 Signal Processor

The design of the signal processor was based on a project by *Amanda Ghassaei* where an input configuration for an analog signal was presented. *Ghassaei* stated the following three necessary modifications to the circuit to make the microphone's output signal compatible with an Arduino [13].

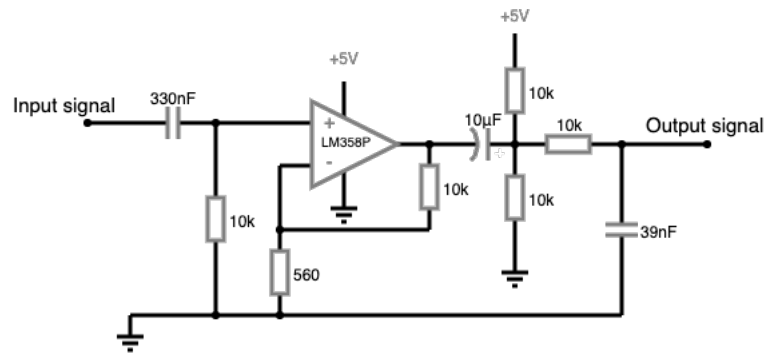
- Frequency cutoff-filter
- Voltage amplifier
- DC-offset

The signal processor detects the input sound with an electret Adafruit Max4466 microphone and outputs an analog signal to the Arduino. The filter was used to cut off unwanted frequencies since the microphone detects frequencies up to 20 kHz and the frequencies for the guitar strings, as presented in Table 2.1, spans between 82,4–392,63 Hz. The amplifier was vital since the output signal of the microphone had an amplitude of order 100 mV, the data in this low voltage signal is harder for

### 3.3. PROGRAMMING

the Arduino to read and therefore the signal was amplified to an amplitude around  $5\text{ V}$ . An active band-pass filter was used consisting of an amplifier, a low-pass and a high-pass filter stage. The components used for the active band-pass filter was the OP amplifier LM358P from Texas Instruments, one  $560\ \Omega$  and one  $10\text{ k}\Omega$  resistor for the amplifying part. For the filtering part two  $10\text{ k}\Omega$  resistors, a  $330\text{ nF}$  capacitor for the high-pass filter stage, and a  $39\text{ nF}$  capacitor for the low-pass filter stage, this is illustrated in Figure 3.6. This leads to the signal being amplified to an amplitude of  $5\text{ V}$  and only consisting of frequencies between  $48 - 408\text{ Hz}$ .

The last modification is the DC-offset of the signal. The output signal from the microphone oscillates around  $0\text{ V}$  and the Arduino's analog input's only detect signals between  $0 - 5\text{ V}$  which results in signals lower than zero being left out, these are necessary when calculating the frequency and therefore a  $2,5\text{ V}$  offset is used. The DC offset works by adding  $2,5\text{ V}$  to the output signal which makes the frequency oscillate around  $2,5\text{ V}$  instead of zero. When the signal is filtered, amplified and offset the signal processor sends the fully readable signal to the Arduino. The components used for the offset where two  $10\text{ k}\Omega$  resistors and one  $10\ \mu\text{F}$  capacitor. By wiring the two resistors in series between the  $5\text{ V}$  output from the Arduino and ground, the voltage in the junction between the resistors was divided into  $2,5\text{ V}$  due to equal resistance on the resistors. The junction is then wired to the op-amp's output via the capacitor to only pass the AC component of the signal. After these steps the output signal of the microphone was ready to be used in the Arduino, the signal processing circuit is illustrated in Figure 3.6.



**Figure 3.6.** Circuit of the active band-pass filter and the DC-offset which creates the signal processing unit. This circuit was made in *Falstad Circuit Simulator*.

### 3.3 Programming

This section goes through the programming part of the two concepts for the guitar tuner. The implemented code is based on the code from *Gylling* and *Svensson's* guitar tuner thesis since they used the same PDA as this tuner. Due to differences

in the interface the code was modified to make it applicable for the tuner in this project.

When buffering data in `main()`, in this case, sampling data into an array, the time for how long the buffering takes depend on how much and what type of code `main()` consists of. It can vary from time to time and therefore the time between the samplings can get inconsistent. In this case when sampling data an interrupt is used to create a fixed sampling rate to make sure that the time between the samplings is the same every time. According to *Ghassaei* the sample rate should be set to  $38,5\text{ kHz}$  to make the code as efficient as possible, this sample rate gives an interval between the samplings of  $26\ \mu\text{s}$ .

The interrupt functions interrupts the main function every  $26\ \mu\text{s}$  to execute its own block of code, and when it is done the code will return to where it was in the main function. The interrupt functions in this code reads and store the current value of the variable `incomingAudio` to the buffer which generates an array `rawData` consisting of data that later is used in the main function.

The two different motor concepts uses the same signal processing method which implies that the two interrupt functions and the majority of the `main()` stays the same for both concepts. An overall flowchart of the interrupt functions is presented in Figure 3.7 and the flowchart for the two different `main()` functions is presented in Figure 3.8. Each code starts by setting up the variables, interrupts and timer used in the code.

### 3.3.1 Stepper Motor

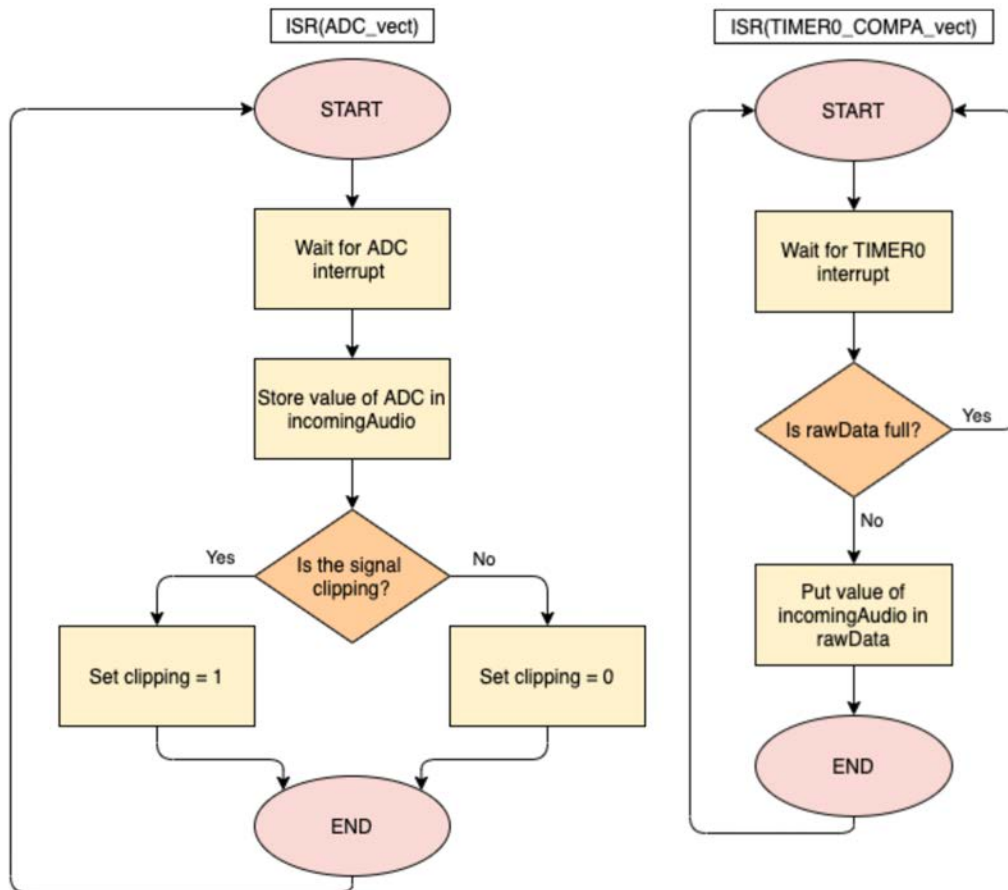
The code starts by checking if `rawData` has collected enough data points, if that is the case, `FreqCalc()` is called which calculates the frequency. If a new frequency is successfully calculated, `rawData` is cleared and the frequency is sent to `calc_error()` which determines the number of steps and the rotation speed for the motor. To determine these variables, an if-else statement is used which consists of three statements. The statements represents an individual frequency interval which executes if the error belongs to one of the intervals. The different statements sets a predetermined value to the motor variables, and `run_motor` executes with the given variables. If `FreqCalc()` does not calculate a new frequency, the code will not clear the `rawData` and the code will then be executed with the old frequency.

### 3.3.2 DC Motor

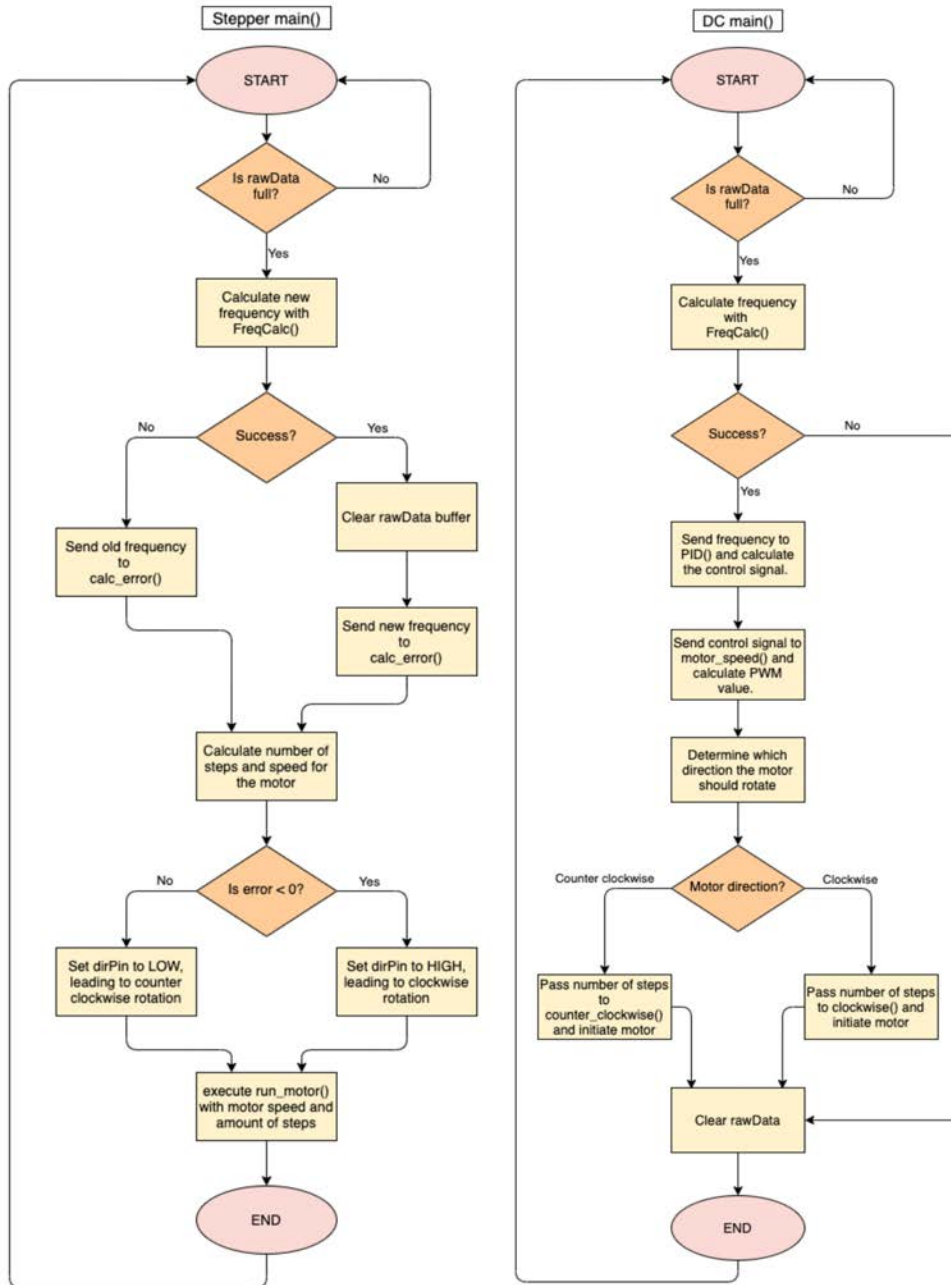
As in the other code, it starts to check if `rawData` is full and if that is the case, `FreqCalc()` is called. If the frequency is unsuccessfully calculated, `rawData` is cleared and `main()` starts over. Otherwise the frequency is passed to PID which calculates the control signal based on the error in Equation 2.5. The coefficients of the PID controller in Equation 2.6 are taken from *Gylling* and *Svensson's* thesis. The control signal is then passed on to `motor_speed()` which determines the PWM value. Then the motor direction is determined and the calculated PWM value is

### 3.3. PROGRAMMING

passed to `clockwise()` respectively `counter_clockwise()` depending on what has been decided. As the last step, `rawData` is cleared and can therefore start sampling new data for new calculations. When a frequency lies within the interval of JND which for the A-string is  $0,38\text{ Hz}$ , the motor stops for one second and the string is considered tuned.



**Figure 3.7.** Flowchart of the two implemented interrupt functions for both concepts. This flowchart was made with *app.diagrams.net*.



**Figure 3.8.** Flowchart of the implemented main functions for the two concepts. This flowchart was made with *app.diagrams.net*.



## Chapter 4

# Result

The finished prototype that was described in Chapter 3 can be seen in Figure 4.1, where it is presented as a portable hand held device. It is used simply by placing the nozzle on the tuning peg, flipping on the power switch and strumming the string until the motor stops. The motor used in the prototype was a DC motor.



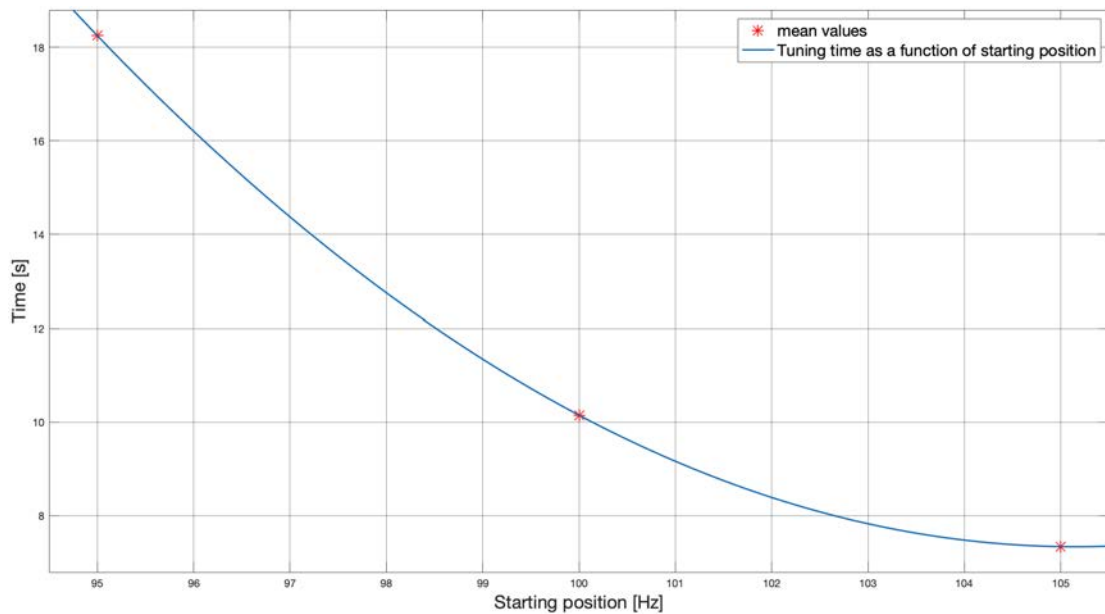
**Figure 4.1.** The finished concept of the guitar tuner.

In order to evaluate the tuner, a series of tests were conducted. The A-string on a guitar was deliberately set out of tune to a certain frequency with the help of the mobile application *n-Track Tuner* available on *App Store*. Then the automatic tuner was used to tune the guitar to the correct frequency of  $110\text{ Hz}$ , and when the tuner deemed the guitar to be in tune, the frequency was once again measured with *n-Track Tuner*. The different starting positions were  $95, 100$  and  $105\text{ Hz}$ , each starting position was tested ten times, and every test conducted was timed. The results are presented in Table 4.1.

**Table 4.1.** Testing data from different starting frequencies

	95 Hz		100 Hz		105 Hz	
	Time (s)	Frequency (Hz)	Time (s)	Frequency (Hz)	Time (s)	Frequency (Hz)
1	14,51	110,1	10,25	109,7	4,56	110,4
2	23,06	110,2	9,23	110,0	14,5	109,3
3	23,72	108,9	11,46	108,4	5,02	110,4
4	12,75	109,6	11,96	108,7	4,42	109,3
5	11,8	110,0	9,65	110,4	5,48	110,3
6	18,2	110,0	10,15	109,4	7,34	110,0
7	16,9	110,6	10,26	110,7	5,53	110,3
8	26,4	110,8	9,95	106,8	16,33	110,0
9	24,07	109,9	8,23	110,2	5,58	110,4
10	11,03	110,3	10,35	109,4	4,65	109,9
mean	18,24	110,04	10,15	109,37	7,34	110,03

Furthermore, to get an idea of how the tuning time depended on how far off pitch the guitar was when the tuning started, the time was plotted as a function of the starting position in Figure 4.2. It was done by fitting a curve to the mean values of the tuning time in Table 4.1.

**Figure 4.2.** A graphical illustration of the tuning time made in *MATLAB*.

## Chapter 5

# Discussion

### 5.1 Reliability

To determine reliability, the obtained frequencies will be compared with the JND for the A-string, meaning frequencies in the interval of 109,62–110,38  $Hz$  is considered correct. Looking at data from Table 4.1 the starting frequencies of 95 and 105  $Hz$  have mean frequencies that is inside the JND interval, but for the starting frequency of 100  $Hz$  it lies just outside. The reason for that could be because it has a few deviating result and that test number 8 has a frequency of 106,8  $Hz$ , which is far away from the desired one. This deviation could depend on disturbance in the sampling data, resulting in a miscalculated frequency that happened to be inside the tolerance, this may question the reliability. Out of the 30 measurements made, 14 were inside the JND interval and 11 missed out with less than 0,32  $Hz$  of the interval. Even though other measurements came close to the target frequency of 110  $Hz$ , the criteria of the JND was not met for the majority of the testing. The JND interval however, consists of the frequencies where the pitch is interpreted as the correct one, which could argue the point that the 11 measurements within 0,32  $Hz$  of the interval are closer to the correct pitch than what the data in the table shows. Other aspects on the tuner's trustworthiness could be the reliability of the application *n-Track Tuner* which read the test results. This will not be taken into consideration since before testing, it was compared with another frequency detector and had identical result.

### 5.2 Tuning Time

When examining the testing data presented in the previous chapter a couple of things come to mind. From Figure 4.2 the fact, that the further away from the correct frequency you start, the longer it takes to get in tune, can be stated. Secondly, a conclusion can be drawn that the time it takes to tune a string isn't linearly dependent on how far off tune the guitar is to begin with. For example, if one would examine the fitted curve in Figure 4.2, a drastic decrease of about two seconds in

time can be seen when comparing starting positions 95 *Hz* to 96 *Hz*. In comparison if the same difference in frequency is viewed at a starting position closer to the wanted frequency of 110 *Hz*, the drop in tuning time is far less significant. This would make sense given that for most cases some fine tuning was required as the correct pitch got close.

Looking at the test results in Table 4.1 there is a clear lack of consistency in the 95 *Hz* segment with times ranging from 11,03 *s* to 24,07 *s*. This could be explained by the fact that if the sampled frequency would be calculated wrongly and declare a frequency higher than 110 *Hz*, the motor could rotate in the wrong direction. Given that an electret microphone was used to sample data to the pitch detecting algorithm, there is always a chance that unwanted frequencies from outside noise could corrupt the ones that were given by the guitar, despite the use of a filter. Furthermore, the tests performed from 100 *Hz* and 105 *Hz* proved to be far more consistent in the time aspect with only a couple of significant outliers in the 105 *Hz* segment.

### 5.3 Motor Type

The constructions with the two different motor types are very similar when looking at the electrical circuit. They use the same signal processor and they both are connected to the circuit via a motor driver, so switching between the two types can easily be implemented by only changing the code. The two motors used in this project has the same order of size and can both fit into the case.

The biggest problem with the stepper motor in this project though was the fact that the nominal torque of the stepper motor wasn't enough to rotate the tuning peg. To improve the torque a new stepper motor must be used, which increases the size and weight. The stronger stepper motor will therefore not fit into the current case and the new design would be bulky and not easily managed by the user.

The behaviour of a DC and stepper motor is very different. The stepper motor will based on the calculated frequency rotate a fixed amount of steps at a certain speed and then it will wait for a new frequency to be calculated. The stepper motor is very precise and can rotate few steps at a low speed which should give precise tuning. The problem with this implementation on a guitar tuner though is that there is no way of knowing how many steps the peg should turn, and therefore the stepper motor can get stuck and rotate around the ideal state if the amount of steps is too high. The risk of this can be minimized by defining very fine step intervals in the code but that takes a lot of testing. The implementation of the stepper motor in this project could not be tested since it was too weak and therefore not much time was spent on improving this.

The DC motor takes a PWM signal as input and rotates at a certain speed until a new frequency is calculated and another signal can be sent to the motor. This implementation is good since it will continuously rotate and lower the speed as it gets closer to the right frequency and when it is in the right tune, the guitar will stop

## 5.4. CONCLUSION

rotating the motor, and if the peg exceeds the correct position it will simply change direction to find the right one. The DC motors torque decrease as the PWM value lowers, this gives a smaller range of the motor's speed since to low PWM signals will not give enough torque to rotate the tuning peg. Therefore the motor has to have a low initial rpm, otherwise the motor will rotate too fast and the tuning peg will exceed its desired position, and the DC motor will behave the same as the stepper motor and oscillate around the ideal state without being able to stay at the right tune. The DC motor in this project has a low rpm and therefore this problem will not occur.

## 5.4 Conclusion

The best choice of motor for the guitar tuner is the DC motor. With its advantage in torque related to size it is the best fit for a compact and user friendly design. Also, the fact that the stepper motor rotates a fixed amount of steps is a disadvantage and the DC motor is once again the more favourable choice. The time it takes for the guitar tuner to tune the string varies between every try, in some instances a lot, and it is dependent on how far off the pitch is. For these reasons, there is no way to answer exactly how long it takes to tune an arbitrary string. However it can be stated with some certainty that it is dependent on how far off pitch it is to begin with. What can also be said is that a longer tuning time does not affect the result of the final frequency, which is more important than if the tuner was fast and had an off pitch result. What can be said about the reliability of the tuner is that 25 out of the 30 measurements, i.e. 83,3%, were within 0,32 Hz from the JND interval. However the tuner is not deemed reliable because of the significant outliers, and since only 14 measurements were inside the JND interval.

## 5.5 Future Developments

Since this project only built a prototype, there are many areas that could be developed and improved in future work. The fact that this tuner only adjusts the A-string could be expanded so that it could tune every string with its standard tuning. To be suitable for more experienced users, the guitar could also have manual input of frequency from the user, for that a display and a menu may need to be implemented. Since the frequency calculating algorithm is already done, this kind of improvements does not lead to major changes in the existing construction since all the electrical circuits and parts will stay the same. What could be developed in that aspect is the size of the case and make its shape more suitable for a hand, to do that, the batteries used could be improved since for now, the batteries take a lot of space in the case. To improve the reliability of the tuner, the PID coefficients and the sample rate could be researched and developed since it was taken from previous work and may not have been the best option for the guitar used in this project.



# Bibliography

- [1] Ruben Svensson and Martin Gylling. “Robotic Electric Guitar Tuner”. DEGREE PROJECT IN TECHNOLOGY, FIRST CYCLE. Royal Institute of Technology, May 2017. URL: <https://www.diva-portal.org/smash/get/diva2:1200615/FULLTEXT01.pdf> (visited on 05/18/2021).
- [2] Kyle Gann. *The Arithmetic of Listening: Tuning Theory and History for the Impractical Musician*. University of Illinois Press, 2019. ISBN: 9780252042584. URL: <http://www.jstor.org/stable/10.5406/j.ctvpj7j1r> (visited on 04/07/2021).
- [3] P.U.P.A Gilbert. *Chapter 13 - Vibration of strings*. Ed. by P.U.P.A Gilbert. Third Edition. Academic Press, 2022, pp. 229–244. ISBN: 978-0-12-824347-3. URL: <https://www.sciencedirect.com/science/article/pii/B9780128243473000133> (visited on 04/05/2021).
- [4] J. William Gannon and Rex A. Weyler. *Just intonation tuning*. U.S. Patent: 5 501 130. Mar. 1996.
- [5] ISO Central Secretary. *Acoustics — Standard tuning frequency (Standard musical pitch)*. en. Standard ISO 16:1975. Geneva, CH: International Organization for Standardization, 1975. URL: <https://www.iso.org/standard/3601.html> (visited on 04/05/2021).
- [6] Alain de Cheveigne and Hideki Kawahara. “YIN, a fundamental frequency estimator for speech and music”. Apr. 2002.
- [7] Tim J. Mellow and Leo L. Beranek. *Acoustics: Sound Fields and Transducers*. Elsevier Inc, 2012. ISBN: 978-0-12-391421-7. URL: <https://www.sciencedirect.com/book/9780123914217/acoustics-sound-fields-and-transducers#book-info> (visited on 04/06/2021).
- [8] Arduino.cc. *Arduino Uno*. 2021. URL: <https://store.arduino.cc/arduino-uno-rev3> (visited on 04/24/2021).
- [9] Hans Johansson. *Elektroteknik*. Institutionen för maskinkonstruktion, 2013.
- [10] Bill Earl. *All About Stepper Motors*. Institutionen för maskinkonstruktion, 2015.
- [11] Torkel Glad and Lennart Ljung. *Reglerteknik : grundläggande teori*. (4., [omarb.] uppl.) Studentlitteratur AB, 2006. ISBN: 91-44-02275-1inb.

## BIBLIOGRAPHY

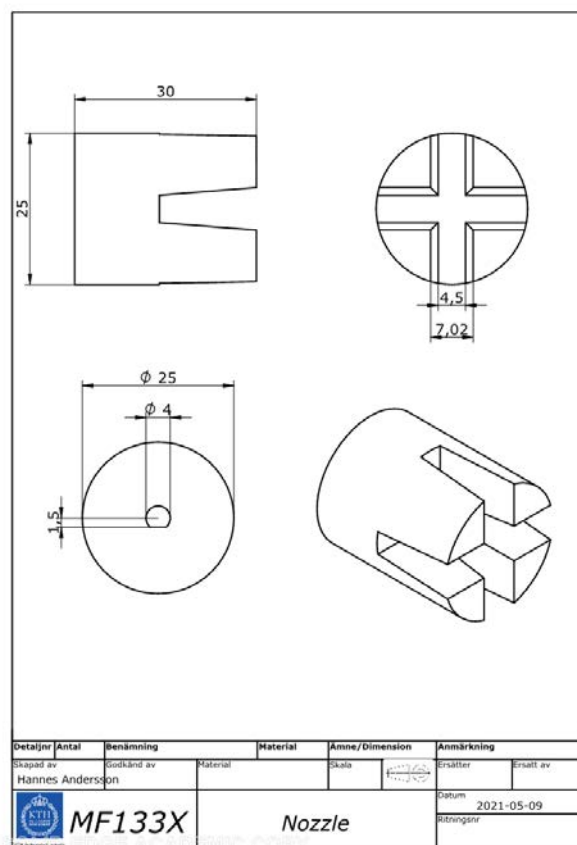
- [12] Adafruit. *Adafruit MAX4466*. 2021. URL: <https://www.adafruit.com/product/1063> (visited on 04/24/2021).
- [13] Amanda Ghassaei. *Arduino Audio Input*. URL: <https://www.instructables.com/Arduino-Audio-Input/> (visited on 04/07/2021).



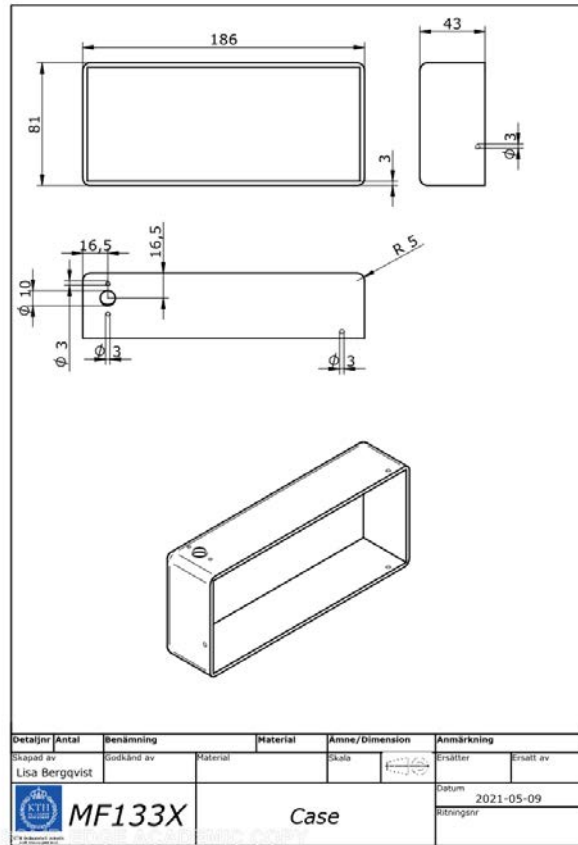
# Appendix A

## Technical Drawings

### A.1 This is the technical drawing of the Nozzle

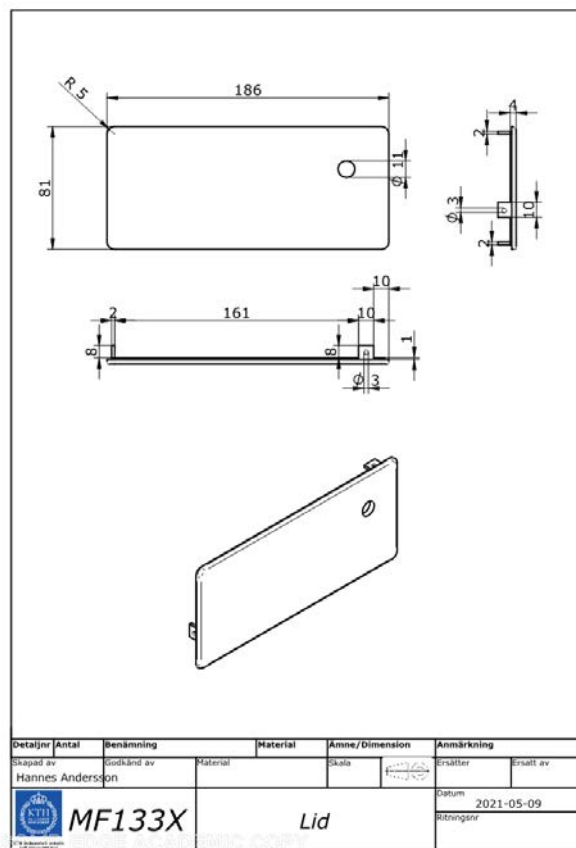


**A.2 This is the technical drawing of the case**



**A.3 This is the technical drawing of the lid**

A.3. THIS IS THE TECHNICAL DRAWING OF THE LID






# Appendix B

## Data Sheet for DC Motor

**DATI TECNICI**  
**TECHNICAL DATA**

serie **L149**



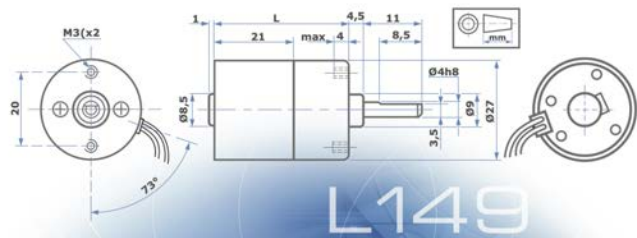
Soppressione disturbi con VDR sul collettore.  
Spazzole in metallo prezioso (Au - Ag - Cu)  
Direzione di rotazione secondo polarità  
Può essere montato in ogni posizione  
Massimo carico radiale: 10N  
Massimo carico assiale: 5N  
Temperatura di esercizio: -20°C/60°C  
Peso approssimativo: 55g

VDR interference suppression on the collector  
Precious metal brushes (Au - Ag - Cu)  
Direction of rotation depending on polarity  
Can be mounted in any position  
Maximum radial shaft load: 10N  
Maximum axial shaft load: 5N  
Temperature range: -20°C/60°C  
Approx weight: 55g


Valori tipici a temperatura ambiente +20°  
Tolleranza +/- 10%

Typical values at ambient temperature +20°  
Tolerance +/- 10%

TIPO TYPE	TENSIONE NOMINALE NOMINAL VOLTAGE		L mm	RAPPORTO -1 RATIO TO:1	COPPIA NOMINALE NOMINAL TORQUE		VELOCITÀ SPEED		CORRENTE CURRENT				
	v	mm			Nom	SENZA CARGO NO LOAD		CON COPPIA NOMINALE AT NOMINAL TORQUE		SENZA CARGO NO LOAD		CON COPPIA NOMINALE AT NOMINAL TORQUE	
						rpm	mA	rpm	mA	rpm	mA		
L149-B-10	4	4,5	36	10	1,5	255	165	<35	100	<35	85		
	12	5				215	120	<30	85				
L149-B-21	4	4,5	36	20,8	2,5	125	80	<35	100	<35	85		
	12	5				105	80	<30	85				
L149-B-43	4	4,5	41	43,3	3,8	60	40	<35	100	<35	85		
	12	5				52	32	<30	85				
L149-B-90	4	4,5	41	90,3	8	30	18	<35	100	<35	85		
	12	5				25	13	<30	85				
L149-B-188	4	4,5	46	188	14	14	9	<35	100	<35	85		
	12	5				12	7	<30	85				
L149-B-392	4	4,5	46	391,8	20	7	5	<35	90	<35	75		
	12	5				9	4	<30	75				



L149


 Viale Piave, 80/82 - 23879 VERDERIO (LC) ITALY  
 Tel. 039.510611-499 Fax 039.513617  
 www.micromotors.eu - info@micromotors.eu



## Appendix C

# Arduino code

### C.1 This code is used for the DC motor

```
/*
  School: KTH Royal Institute of Technology
  Course: MF133X Degree Project in Mechatronics, First Cycle
  Made by: Hannes Andersson and John Sjöberg
  finalized: 2021-05-8
  TRITA-ITM-EX 2021:26
  -----

  This code is used to read an analog signal, store it in a buffer,
  calculate the frequency of the signal, compare the calculated frequency
  with the frequency of the A-string (110 Hz), and from that, rotate the
  motor at a certain speed until and direction. The code repeats this
  until the correct frequency is detained.
*/
#define LENGTH 512
// Sample Frequency in kHz
const float sample_freq = 15625;
//Guitar Tuning table
const float stringFreq =110;
//The largest expected period of a string
int stringMaxPeriod = (int)sample_freq / (stringFreq / 2);
//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. ARDUINO CODE

```

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 = 1;
float error_sum;
float Kp = 1;
float Ki = 10;
float Kd = 0;
//JND in frequency for the A-string
float tol = 0.38;
float pid_range = 30;
float pwm_output;
//Motor Pins and Variables
int enA = 10;
int in1 = 9;
int in2 = 8;
//Variables for master state-machine
void setup() {
    //Initilaize pins and variables
    analogRead(A0);
    pinMode(enA, OUTPUT);
    pinMode(in1, OUTPUT);
    pinMode(in2, OUTPUT);
    pinMode(clipLight, OUTPUT);
    count = 0;

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

```



### C.1. THIS CODE IS USED FOR THE DC MOTOR

```
ADCSRA |= (1 << ADSC); //start ADC measurements
//set timer0 interrupt at 2kHz
TCCROA = 0; // set entire TCCROA register to 0
TCCROB = 0; // same for TCCROB
TCNT0 = 0; //initialize counter value to 0
// set compare match register for 10kHz increments
OCROA = 15; // = (16*10^6) / (10000*64) - 1 (must be <256)
// turn on CTC mode
TCCROA |= (1 << WGM01);
// Set CS01 and CS00 bits for 64 prescaler
TCCROB |= (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(TIMERO_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++;
    }
}
//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) * (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
```

## APPENDIX C. ARDUINO CODE

```

//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;
      dold = dpt;
      rold = 0;
      r = 0;
      dt = 0;
      dpt = 0;
      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 {

```

C.1. THIS CODE IS USED FOR THE DC MOTOR

```
    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) {
    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 / 4)
            ) {
            filtered_freq = freq_old;
            freq_per = freq_old;
        }
    }
}
// Remakes harmonics to the
// fundamental tone in a interval of +- 30 Hz
```

## APPENDIX C. ARDUINO CODE

```

    if (freq_per < (stringFreq - 30 ) ) {
        PID_Control(stringFreq, freq_per * 2);
    }
    else if (freq_per > (stringFreq + 30 ) ) {
        PID_Control(stringFreq, freq_per / 2);
    }
    else {
        PID_Control(stringFreq, freq_per);
    }
}
count = 0;
}
}
void PID_Control(float stringfreq, float inputFreq) {;
    old_output = error;
    error = stringfreq - inputFreq;
    error_sum += error / sample_freq;

    pid_output = Kp * error + Ki * error_sum + Kd * (error - old_output)
    * sample_freq / 1000;
    if (pid_output < -pid_range)
    {
        pid_output = -pid_range;
    }
    else if (pid_output > pid_range)
    {
        pid_output = pid_range;
    }
    if (abs(error) < tol) {
        error = 0;
        digitalWrite(in2, LOW);
        digitalWrite(in1, LOW);
        delay(1000);
        return;
    }
    else {
        pwm_output = map(pid_output, -pid_range, pid_range, 0, 100);
        motor_speed(pwm_output);
    }
}
}
void motor_speed(float pwm) {
    if (pwm <= 50)
    {

```

## C.2. THIS CODE IS USED FOR THE STEPPER MOTOR

```
        counter_clockwise(map(pwm, 0, 50, 100, 220));
    }
    else if (pwm > 50)
    {
        clockwise(map(pwm, 50, 100, 245, 255));
    }
}
void counter_clockwise(int PWM) {
    digitalWrite(in2, LOW);
    digitalWrite(in1, HIGH);
    analogWrite(enA, (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);
    }

    FreqCalc();

}
```

## C.2 This code is used for the stepper motor

```
/*
  School: KTH Royal Institute of Technology
  Course: MF133X Degree Project in Mechatronics, First Cycle
  Made by: Hannes Andersson and John Sjöberg
  finalized: 2021-05-8
  TRITA-ITM-EX 2021:26
  -----
  This code is used to read an analog signal, store it in a buffer,
  calculate the frequency of the signal, compare the calculated frequency
  with the frequency of the A-string (110 Hz), and from that, rotate the
  motor at a certain speed until and direction. The code repeats this
```

## APPENDIX C. ARDUINO CODE

```

    until the correct frequency is detained.
*/
// Define the stepper motor direction and step pin
#define dirPin 2
#define stepPin 3

#define LENGTH 512
// Sample Frequency in kHz
const float sample_freq = 15625;
//Guitar Tuning table
const float stringFreq = 110;
//The largest expected period of a string
int stringMaxPeriod = (int)sample_freq / (stringFreq / 2);
//index of current string
//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
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

float tol = 0.38; //JND in frequency
float error;
int steps, rot_speed;

void setup() {

    //Initilaize pins and variables
    analogRead(A0);
    pinMode(clipLight, OUTPUT);
    count = 0;

    cli();//disable interrupts
    //set up continuous sampling of analog pin 0

```

## C.2. THIS CODE IS USED FOR THE STEPPER MOTOR

```
//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); //| (1 << ADPS0); //set ADC
//clock with 32 prescaler- 16MHz/32=500kHz
ADCSRA |= (1 << ADSC); //enable auto trigger
ADCSRA |= (1 << ADIF); //enable interrupts when measurement complete
ADCSRA |= (1 << ADSC); //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(TIMERO_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++;
    }
}
```

```

}

//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) * (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
float 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;
            dold = dpt;
            rold = 0;
            r = 0;
            dt = 0;
            dpt = 0;
            for (j = 0; j < len / 2; j++) {
                if (j + tau >= len) {
                    r = 0;
                    rt = (rawData[j] - 128) * (rawData[j] - 128) / 256;
                }
            }
        }
    }
}

```



## C.2. THIS CODE IS USED FOR THE STEPPER MOTOR

```
    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) {
    break;
}
if (!tau) {
    thresh = rold * 0.5;
    pd_state = 1;
}
```

## APPENDIX C. ARDUINO CODE

```

}
// 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 / 4)
    ) {
      filtered_freq = freq_old;
      freq_per = freq_old;
    }
  }
}
count = 0;
}
// Remakes harmonics to the
// fundamental tone in a interval of +- 30 Hz
if (freq_per < (stringFreq - 30 ) ) {
  freq_per = (freq_per * 2);
}
else if (freq_per > (stringFreq + 30 ) ) {
  freq_per = (freq_per / 2);
}
else {
  freq_per = freq_per;
}
return freq_per;
}

// Calculates error and determines amount of steps and speed
void error_calc(float string_freq, float input_freq) {
  error = string_freq - input_freq;
  if (error < tol && error > -tol) {
    error = 0;
    delay(1000);
    return;
  }

  else if (error < 0) {
    digitalWrite(dirPin, HIGH);
  }
}

```

## C.2. THIS CODE IS USED FOR THE STEPPER MOTOR

```
    }

    else if (error > 0) {
        digitalWrite(dirPin, LOW);
    }

    if ( abs(error) < 3) {
        rot_speed = 15000;
        steps = 5;
    }

    else if ( abs(error) >= 3 && abs(error) < 6) {
        rot_speed = 10000;
        steps = 10;
    }

    else if ( abs(error) >= 6 && abs(error) < 20) {
        rot_speed = 10000;
        steps = 20;
    }

    else {
        rot_speed = 10000;
        steps = 1;
    }

    run_motor(rot_speed, steps);
}

// Rotates the motor with a specific amount of steps and speed
void run_motor(int _speed, int steps) {
    for (int i = 0; i < steps; i++) {
        digitalWrite(stepPin, HIGH);
        delayMicroseconds(_speed);
        digitalWrite(stepPin, LOW);
        delayMicroseconds(_speed);
    }
}

void loop() {
    if (clipping) {
        digitalWrite(clipLight, HIGH);
    }
    else {
        digitalWrite(clipLight, LOW);
    }
    freq_per = FreqCalc();
}
```

## APPENDIX C. ARDUINO CODE

```
    error_calc(stringFreq, freq_per);  
}
```

## Appendix D

# Acumen Code

```
//Andra program - simulering

//A simple simulation of an automatic guitar tuner adjusting a
//tuning peg on a guitar.

//Created by Group 8 - Degree Project in Mechatronics 2021
//Hannes Andersson
//John Sjöberg
//Date: 2021-03-28

//In Main the objects "tuner" and "guitar" are created.
//Then the variables y and r are declared in order to simulate the movement.
//Atlast the variables derivatives are used to increase their value,
//these are constantly added to their respective object.

model Main(simulator) =
  initially
    _3DView = (),
    tuner = create Tuner((0,0,0),(0,0,0)),
    guitar = create Guitar((0,0,0)),
    y = 0, y' = 0,
    r = 0, r' = 0
  always
    _3DView = ((-20,4,5.6),(0,10,0)),
    if y<5.7
      then y' = 1
      else y' = 0,
```

## APPENDIX D. ACUMEN CODE

```

tuner.pos = (0,y,0),
  if y > 5.69
    then r' = 0.4
    else r' = 0,
tuner.rot = (0,r,0),
guitar.rot =(0,r,0)

//An object "Tuner" is created which represents the guitar tuner.
//The argument "pos" enables the tuner to move as "y" increases.
//The argument "rot" enables the cylinder part to rotate as "r" increases.

model Tuner(rot,pos) =
  initially
    _3D = ()
  always
    _3D = (Box
      center = pos+(0,0,0)
      size = (2,3,2)
      color = red
      rotation = (0,0,0),
      Cylinder
      center = pos+(0,2,0)
      size = (1,0.5)
      color = green
      rotation = rot+(0,0,0))

//An object "Guitar" is created.
//The argument "rot" enables on of the tuning pegs to rotate together with
//the cylinder on the tuner as "r" increases.

model Guitar(rot) =
  initially
    _3D = ()
  always
    _3D = (Box //Headstock
      center = (0,10,0)
      size = (5.5,3,1)
      color = blue
      rotation = (0,0,0),
      Box //Tuning peg

```

```

center = (0,8.15,0)
size = (0.5,0.7,0.2)
color = yellow
rotation = rot+(0,0,0),
Box //Tuning peg
center = (1.5,8.15,0)
size = (0.5,0.7,0.2)
color = yellow
rotation = (0,0,0),
Box //Tuning peg
center = (-1.5,8.15,0)
size = (0.5,0.7,0.2)
color = yellow
rotation = (0,0,0),
Box //Tuning peg
center = (0,11.85,0)
size = (0.5,0.7,0.2)
color = yellow
rotation = (0,0,0),
Box //Tuning peg
center = (1.5,11.85,0)
size = (0.5,0.7,0.2)
color = yellow
rotation = (0,0,0),
Box //Tuning peg
center = (-1.5,11.85,0)
size = (0.5,0.7,0.2)
color = yellow
rotation = (0,0,0),
Box //Neck
center = (10.25,10,0)
size = (15,1.5,1)
color = 0.4*yellow + 0.6*white
rotation = (0,0,0),
Box //Body
center = (22.75,10,-1)
size = (10,7,3)
color = blue
rotation = (0,0,0))

```







TRITA -ITM-EX 2021:26