OSH Recorder : Frequency Detection and Calibration

TODO: RPM Updater / Control Mechanism

Kyle Fogarty[*]

December 2021

1 Introduction

In this document we will explore the problem of pitch detection and calibration for use in our OSH autonomous recorder player. We will begin by formally defining the problem we wish to tackle, before introducing the relevant theory required. Finally, we will finish by presenting the calibration control algorithm that will be implemented within our OHS recorder project.

In building our OSH recorder we have learnt that getting the 'right sound' out of the recorder is a difficult task. While a human player can, perhaps subconsciously, provide the correct air flow over the labium to produce the correct sound for each, automating this task is not easy. We will aim to build a calibration system that ensures that the correct pitch (fundamental frequency) of each note is achieved.

2 Problem Statement

In this work we will consider the OSH recorder system as a parameterised system and we will develop a calibration algorithm such that each musical note played is as close to its fundamental frequency as possible. We begin by noting that the only parameter we may control, to control the air flow rate Q, is the number of revolutions per minute (RPM) of the stepper motor which operates the compression of the artificial lung; our aim will be to find the correct value of RPM for each note.

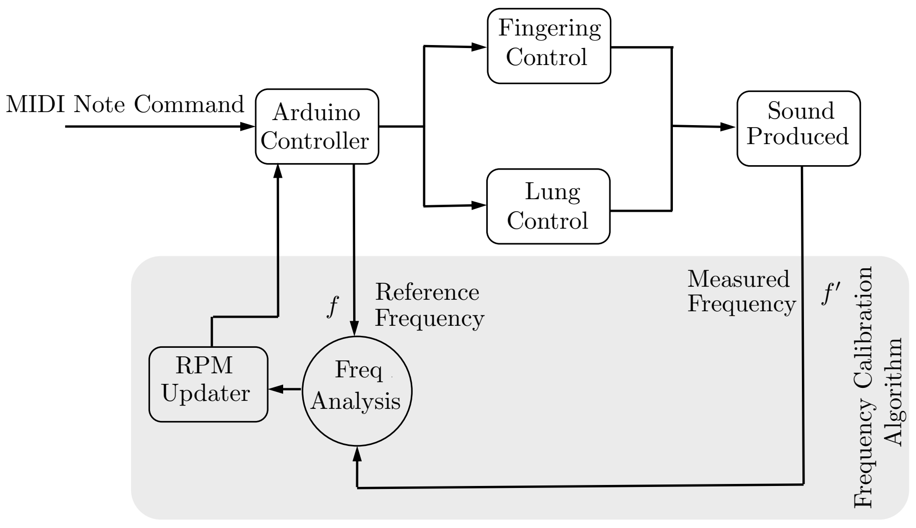

Figure 1: Schematic of how the 'calibration algorithm' fits into the OSH recorder player; In this document we will focus on the components shown in grey.

[*]MSc RAS student in the department of Computer Science, University of Lincoln, Lincoln, LN6 7TS, UK.
. Electronic mail : `kyle.thomas.fogarty@gmail.com`

Abstracting this problem, as in figure 1, we can see there are three main components:

1. Measure the frequency spectrum of the note that is played by the recorder.
2. Perform analysis of this measured frequency against a reference (theoretical) frequency.
3. Update the RPM of the stepper motor appropriately to minimise the difference.

We will consider how each of the following components can be implemented in the following sections.

3 Spectrum Detection

Fundamentally, sound is the oscillation of particles that propagate in a medium as longitudinal waves. Longitudinal waves are defines by regions of compression (high particle density) and rarefaction (low particle density), where the distance between these regions defines the wavelength of the wave [2], and so to record the signal of the sound wave we must be able to detect the changes in air pressure. This is typically done by measuring the displacement (a signed quantity) of a diaphragm, within a microphone, from its equilibrium position as a function of time; we will refer to this quantity as the amplitude spectrum.

3.1 Frequencies and Superposition

Before talking about how a frequency distribution can be obtained from the amplitude signal it is first important to consider what we mean by frequencies in this context. Fourier analysis tells us that any function[1] $f(x)$ can be represented as the sum of sines and cosines such that:

$$f(x) = a_0 + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{2\pi nx}{L}\right) + b_n \sin\left(\frac{2\pi nx}{L}\right) \right] \tag{1}$$

While the details of equation 1 are not too important, the salient point is that we many think of any function (including our amplitude signal) as a superposition of trigonometric functions with increasing frequency, as in figure 2. The question now is how can we decompose a function $f(x)$ into its fundamental frequencies.

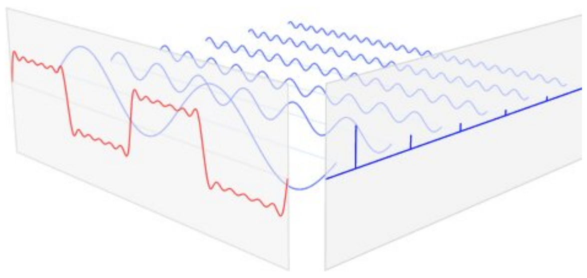

Figure 2: Approximation of a square wave with sine waves of increasing frequency. Figure taken from Ritchie Vink's blog.

3.2 The Fourier Transform

For the following we will assume we have access to the amplitude signal, and we will denote it as $f(t)$. We now wish to decompose this signal into its frequencies which can be done by application of the Fourier transform. We begin by formally introducing the Fourier Transform (FT) for continuous functions, before considering the Discrete Fourier transform (DFT) and the Fast Fourier Transform (FFT) algorithm.

[1]that is suitably nice, etc.

3.2.1 Continuous time FT

We begin with a formal definition of the Fourier Transform, which is taken from [1].

Definition 1 (Fourier Transform) *Let $f(t)$ be defined on $\mathbb{R}$, then*

$$\mathcal{F}\{f(t)\} = F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t} dt \tag{2}$$

is the complex Fourier Transform of $f(t)$, under the condition that $f(t)$ is absolutely integrable.

The condition of absolutely integrable in the above definition means that

$$\int_{-\infty}^{\infty} |f(t)| dt < \infty \tag{3}$$

holds. It is important to note that the function that is returned after the integral transform is complex values $F(\omega) \in \mathbb{C}$; the frequency can be interpreted as the magnitude of the complex function and while the argument corresponds to the phase-shift. Furthermore, it is also import to note that we only have access to samples of $f(t)$, not the continuous function itself, and so we must move from talking about the continuous domain into the discrete domain. This leads us to the Discrete Fourier Transform.

3.2.2 Discrete FT

The Discrete Fourier Transform is analogous to the continuous version, except it operates on a *finite* sequence of values; typically the sequence will be stored as a vector for computation.

Definition 2 (Discrete Fourier Transform) *The discrete Fourier transform transforms a sequence of N complex numbers $\{\boldsymbol{x}_n\} = x_1, x_2, ..., x_n$ is given by $\mathcal{F}(\boldsymbol{x}) = \{\boldsymbol{X}_\omega\}$ and each component is computed via*

$$X_\omega = \sum_{n=1}^{N} x_n e^{i\frac{2\pi\omega n}{N}} = \sum_{n=1}^{N} x_n \left(\cos\left(\frac{2\pi\omega n}{N}\right) + i\sin\left(\frac{2\pi\omega n}{N}\right)\right) \tag{4}$$

Hence, if we are able to perform the DFT on our samples of the amplitude signal we will return a sequence of complex values, where again the magnitude represents the frequency. It is important to note that the DFT as defined in equation 4 is computationally expensive: for each ω , of which there are N, there are N operations to compute which leads to a complexity of order $\mathcal{O}(N^2)$. This computational complexity can be beaten with the Fast Fourier Transform algorithm which we introduce next.

3.2.3 The FFT

We will not discuss the FFT algorithm in detail here, more information can be found here. The salient feature is that that the FFT has a complexity of $\mathcal{O}(N\log(N))$, which is far better than naively applying the definition of DFT in general. There are many libraries that contain implementations of the FFT algorithm, including `arduinoFFT` for the Arduino sketches and `SciPy` for Python.

3.3 Sampling considerations

When considering samples of a signal, it is important to consider two factors:

1. **Sample Frequency:** How often should samples of the signal be taken?
2. **Number of samples:** How many samples of the signal should be taken in total?

We will consider each of these considerations in turn:

3.3.1 Sample Frequency

Assuming we have a very large number of samples, it is clear that the higher the sampling frequency the more of the signal is captured. However, if our number of samples cannot be arbitrarily high then a very high sampling frequency with a low number of samples will mean that we don't capture much of the signal; hence we seek to find a compromise. Ideally, we would like to find the smallest sampling frequency from which we can reconstruct the full signal as this would allow use to observe the signal for the longest amount of time for a finite number of samples. This sort of question is the domain of information theory and in this work we will make use of the Nyquist–Shannon Sampling Theorem which is stated below.

Theorem 1 (Nyquist–Shannon Sampling Theorem) *A continuous-time signal $f(t)$ can be sampled at a frequency f_s in order to get a discrete-time copy of it $f[n]$, and afterwards be reconstructed perfectly to its original form $f(t)$ if $f_s > 2f_{\max}$ where $f_{\max}$ is the maximum frequency value of the $f(t)$ signal spectrum.*

Hence, in our case with the recorder, we must have $f_{max} = 1396.91$ Hz which corresponds to the note F_6 (This needs to be checked!) and so our sampling frequency should be greater than 2800 Hz.

3.3.2 Number of Samples

As mentioned above, the higher then number of samples the more of the signal that can be observed. For the FFT algorithm to work most efficiently, the number of samples should be a power of 2 (2^n for $n \in \mathbb{N}$). For the Arduino implementation memory is a serious constraint, with the Arduino Uno having a maximum number of samples of 128. This has the consequence, that the resolution in frequency space is approximately $f_s/N_s \approx \frac{2800}{128} = 21$ Hz. A Python implementation doesn't have the same constraint but then this would have to communicate with the Arduino over serial communication to update the RPM.

4 Implementation

4.1 Arduino

The frequency detection system has been implemented on an Arduino Uno, the circuit design is shown below in figure 3. We make use of the `arduinoFFT.h` library which may be included in the Arduino IDE using `Sketch`→`Include Library`→`Manage Libraries...`

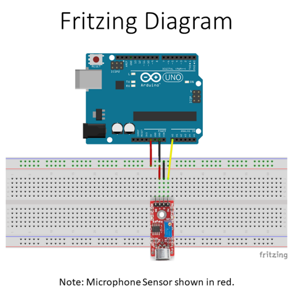

Figure 3: Arduino microphone setup. Figure taken from Adruino.cc

4.1.1 Arduino Code

```
/* ------------------------------------
 *  Arduino Frequency Detection Using FFT
 * ------------------------------------
 * This sketch builds upon code provided
 * one the website:http://shorturl.at/zRU57.
 * It makes use of the arduinoFFT lib which
 * can be installed via the libraries manager.
 *
 * Written : Kyle Fogarty
 * Date : 27th Dec 2021
 */


#include "arduinoFFT.h"

#define SAMPLES 128             // Number of amplitude samples taken from microphone
#define SAMPLING_FREQUENCY 2000 // [Hz] -> Nyquist{Shannon sampling theorem
                                //         requires sample freq = 2 * highest signal freq
                                //         that is to be recovered.

#define MICROPHONE_PIN 0        // Microphone plugged into the Analog

arduinoFFT FFT = arduinoFFT();  // Instantiate the FFT module

                                // [s] sampling period = 1 / (sampling freq)
unsigned int sampling_period_us round(1000000*(1.0/SAMPLING_FREQUENCY));

double FFTReal[SAMPLES];  // Array to hold the Real part of FT
double FFTIm[SAMPLES];    // Array to hold the Im part of FT

void setup() {Serial.begin(9600);}

void loop() {

/* ----------------
 * Signal Sampling
  ----------------*/

    for(int i=0; i<SAMPLES; i++)
    { unsigned long microseconds = micros();
      vReal[i] = analogRead(MICROPHONE_PIN);
      vIm[i] = 0;
      while(micros() < (microseconds + sampling_period_us)){}
    }

/* ----------------
 *    Perform FFT
  ----------------*/

    // Compute the Fourier TF
    FFT.Compute(FFTReal, FFTIm, SAMPLES, FFT_FORWARD);
    FFT.ComplexToMagnitude(FFTReal, FFTIm, SAMPLES);
    double peak = FFT.MajorPeak(FFTReal, SAMPLES, SAMPLING_FREQUENCY);

/* ----------------
 * Print Freq at Peak
 *   and bin range
  ----------------*/
```

```
    Serial.print("Freq Detected: ");
    Serial.print(peak);// Print out what frequency is the most dominant.

    double bin = peak / (SAMPLING_FREQUENCY/SAMPLES);
    int bin_low = bin;
    int bin_high = bin_low + 1;
    Serial.print(" | Bin Range: ");
    Serial.print(bin_low * (SAMPLING_FREQUENCY/SAMPLES));
    Serial.print(" - ");
    Serial.println(bin_high* (SAMPLING_FREQUENCY/SAMPLES));
    delay(250);
}
```

References

[1] Ronald Newbold Bracewell and Ronald N Bracewell. *The Fourier transform and its applications*, volume 31999. McGraw-Hill New York, 1986.

[2] Hugh D Young, Roger A Freedman, TR Sandin, and A Lewis Ford. *University physics*, volume 9. Addison-Wesley Reading, MA, 1996.