Abstract

The main goal of this project was to build a real-time tuner capable of detecting sound frequencies so that musicians can analyze their own sound and improve their performance. It begins by capturing audio input through a MAX9814 microphone module connected to a Teensy 4.1 microcontroller. The analog signal is processed using a Fast Fourier Transform (FFT) to identify the most dominant frequency. However, the maximum FFT size supported by Teensy is 1024, which divides the frequency range into bins that are approximately 43.07 Hz wide (based on the 44,100 Hz sampling rate and FFT size). This means the tuner can only identify the most dominant bin, not the precise frequency. To improve accuracy, I added parabolic interpolation, which models a parabolic curve using the magnitudes of adjacent FFT bins to estimate the peak between them. This allows the tuner to estimate sub-bin frequencies and produce more accurate pitch detection. Even then, there were odd inconsistencies that I could only narrow down to a noisy 5V power supply(mic module powered by 4 AA). To account for the voltage instability caused by the batteries' internal resistance, I added a 3.3microfarad capacitor to the microphone's power line. Since this project is tailored towards musicians, I wanted to find a way to turn Hz into an easily understandable tuning format, which would be notes and cents. In short, I converted Hz to the Musical Instrument Digital Interface(MIDI) standard.

How does audio sampling and FFT(Fast Fourier Transform) work?

1. The mic gives us an analog sound signal
   - This is a raw, wavy voltage that matches air pressure from sound.
2. The Teensy samples the analog signal
   - It measures the amplitude of the wave at regular intervals (for example. 44,100 samples per second, or measurements per second).
3. We collect 1024 of these samples
   - This creates 1024 chunks of sound.
4. We feed this 1024 sample chunk into the FFT
   - The FFT analyzes this whole chunk to figure out what frequencies are present in it.
5. The FFT divides the frequency range into 1024 bins
   - Each bin represents a specific range of frequencies (like 430–473 Hz).
   - It tells us how strong the sound is in each bin, but not the exact frequency.
   - However, we will only loop through bins 1-116(up to 5000 Hz), since this accounts for the appropriate sound frequencies that musical instruments can produce.
6. We find the bin with the highest energy (the dominant one)
   - That bin tells us where the loudest frequency is.
7. We estimate the actual frequency based on the bin index
   - Ex: **frequency = binIndex * (sampleRate / FFT size)**

Use MIDI notes to Convert Sound Frequencies to Musical Notes + Cents

- Need to tailor this project towards musicians
- A **MIDI note** is just a number that represents a specific musical pitch in the **MIDI** (Musical Instrument Digital Interface) standard
- MIDI notes increase by one for each semitone, and there are 12 semitones for one octave.
  - Ex: MIDI note number of A4 is 69, and MIDI note number of A3(one octave down) is 57
- Frequency Equation: **frequency = 440 * 2^(n / 12)**, where n is the number of semitones away from the reference note(A4, or 440 Hz)
  - A3 = 220 Hz
  - A4 = 440 Hz
  - A5 = 880 Hz
- Sound Frequency to MIDI note conversion: **midi_note = 69 + 12 * log$_2$(frequency / 440.0)**;
  - Solve for n in Frequency Equation → 12 * log$_2$(frequency / 440)
  - Set the MIDI note value of A4 by adding by 69.

At this point, I encountered issues and struggled to find a proper 5V power supply for my microphone module, so I discovered many different methods to create my own 5V power supply. In the end, I used a battery holder with 4 AA lithium rechargeable batteries and a battery snap connector found in the Arduino UNO starter kit to connect the battery holder to the microphone module. After testing with a digital multimeter, the voltage found using the battery holder was around 5.1 Volts.

Even after fixing my power supply issue, my tuner was not exactly accurate due to the fact that the displayed frequency was outputting the bin that the input frequency was the most dominant in. To address this issue, I decided to add **Parabolic Interpolation**.

Adding Parabolic Interpolation

- During my first MatLab course at CSULB(EE 202), we were able to model a polynomial given a set of data points.
- What if we were to apply these concepts to model a parabola using adjacent FFT bins. This will allow my program to not just select the bin where the most dominant frequency is in, but the precise frequency according to the model. However, we will need to model a new parabola each time the input signal is received, so we will use only 3 adjacent FFT bins.

Deriving Interpolation Index:

1. y= ax^2 + bx +c
   a. We know that the peak of the parabola occurs at x = -b/2a

b. Define MagL → x = -1 → a - b + c
c. Define MagC → x = 0 → c
d. Define MagR → x = 1 → a + b + c, now solve for a and b
e. MagR - MagL = 2b → **b = (MagR - MagL) / 2**
f. MagL = a - b + MagC; MagR = a + b + MagC
g. → a - b + a + b = MagL - 2MagC + MagR
h. a= (MagL - 2MagC + MagR) / 2

<u>**Voltage filtering:**</u>
Here is some data from the tuner at this point of the project testing at 470 Hz using a tone generator:

470 Hz
1. Detected: A#4 Freq: 466.55 Hz Cents: 1.4
2. Detected: A#4 Freq: 471.77 Hz Cents: 20.7
3. Detected: A#4 Freq: 468.06 Hz Cents: 7.0
4. Detected: A#4 Freq: 472.29 Hz Cents: 22.6
5. Detected: A#4 Freq: 472.53 Hz Cents: 23.5
6. Detected: A#4 Freq: 471.65 Hz Cents: 20.2
7. Detected: A#4 Freq: 471.15 Hz Cents: 18.4
8. Detected: A#4 Freq: 473.15 Hz Cents: 25.7
9. Detected: A#4 Freq: 470.97 Hz Cents: 17.8

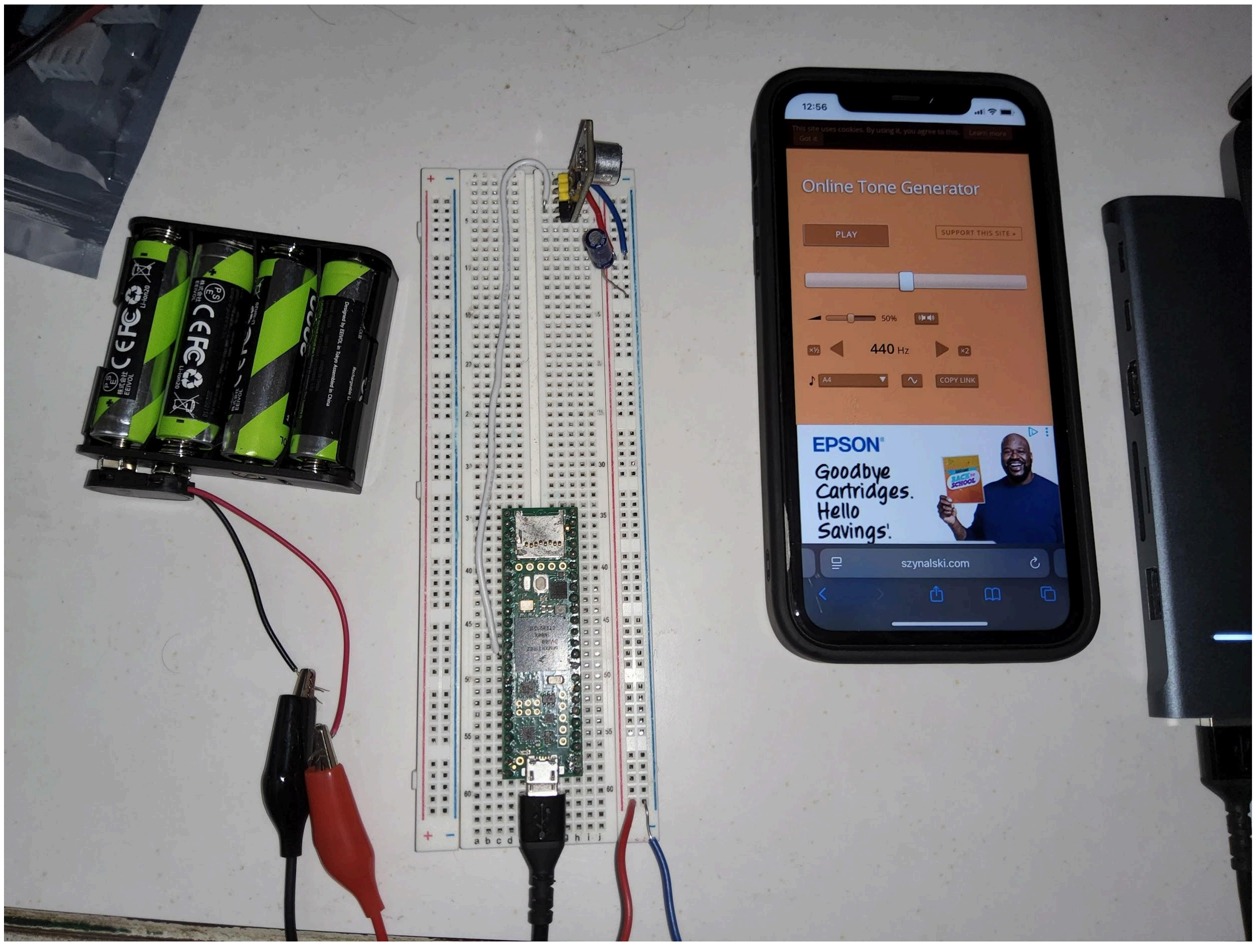
As you can see, the frequency fluctuates between 0-4 Hz, causing the cents to fluctuate by a large amount(goal is to have 14 cents for 470 Hz, since its out of tune, and 0 cents for 440 Hz, which is perfectly in tune for the note A). This inconsistency is unfit for a tuner for a musician, so I narrowed down the possible causes for high fluctuation from voltage ripple from 4 AA batteries(1.5V 3000mWh Li-ion rechargeable batteries). As a result, I added a 3.3 microfarad capacitor to fill in voltage dips and absorb voltage spikes.

<u>**Conclusion and Theoretical Solutions:**</u>
Even after all of this filtering, this project was not yet ready to be professionally used or even casually used for a student. In short, the tuner was simply not consistent enough in its readings. However, the data seemed really close to my desired results, and it's possible that the problems lies in the low sampling rate, lack of FFT size, and microphone module, all of which are innate hardware specs. Perhaps in the future I will return to this project with a different approach using the same materials, and maybe even add an interface so that the product can be used without depending on the Arduino IDE for outputs.

<u>**Code:**</u>
```
#include <Audio.h>        //mic input, FFT, waveform recordings
```

```
#include <SerialFlash.h> //Teensy-specific library to read/write data from onboard or
external flash memory

//Audio system objects
AudioInputAnalog          micInput;  //Teensy is hardwired to use A2 for analog mic
inputs
AudioAnalyzeFFT1024       fft;
AudioConnection           patchCord1(micInput, fft);


void setup() {
 Serial.begin(9600);
 AudioMemory(60);  //Allocate audio memory
 delay(1000);


}

void loop() {

//NEWEST VERSION
 if (fft.available()) {
 int maxIndex = 0; //FFT with bin with strongest frequency
 float maxValue = 0.0; //Magnitude of strongest bin

 //loop trhough all of the FFT bin to find the bin with highest magnitude
 for (int i = 1; i < 116; i++) {
   float magnitude = fft.read(i);
   if (magnitude > maxValue) {
     maxValue = magnitude;
     maxIndex = i;
   }
 }

 //get magnitudes of peak bin and 2 adjacent bins
 //  instead of having to manually define each bin, teensy internally returns the
magnitude each bin we loop trhough, so we can simply subtract and add 1 to the
maxIndex
 float magL = fft.read(maxIndex - 1);
 float magC = fft.read(maxIndex);
 float magR = fft.read(maxIndex + 1);
 //avoid division by zero
 float denom = (magL - 2 * magC + magR);
 float delta = 0.0;

 if (denom != 0.0) {
    //b = -magL + magR
    //a = denom
    // -b/2a = (magL - magR) / (2 * denom), which gives us the estimated peak of the
wave
    delta = (magL - magR) / (2 * denom); //this is the offset of maxIndex bin in BIN
UNITS, which will tell us the exact bin we are looking for by adding it to the
maxIndex
 }

 //Use the corrected index to estimate frequency
 float interpolatedIndex = maxIndex + delta;
 float frequency = interpolatedIndex * (44100.0 / 1024.0);
 //converted bin units to frequency
 //bin width is about 43.066 Hz



 //convert frequency to MIDI note
 float midiNote = 69 + 12 * log2(frequency / 440.0);
 int nearestNote = round(midiNote);
 int noteIndex = nearestNote % 12;
 int octave = nearestNote / 12 - 1;
 const char* noteNames[] = {"C", "C#", "D", "D#", "E", "F",
                            "F#", "G", "G#", "A", "A#", "B"};
 char note[12] = {0};
 sprintf(note, "%s%d", noteNames[noteIndex], octave);

 float idealFreq = 440.0 * pow(2.0, (nearestNote - 69) / 12.0);
 float cents = 1200 * log2(frequency / idealFreq);

 //Output
 Serial.print("Detected: ");
 Serial.print(note);
 Serial.print("   Freq: ");
 Serial.print(frequency, 2);
 Serial.print(" Hz   Cents: ");
 Serial.println(cents, 1);
 // //CPU Temperature Test
 // float celsius = tempmonGetTemp();
 // Serial.print("CPU Temp: ");
 // Serial.println(celsius);


 delay(100);



}
}
```

Schematic:



Acknowledgements:

I used ChatGPT throughout this project to help me better understand concepts related to audio signal processing, FFT analysis, and embedded implementation. While the AI assisted with explanations and occasional code suggestions, all hardware integration, testing, debugging, and final implementation were done independently. The project helped me develop a deeper understanding of real-time pitch detection and embedded system design.