Sound Driven Password Lock Utilizing FFT

Physics 4BL, Winter 24, March 23, 2024
Lab Section 2, Group 5

Onur Calisir and Steven Wang

Abstract

An FFT is a powerful tool that can take sound waves over time and produce frequency peaks that are dominant throughout the sample. This technology can be applied to password locks by taking a known sample of sound, running the FFT, and matching the produced frequency peaks to an existing database. A circuit board with Arduino was made and utilized to produce an intricate design of the password lock using sound. After successfully completing and confirming the microphone, the FFT circuit board was calibrated to create a tolerance range of about ±5 Hz for the continuous sound signals. The circuit board (Figure 2) simulated the password lock mechanics using a red and green LED light. The red light indicates unsuccessful attempts and the green light indicates a successful password match. The password system was characterized to be successful when using continuous sound signals. There was a discrepancy for harmonic and voice-associated password attempts due to inconsistency for the sound signals which required a higher tolerance range. A voice-driven password system is thus conceivable but needs much more sophisticated machinery and fine-tuning.

Introduction and Theory

Background

A sound wave can be deconstructed into the sum of various sine waves. This transformation can be described as a Fourier transform [1].

$$X(\omega) = \int_{-\infty}^{+\infty} x(t)e^{-j\omega t}dt \quad \text{(Eq. 1)}$$

This equation allows for a function of time to be transformed into a function of frequency which is useful for analysis and recording audio signals. As a sound is recorded, a graph of time against the amplitude of the sound wave can be recorded through a microphone; this can then be transformed using Eq. 1 to produce a graph of the frequency of the sound wave against its amplitude. This powerful tool allows for the analysis of sound, voice communication, and even earthquake magnitude.

This concept is developed further into the Fast Fourier Transform (FFT) which is an algorithm that can take a discrete signal instead of a continuous signal as an input [2]. This dramatically improves the functionality of the Fourier transform and is a faster computational method to perform Fourier transforms. The FFT technology is fundamental in the analysis of sounds because as a sound wave is recorded and put through the FFT, the most dominant signals can be captured and displayed. This useful technique of the FFT can be extracted into the concept of a sound-based password lock that uses the powerful deconstruction of an FFT to determine if the input of a sound matches preexisting sound frequencies.

Using an FFT, a sound-driven password lock can be created using the fundamental principles of a Fourier transform. Different sounds were recorded and then put through an FFT which determined the frequency of the sound. The sound inputs were also considered with the Nyquist-Shannon Sampling Theorem which dictates that the sampling frequency of the Arduino has to be twice as large as the given inputs [3]. This is because if a high frequency is inputted with a low sampling rate, the frequency may appear lower than its true frequency. Before recording and deciding on a tolerance range, the sampling frequency of the microphone had to be determined. This was done by having a counter variable in the Arduino code that would increase with every output and then dividing by the recording duration.

After determining the true sampling frequency, sound frequencies of varying wavelengths were put into a password bank for storage. The same sounds were recorded through the microphone to see if an FFT could pick the dominant frequencies and match them with the stored password bank; successful operation of the password lock could then be determined if the match was correct.

Correct utilization of both the FFT and sound inputs can conclude the usefulness of these technologies. As the FFT takes dominant signals in a sound wave, the peaks that are produced show certain frequencies that can be used as a password input. Since each sound wave has a unique set of dominant frequencies,

the FFT can be used to isolate these frequencies to match them into a collection of previously recorded and isolated dominant frequencies. Successful matches can then be visually seen using LED lights to showcase correct matches.

The experiments revealed the optimal usage of continuous sound signals compared to harmonic and voice frequencies. This led to more accurate password matches with the continuous sound signals as compared to the other frequencies used. Due to this, the tolerance range had to be modified to account for the stark variability of discontinuous sounds, and conclusions about the FFT-driven sound lock system were revealed to rely on better microcontrollers and higher frequencies.

Methods

In the first experiment, **Figure 1** illustrates how the microphone was set up to determine if the microphone could record various inputs generated. An Arduino code was used to show the recorded sound waves as integers to determine if the microphone got any inputs successfully.

The second experiment was done by creating a schematic that controls when the input of sound could be recorded using a button and then performing an FFT on the sound wave to obtain the tolerance range. **Figure 1** shows how the sound waves were recorded using an Arduino code that would later be transformed using an FFT program. Each sound wave was taken five times to produce the tolerance range.

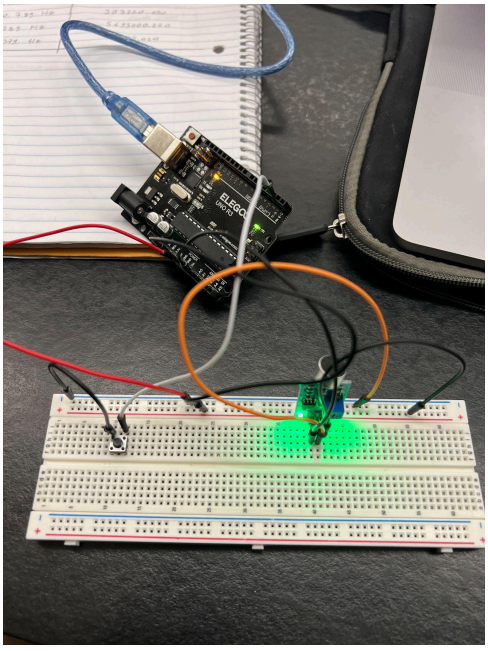

Figure 1. Microphone and Button Input for FFT tolerance
This circuit diagram shows how the FFT tolerance was recorded using the button.

The password lock was calibrated and a tolerance was created to take into account any variable differences between sound inputs and matches. Four different sounds were used to calibrate the password lock and create the tolerance range.

The first was a 200Hz chirp sound. This was done by using the Audacity program and making the setting go from 200 Hz to 200 Hz to maintain the consistency of the sound wave. The second sound wave that was sampled was a 400 Hz square wave using an online tone generator [4] which allowed for a continuous signal to be provided to the microphone. The third signal made use of harmonic sound waves that were taken from an online generator [5] where the C-major chord was continuously played for the duration of the recording window. The final sound wave

input was a group member saying “1,2,3.” The tolerance ranges were concluded to be ±5 Hz. This was extended for harmonic and voice sounds to 15 Hz.

The circuit board replicated LED lights as the password lock. **Figure 2** highlights the important aspects of the password lock. The green light will showcase a successful match whereas the red light shows an unsuccessful attempt. This is the fundamental experiment that puts together the FFT Arduino code and the existing sound-generated frequency database. The left-most button will allow for an input to be created and stored in an array where the FFT is performed. The next button then allows the user to play a sound to see if there are any matches.

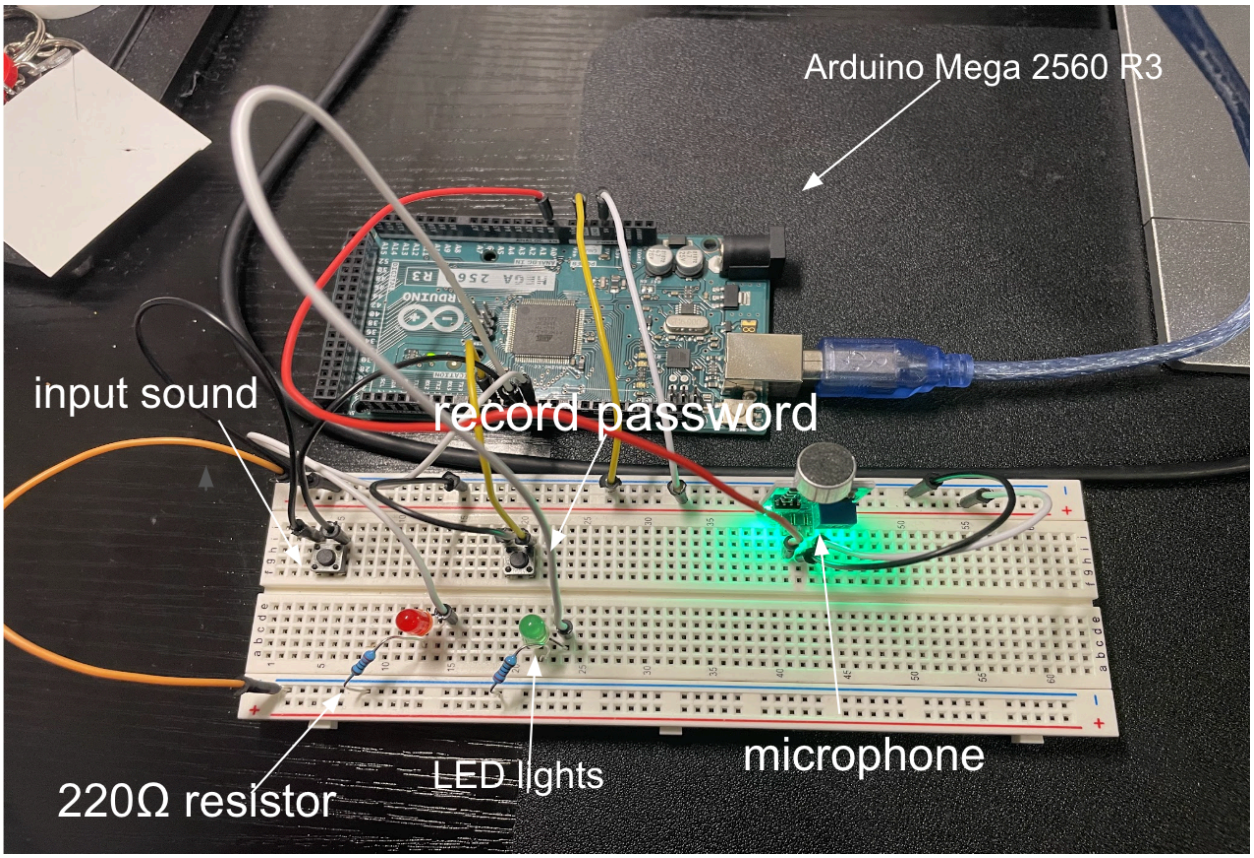

Figure 2. Password Lock using Sound.
The circuit board illustrates how the password lock circuit was made. Resistors are 220 ohms.

Results and Analysis

The results and analysis section will be separated into 4 subsections, each subsection will cover the analysis of one of the 4 tested signals. For each test signal selected, a brief recording of the test signal was recorded using the Arduino "Record" script (See Appendix). Using the analog microphone voltage readings, the signal was reconstructed in Python, and the FFT analysis was done. The FFT analysis returns the 5 most dominant frequencies within the Nyquist frequency band $[0, \frac{f_s}{2}]$. For each test signal recording, the Arduino sketch calculates the unique sampling rate using a counter variable that is incremented with every analog data recorded, by dividing the counter variable by the recording duration which is selected as 600 milliseconds to match the recording duration that is attainable using the Arduino "FFT Analysis" script (See Appendix) which is limited to collecting only 512 samples that can be held in the dynamic memory of the device and only has an internal sampling rate of 930 Hz. Using the experimental sampling frequency calculated by the Arduino sketch and the analog voltage data from the microphone, the Python FFT analysis can be done and the findings are to be compared with experimental results from the remaining of the tests.

For each signal, 20 tests were performed. First, 10 expected matches were conducted: these tests were performed by playing the password signal and observing the variance in the calculated dominant frequencies between the password array and the input array. Next, 10 expected rejections were tested: these were done by playing a combination of the other 3 test signals for a given password array, and the number of matches and the accuracy of the system rejections to these mismatches were measured. The test signals and results are shown below:

200 Hz Chirp Signal

The recording of the 200 Hz chirp signal resulted in a total count of 1496 data readings in the recording duration and yielded a sampling rate of $f_s = 1496/0.6 = 2493.33\ Hz$. The FFT analysis of the data set is shown in Figure 3, with the 5 most dominant peaks listed below. The Python analysis for this data set proved to be the cleanest and most accurate data set, capturing the effects of higher harmonic frequencies and the dominance of the 200 Hz root signal very accurately.

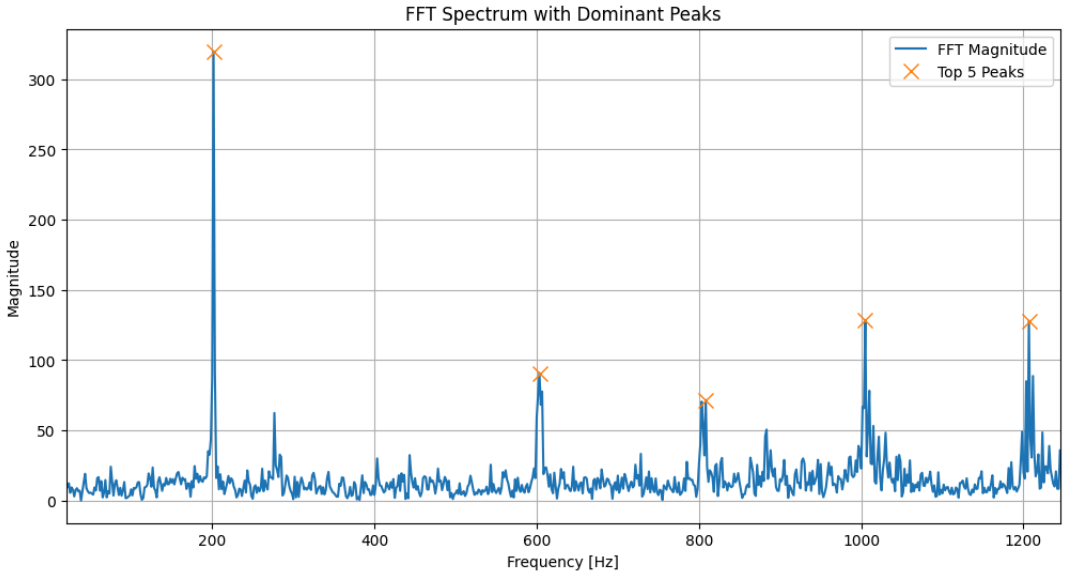

Figure 3: FFT Spectrum with Dominant Peaks for 200 Hz Chirp signal

The FFT analysis of the 200 Hz chirp signal yielded the following 5 most dominant frequencies by Python: 201.67 Hz, 603.33 Hz, 808.33 Hz, 10055.00 Hz, and 1206.67 Hz. Observing the most dominant frequencies, it can be seen that the greatest amplitude is achieved for 200 Hz, which is what is expected using a uniform 200 Hz test signal. The following dominant frequencies are also expected, as each of these frequencies is multiples of 200, which means the higher harmonic frequencies were captured by the Python program.

The Password Array set by the Arduino sketch to use by this test signal is shown below:

200 Hz chirp Password = [221.60, 223.42, 181.64, 179.82, 303.34] Hz

The Arduino sorts the array to be in decreasing order of the calculated amplitudes for the frequencies calculated to be the most dominant. It can be observed that the Arduino was able to capture 4 frequencies in the neighborhood of the 200 Hz frequency as expected, but failed to fully identify the 200 Hz peak like the Python analysis. This can be explained by the limitations of the FFT analysis done directly on Python, as the FFT sample size was limited to only 512 samples due to memory limitations, compared to 1496 samples analyzed by Python. However, for the project accuracy, an important comparison will be done between the analyzed data outputted by the Arduino code. Table 1 shows the successful match tests conducted using the 200 Hz chip signal. If the calculated dominant frequencies by the Arduino program are within the 5 Hz frequency tolerance with the Password array, it will be considered a successful match and the acceptability of the selected tolerance range will be further explored.

Table 1: 200 Hz Chirp Signal, Successful Match Experiment Results

Test #	Frequency Array [Hz]	Match Ratio	Successful Match within Tolerance
1	[221.60, 181.64, 223.42, 263.38 ,303.34]	⅘	Yes
2	[181.64, 221.60, 223.42, 263.38, 303.34]	⅘	Yes
3	[181.64 221.60 223.42 303.34 263.38]	⅘	Yes
4	[221.60, 223.42, 181.64, 303.34, 263.38]	⅘	Yes
5	[221.60, 223.42, 181.64, 303.34, 263.38]	⅘	Yes
6	[221.60, 223.42, 181.64, 303.34, 263.38]	⅘	Yes
7	[221.60, 223.42, 181.64, 263.38, 303.34]	⅘	Yes
8	[221.60, 223.42, 181.64, 263.38, 303.34]	⅘	Yes
9	[221.60, 181.64, 223.42, 263.38, 179.82]	⅘	Yes
10	[221.60, 223.42, 181.64, 263.38, 303.34]	⅘	Yes

Observing the results of the successful match experiment, it can be seen that the Arduino program was very accurate in calculating the dominant frequencies the input array frequencies are almost identical for each test and all tests proved to match ⅘ of the Password array frequencies. Overall it can be concluded that the system is very successful in determining a successful match given a 200 Hz chirp password and a 200 Hz chirp input. Furthermore, the 5 Hz frequency tolerance was more than acceptable, as all matched passwords were identical.

Table 2 shows the results of the successful rejection experiment for the 200 Hz chirp signal. For this experiment, 4 recordings of 440 Hz square wave, 4 recordings of “Onur 1-2-3” and 2 recordings of C-major were used.

Table 2: 200 Hz Chirp Signal, Successful Rejection Experiment Results

Test #	Frequency Array [Hz]	Match Ratio	Successful Rejection
1	[394.16, 346.93, 392.34, 345.12, 94.45]	0/5	Yes
2	[394.16, 346.93, 392.34, 345.12, 94.45]	0/5	Yes
3	[394.16, 346.93, 94.45, 345.12, 188.91]	0/5	Yes
4	[394.16, 346.93, 94.45, 345.12, 392.34]	0/5	Yes
5	[245.21, 127.15, 257.93, 123.52, 256.11]	0/5	Yes
6	[125.33, 127.15, 250.66, 252.48, 123.52]	0/5	Yes
7	[127.15, 128.96, 254.30, 125.33, 252.48]	0/5	Yes
8	[437.75, 435.94, 197.99, 439.57, 196.17]	0/5	Yes
9	[192.54, 190.72, 194.36, 381.45, 188.91]	0/5	Yes
10	[365.10, 332.40, 197.99, 363.28, 435.94]	0/5	Yes

Table 2 shows that the system is very efficient and successful in rejecting inputs that are not the same as the Password signal. A match ratio of 0/5 is very effective and proves that the 5 Hz tolerance is useful in successfully determining acceptances and rejections.

440 Hz Square Wave

The recording of the 440 Hz Square wave signal resulted in a total count of 1569 data readings in the recording duration and yielded a sampling rate of $f_s = 1569/0.6 = 2615.00\,Hz$. The FFT analysis of the data set is shown in Figure 4, with the 5 most dominant peaks listed below.

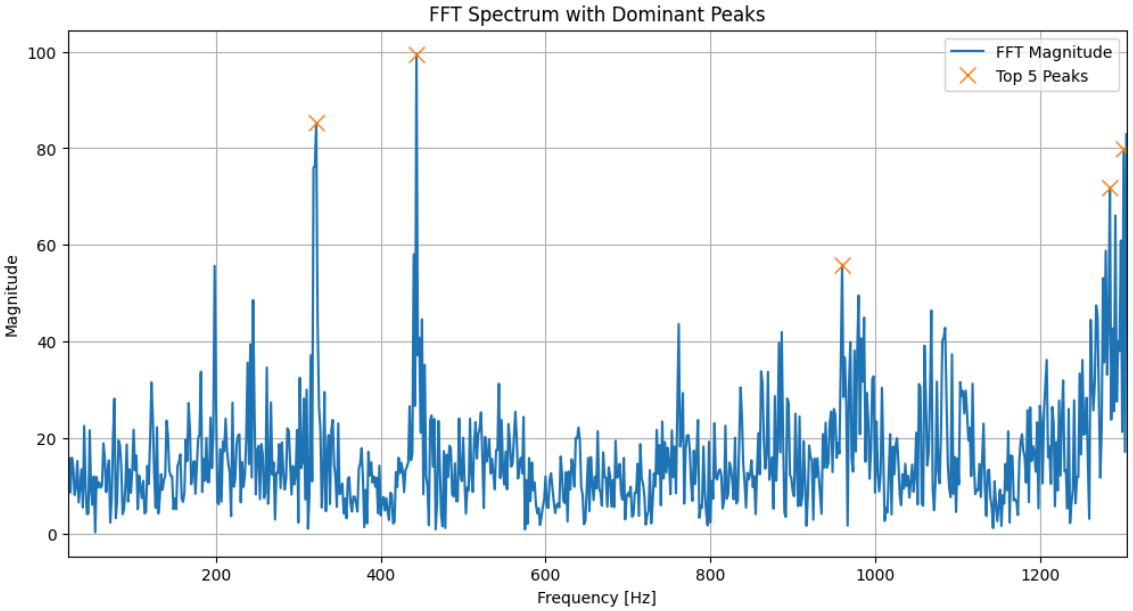

Figure 4: FFT Spectrum with Dominant Peaks for 440 Hz Square Wave

The FFT analysis of the 440 Hz Square wave signal yielded Python's 5 most dominant frequencies: 321.67 Hz, 443.33 Hz, 960.00 Hz, 1285.00 Hz, and 1301.67 Hz. Observing the most dominant frequencies, it can be seen that the Python analysis successfully identifies the 440 Hz from the square wave as the most dominant frequency with the largest amplitude, however, some other frequencies detected were seen to be outside the sampling rate of the Arduino, which causes the Arduino FFT analysis to not be able to successfully consider these frequencies.

The Password Array set by the Arduino sketch to use by this test signal is shown below:

440 Hz Square Wave Password = [394.16, 346.93, 94.45, 441.39, 392.34] Hz

It can be observed that the Arduino was able to capture the 440 Hz dominant frequency, however, the rest of the dominant frequencies identified by the Arduino are at best seen as random. This can be explained by noise from surroundings or the effects of windowing in the FFT analysis of the Arduino.

However again for the project accuracy, an important comparison will be done between the analyzed data outputted by the Arduino code. Table 3 shows the successful match tests conducted using the 440 Hz Square Wave signal. If the calculated dominant frequencies by the Arduino program are within the 5 Hz frequency tolerance with the Password array, it will be considered a successful match and the acceptability of the selected tolerance range will be further explored.

Table 3: 440 Hz Square Wave, Successful Match Experiment Results

Test #	Frequency Array [Hz]	Match Ratio	Successful Match within Tolerance
1	[394.16, 346.93, 94.45, 392.34, 345.12]	4/5	Yes
2	[394.16, 94.45, 346.93, 441.39, 392.34]	5/5	Yes
3	[394.16, 346.93, 94.45, 392.34, 345.12]	4/5	Yes
4	[394.16, 346.93, 94.45, 345.12, 392.34]	4/5	Yes
5	[394.16, 346.93, 94.45, 441.39, 188.91]	4/5	Yes
6	[394.16, 346.93, 94.45, 345.12, 441.39]	4/5	Yes
7	[394.16, 346.93, 94.45, 188.91, 345.12]	3/5	Yes
8	[394.16, 345.12, 346.93, 392.34, 94.45]	4/5	Yes
9	[346.93, 394.16, 94.45, 188.91, 441.39]	4/5	Yes
10	[394.16, 346.93, 94.45, 345.12, 188.91]	3/5	Yes

Observing the results of the successful match experiment, it can be seen that the Arduino program was very accurate in calculating the dominant frequencies; the input array frequencies are almost identical for each test and the average match ratio is found to be 3.9/5. Overall it can be concluded that the system is very successful in determining a successful match given a 440 Hz Square wave password and input. Furthermore, the 5 Hz frequency tolerance was more than acceptable, as all matched passwords only show an average standard deviation of 0.078.

Table 4 shows the results of the successful rejection experiment for the 440 Hz Square Wave signal. For this experiment, 4 recordings of 200 Hz chirp signal, 4 recordings of “Onur 1-2-3” and 2 recordings of C-major were used.

Table 4: 440 Hz Square Wave, Successful Rejection Experiment Results

Test #	Frequency Array [Hz]	Match Ratio	Successful Rejection
1	[21.80, 19.98, 426.86, 428.67, 423.22]	0/5	Yes
2	[417.77, 43.59, 463.18, 50.86, 232.50]	0/5	Yes
3	[381.45, 383.26, 386.89, 388.71, 385.08]	1/5	Yes
4	[23.61, 21.80, 25.43, 19.98, 30.88]	0/5	Yes
5	[134.41, 139.86, 138.05, 132.60, 274.28]	0/5	Yes
6	[143.50, 141.68, 145.31, 138.05, 150.76]	0/5	Yes
7	[150.76, 148.95, 152.58, 145.31, 158.03]	0/5	Yes
8	[181.64, 221.60, 223.42, 263.38, 179.82]	0/5	Yes
9	[221.60, 181.64, 223.42, 263.38, 179.82]	0/5	Yes
10	[366.91, 368.73, 365.10, 370.55, 372.36]	0/5	Yes

Table 4 shows that the system is very efficient and successful in rejecting inputs that are not the same as the Password signal. An average match ratio of 0.02 is very effective and proves that the 5 Hz tolerance is useful in successfully determining acceptances and rejections.

Onur “1-2-3” Test

The recording of the Onur “1-2-3” signal resulted in a total count of 1471 data readings in the recording duration and yielded a sampling rate of $f_s = 1471/0.6 = 2451.67\ Hz$. The FFT analysis of the data set is shown in Figure 5, with the 5 most dominant peaks listed below.

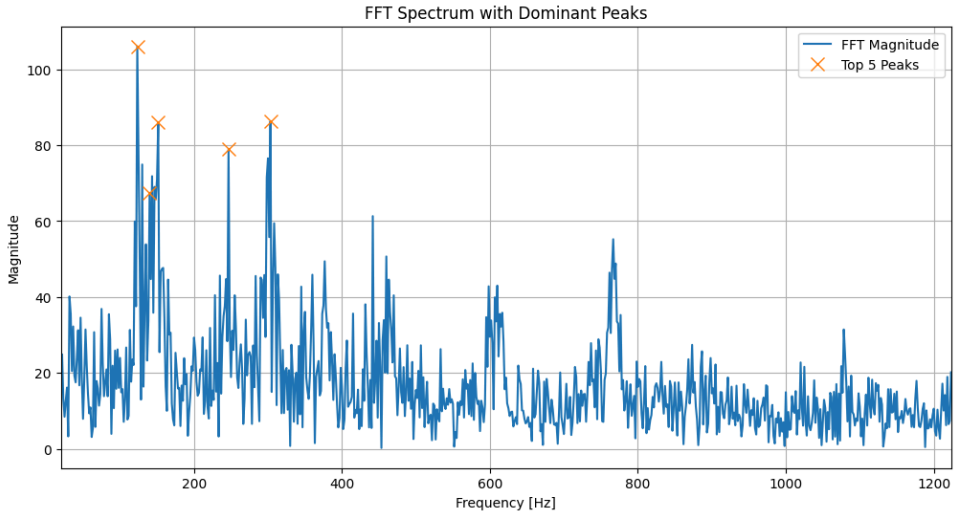

Figure 5: FFT Spectrum with Dominant Peaks for Onur “1-2-3” test signal

The FFT analysis of the Onur “1-2-3” test signal yielded Python's 5 most dominant frequencies: 123.33 Hz, 140.00 Hz, 151.67 Hz, 246.67 Hz, and 303.33 Hz. Observing the most dominant frequencies, it can be seen that the human voice input test signal shows a lot of fluctuations in the FFT spectrum, and the most dominant frequencies returned by the Python analysis are all found to be rather lower frequencies than previous tests. All dominant frequencies identified being in the Nyquist frequency range of the Arduino FFT analysis, this time it is expected that the Password Array set by the Arduino be similar to the one found by the Python analysis.

The Password Array set by the Arduino sketch to using this test signal is shown below:

Onur “1-2-3” Password 1 = [132.60, 261.56, 254.30, 268.83, 245.21] Hz

Onur “1-2-3” Password 2 = [125.33, 127.15, 123.52, 132.60, 252.48] Hz

Contrary to previous tests conducted for the Arduino FFT system, for a non-uniform human-generated signal, more variance in the frequency identification was observed. It was seen that the frequencies returned by the Arduino for 2 different passwords shared some similar frequencies, but also some that largely differed.

It can be seen that Password 2 proved to capture some frequencies that were very similar to those found by the Python analysis. Furthermore, initial testing of the system showed that the previously set tolerance range of 5 Hz proved more difficult to match when using human-generated passwords. As a result, the experiments done on the Onur “1-2-3” test signal will compare the effectiveness of tolerance ranges instead of acceptance/rejection efficiencies. Table 5 shows the results for the tested input tests using a 5 Hz tolerance range for successful matching.

Table 5: Onur “1-2-3”, Effectiveness of 5 Hz tolerance range

Test #	Frequency Array [Hz]	Match Ratio	Successful Match
1	[243.4, 23.61, 19.98, 21.8, 123.52]	1/5	No
2	[245.21, 239.77, 121.7, 237.95, 243.4]	1/5	No
3	[245.21, 123.52, 243.4, 130.78, 201.62]	2/5	No
4	[250.66, 121.7, 252.48, 248.85, 245.21]	2/5	No
5	[245.21, 241.58, 121.7, 118.07, 123.52]	1/5	No
6	[254.3, 414.14, 132.6, 125.33, 127.15]	2/5	No
7	[130.78, 257.93, 252.48, 268.83, 265.2]	4/5	Yes
8	[274.28, 136.23, 268.83, 138.05, 272.46]	2/5	No
9	[136.23, 141.68, 139.86, 134.41, 272.46]	2/5	No
10	[257.93, 128.96, 256.11, 127.15, 263.38]	3/5	Yes

Table 5 shows that only 2 of the 10 tests were a successful match using the 5 Hz tolerance with Password array 1. This shows that the 5 Hz tolerance was not an acceptable tolerance range for signals that are not uniform and computer generated such as the 200 Hz chirp and 440 Hz Square. The high inaccuracy in detecting a match for the Onur “1-2-3” test signal signifies the need for a higher tolerance range to accommodate the difficulty in producing the same input signal by a human user. The tone and pitch of the password and input recordings and frequencies detected vary greatly with a human-generated signal so to increase the effectiveness of the system, a 15 Hz frequency tolerance will be tested.

The selection of the 15 Hz tolerance range is useful for 2 main reasons, first, the tolerance range does not increase drastically to the point where the security of the lock is compromised, meaning it is not super easy to unlock the password array with any kind of voice input. Secondly, it allows to accommodate the random changes in the input sound signal and makes the lock more reliable. Table 6 shows the results of a second round of testing for Onur "1-2-3" signal with a 15 Hz frequency tolerance range.

Table 6: Onur "1-2-3", Effectiveness of 15 Hz tolerance range

Test #	Frequency Array [Hz]	Match Ratio	Successful Match
1	[245.21, 252.48, 121.7, 119.88, 243.4]	3/5	Yes
2	[248.85, 250.66, 121.7, 245.21, 127.15]	3/5	Yes
3	[256.11, 130.78, 257.93, 245.21, 250.66]	2/5	No
4	[248.85, 247.03, 254.3, 123.52, 125.33]	3/5	Yes
5	[127.15, 125.33, 261.56, 243.4, 132.6]	4/5	Yes
6	[132.6, 403.24, 250.66, 134.41, 408.69]	3/5	Yes
7	[257.93, 130.78, 263.38, 256.11, 252.48]	2/5	No
8	[125.33, 128.96, 130.78, 261.56, 247.03]	4/5	Yes
9	[132.6, 130.78, 134.41, 128.96, 138.05]	4/5	Yes
10	[250.66, 130.78, 125.33, 132.6, 259.75]	4/5	Yes

Table 6 shows that the 15 Hz tolerance range has more successful attempts than the 5 Hz tolerance tests. 80% of the tests were determined to be matches with the new tolerance range, which shows that the new tolerance range was not extremely lenient for accepting other inputs as matches while ensuring that the matches can be determined as well. This shows that for human-generated voice inputs, a frequency tolerance of 15 Hz is more useful and valid rather than the previously set 5 Hz tolerance.

C major Piano Chord

The recording of a C major piano chord resulted in a total count of 1694 data readings in the recording duration and yielded a sampling rate of $f_s = 1694/0.6 = 2823.33\,Hz$. The FFT analysis of the data set is shown in Figure 6, with the 5 most dominant peaks listed below.

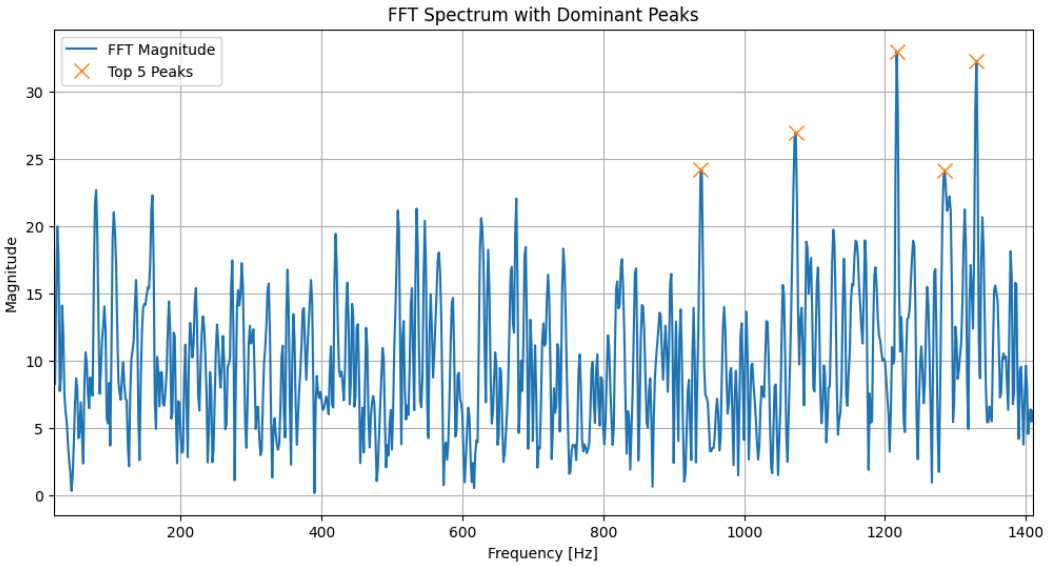

Figure 6: FFT Spectrum with Dominant Peaks for C major piano chord

The FFT analysis of the C major piano chord test yielded the following 5 most dominant frequencies by Python: 938.33 Hz, 1073.33 Hz, 1216.67 Hz, 1285.00 Hz, and 1330.00 Hz. Observing the most dominant frequencies, it can be seen that all frequencies returned by the Python analysis are above the internal sampling rate of the Arduino FFT analysis, meaning that the Arduino will not be picking any of these frequencies as a result. Furthermore, it is known that the resonance frequency of the C major chord is most likely around 520 Hz, meaning that the frequencies from the Python analysis are likely to be higher harmonic frequencies which makes it impossible for the Arduino to consider these frequencies with such a low sampling rate.

The Password Array set by the Arduino sketch to using this test signal is shown below:

C major piano chord Password 1 = [434.12, 365.10, 435.94, 363.28, 183.46] Hz

C major piano chord Password 2 = [365.10, 435.94, 434.12, 197.99, 203.44] Hz

Similar to the “Onur 1-2-3” tests, the C major piano chord tests dominant frequencies from the Arduino largely differed from those returned by the Python Analysis. As expected, the Arduino analysis is unable to analyze frequencies above 930 Hz, the internal sampling rate of the Arduino FFT algorithm, which made it unable to detect the higher harmonic frequencies of the C major piano chord tests the same way the Python analysis did. The two password arrays show some similarities, however, the tolerance range of 5 Hz again showed some similar results to the previous tests, while a 15 Hz tolerance showed increased reliability and accuracy in password unlocking. Table 7 shows the C major test results for the tested input tests using a 5 Hz tolerance range for successful matching.

Table 7: C major piano chord, Effectiveness of 5 Hz tolerance range

Test #	Frequency Array [Hz]	Match Ratio	Successful Match
1	[366.91, 365.10, 368.73, 370.55, 363.28]	2/5	No
2	[365.10, 363.28, 366.91, 361.46, 183.46]	3/5	Yes
3	[377.81, 379.63, 365.10, 366.91, 434.12]	3/5	Yes
4	[365.10, 379.63, 363.28, 381.45, 377.81]	2/5	No
5	[365.10, 363.28, 435.94, 366.91, 168.93]	3/5	Yes
6	[365.10, 366.91, 363.28, 379.63, 368.73]	2/5	No
7	[365.10, 363.28, 168.93, 170.74, 366.91]	2/5	No
8	[365.10, 366.91, 363.28, 437.75, 168.93]	3/5	Yes
9	[435.94, 437.75, 434.12, 432.30, 197.99]	2/5	No
10	[197.99, 434.12, 365.10, 168.93, 432.30]	3/5	Yes

From Figure 7, it can be seen that a 5 Hz tolerance range only gave 50% success in unlocking the Ardunio lock, even though the same signal was played for each test. This shows that the Arduino FFT analysis is highly unreliable when using a piano chord signal, especially with higher harmonic frequencies. Even though it was expected for a computer-generated piano chord signal to be more similar to the other computer-generated recording tests such as the 200 Hz chirp and 440 Hz square wave tests, the C major tests show more similarities to the “Onur 1-2-3” tests. This shows that the inaccuracies observed in the frequency matching are not due to the unpredictable nature of a human-generated signal, but rather the effects of non-uniform, highly changing characteristics of the last two signals.

Similar to “Onur 1-2-3” tests, the C major piano signal will be put into the test with a 15 Hz tolerance range to see if this changed tolerance range will yield better results. Table 8 shows the results of a second round of testing for C major piano chords with a 15 Hz frequency tolerance range.

Table 8: C major piano chord, Effectiveness of 15 Hz tolerance range

Test #	Frequency Array [Hz]	Match Ratio	Successful Match
1	[365.10, 363.28, 366.91, 435.94, 376.00	2/5	No
2	[434.12, 435.94, 197.99, 196.17, 365.10	5/5	Yes
3	[197.99, 434.12, 196.17, 365.10, 435.94	5/5	Yes
4	[435.94, 365.10, 366.91, 183.46, 437.75	4/5	Yes
5	[197.99, 434.12, 203.44, 196.17, 365.10	4/5	Yes
6	[435.94, 434.12, 197.99, 199.80, 432.30	4/5	Yes
7	[379.63, 365.10, 366.91, 377.81, 381.45	1/5	No
8	[434.12, 365.10, 197.99, 432.30, 366.91	4/5	Yes
9	[365.10, 363.28, 435.94, 434.12, 366.91]	3/5	Yes
10	[196.17, 365.10, 363.28, 435.94, 374.18	3/5	Yes

From Table 8 it can be seen that the 15 Hz tolerance range test had higher accuracy for frequency matching, with only 2 tests failing the test. Test 7 can be seen as an outlier as this was the first time a test returned 1/5 match out of every test. Removing test 7, assuming there was an issue with the microphone picking up the recording, the success rate of almost 90%, as expected. This proves the assumption that while uniform signals such as the 200 Hz chirp and 440 Hz Square wave work very effectively with a low tolerance range of 5 Hz, whereas non-uniform frequency signals such as the “Onur 1-2-3” and C major piano signals work best with a higher frequency tolerance range such as 15 Hz. Even though a 5 Hz tolerance range works better for security and robustness reasons, a 15 Hz tolerance is not a major difference that would render the system unusable. For professional use, the system works better with a frequency tolerance between 5-15 Hz, so it could accommodate all kinds of password signals, and show the best results and accuracy when unlocking.

Conclusion

The 200 Hz chirp and 400 Hz square wave produced results that mirrored the theoretical expectancy of success. In both rejection and match attempts, these two sounds were able to match the expected outcome and had a predictable pattern. The human voice attempt was lackluster and needed a higher tolerance range to be able to successfully match the given FFT signals; this pattern would be seen with sound inputs that were not consistent and had more variability. Furthermore, the harmonic C-major chord input also had similar results with the voice inconsistency. With these in mind, a higher tolerance range was built around these two inputs and would be a future point of contention to focus on.

Although the system performed exceptionally well regarding a continuous sound signal, there were fluctuations and uncertainties when harmonic and voice sounds were inputted into the system. This was solved using a larger tolerance range. In doing so, the system was able to be more precise and was able to effectively distinguish between each voice and harmonic sound input. Although tolerance accommodated the continuous signals, the harmonic sound waves and voice inputs had to be modified due to the harmonic sound being continuously played for the tolerance input. This interfered with the FFT analysis but was taken due to it being necessary for tolerance determination. Voice inputs were much harder to decide for tolerance. “1,2,3” was the designated voice input of choice due to the relative maintenance of the pitch and speed, compared to other buzzwords such as “Hello”.This tolerance was accommodated by increasing the ±5 Hz tolerance range to ±15 Hz.

Some limitations that prevented the system from becoming more robust and precise were the limited machinery and materials that could have been used to create a better sound-driven password lock. For future experiments, a better microcontroller would allow for more samples to be taken and more memory for the system. This can increase the system’s functionality to be fine-tuned and accurate. Another improvement is determining an effective way to increase the sampling rate of the system. This will allow for higher frequencies to be stored and utilized for a wider range of sounds to enforce integrity and customization.

Future extensions of this work are to expand upon the breadth of utilization for password locking. These would include mechanical locks, doors, and safes that would allow for personal passwords to be implemented. Another extension would be using the FFT technology and discovering other modes of recognition such as pictures for password locking. The sound-driven password lock thus has been characterized as a system that can take sound inputs and transform them into quantitative states to be used as a password system.

References

[1] Dcbradley@wisc.edu. “Fourier Sound Analyzer.” L.R. Ingersoll Physics Museum, www.physics.wisc.edu/ingersollmuseum/exhibits/waves/fourier/#:~:text=A%20Fourier%20spectrum%20is%20produced,not%20sound%20but%20heat%20flows. Accessed 28 Feb. 2024.

[2] Xuan, O. Z. (2023, April 24). Frequency analysis of audio signals with Fourier transform. Medium. https://medium.com/@ongzhixuan/frequency-analysis-of-audio-signals-with-fourier-transform-f89ac113a2b4

[3] What is FFT and how can you implement it on an Arduino?. Norwegian Creations. (1963, January 1). https://www.norwegiancreations.com/2017/08/what-is-fft-and-how-can-you-implement-it-on-an-arduino/

[4] Online tone generator. onlinetonegenerator.com. (n.d.). https://onlinetonegenerator.com/#google_vignette

[5] Piano Chords: Simple online piano chord player. Magical Music Theory Tools to Learn Music Online for Free. (n.d.). https://muted.io/piano-chords/

Code Appendix

Python:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq

from scipy.signal import find_peaks
def analyze_fft(data, sampling_rate, height, start_freq=20):
    N = len(data)  # Length of the data
    yf = fft(data)  # Compute the FFT
    xf = fftfreq(N, 1 / sampling_rate)  # Generate the frequency bins
    # Compute the magnitude of the FFT and only take the positive half (since it's symmetrical)
    yf_abs = 2.0 / N * np.abs(yf[:N // 2])
    xf_pos = xf[:N // 2]
    # Find the index corresponding to the start frequency (20 Hz)
    start_idx = np.argmax(xf_pos >= start_freq)
    # Find peaks with the specified height and distance criteria, starting from 20 Hz
    peaks, properties = find_peaks(yf_abs[start_idx:], height=height, distance=(10 * N) / sampling_rate)
    peaks += start_idx  # Correct indices because of the offset start_idx
    # Sort the peaks by their magnitude (highest first) and select the top 5
    sorted_indices = np.argsort(properties['peak_heights'])[::-1]
    top_peaks = sorted_indices[:5]
    dominant_frequencies = xf_pos[peaks[top_peaks]]
    # Plot the FFT spectrum
    plt.figure(figsize=(12, 6))
    plt.plot(xf_pos[start_idx:], yf_abs[start_idx:], label='FFT Magnitude')
    # Mark the 5 largest peaks
    plt.plot(xf_pos[peaks[top_peaks]], yf_abs[peaks[top_peaks]], "x", markersize=10, label='Top 5 Peaks')
    # Add labels and title
    plt.xlabel('Frequency [Hz]')
    plt.ylabel('Magnitude')
    plt.title('FFT Spectrum with Dominant Peaks')
    plt.legend()
    plt.grid(True)

    # Set x-axis limits to start from start_freq
    plt.xlim(start_freq, xf_pos[-1])
   # Show the plot
    plt.show()
    return dominant_frequencies
test1 = np.loadtxt('/content/200Hz.txt', skiprows =1)
rate1 = 2493.33
dominant_frequencies = analyze_fft(test1, rate1, 40)
print(f"The 5 most dominant frequencies are: {dominant_frequencies} Hz")
test2 = np.loadtxt('/content/440Hz_square.txt', skiprows =1)
rate2 = 2615
dominant_frequencies2 = analyze_fft(test2, rate2, 40)
print(f"The 5 most dominant frequencies are: {dominant_frequencies2} Hz")
test3 = np.loadtxt('/content/Cmajor_piano.txt', skiprows =1)
rate3 = 2823.33
dominant_frequencies3 = analyze_fft(test3, rate3, 20)
print(f"The 5 most dominant frequencies are: {dominant_frequencies3} Hz")
test4 = np.loadtxt('/content/1_2_3_Onur.txt', skiprows =1)
```

```
rate4 = 2451.67
dominant_frequencies4 = analyze_fft(test4, rate4, 20)
print(f"The 5 most dominant frequencies are: {dominant_frequencies4} Hz")
```

Arduino Code “Record”:

```
const int passwordPin = 3; // Password button connected to pin 2
const int micPin = A0; // Microphone connected to A0
const unsigned long recordingDuration = 600; // Recording duration in milliseconds

void setup() {
  pinMode(passwordPin, INPUT_PULLUP); // Initializes the button pin as an input with an internal pull-up resistor
  Serial.begin(115200);
}
void loop() {
  if (digitalRead(passwordPin) == LOW) {
    while(digitalRead(passwordPin) == LOW);
    Serial.println("Start Recording");
    unsigned long startTime = millis();
    unsigned long count = 0; // Add a counter for the readings
    while (millis() - startTime < recordingDuration) {
      int micValue = analogRead(micPin);
      Serial.println(micValue);
      count++; // Increment the counter
    }
    Serial.println("Stopped recording.");
    Serial.print("Total Readings: ");
    Serial.println(count);
    Serial.print("Estimated Sampling Rate (Hz): ");
    Serial.println(count / (recordingDuration / 1000.0)); // Calculate and print the estimated sampling rate
  }
}
```

Arduino Code “FFT Analysis”:

```
#include <arduinoFFT.h>
#include <Servo.h>
arduinoFFT FFT = arduinoFFT(); // Create FFT object
Servo servo1; // Connected to digital pin 6
// Define pin numbers
const int micPin = A0; // Microphone pin
const int passwordPin = 2; // Password setting button pin
const int inputPin = 3; // Input verification button pin
const int ledGreen = 4;
const int ledRed = 5;
const unsigned int samples = 512; // Number of samples
double samplingFrequency = 930.0; // Sampling frequency in Hz
// Arrays to hold real and imaginary values
double vReal[samples];
double vImag[samples];
// Structure to hold frequency and magnitude
struct FrequencyPeak {
  double frequency;
  double magnitude;
};
```

```
// Arrays to store the dominant frequencies for password and input
FrequencyPeak passwordPeaks[5];
FrequencyPeak inputPeaks[5];
// Function prototypes
void recordAndProcessAudio(FrequencyPeak peaks[5]);
void findTopFrequencies(double* vReal, double* vImag, unsigned int samples, double samplingFrequency, FrequencyPeak
peaks[], unsigned int numberOfPeaks);
bool compareFrequencyArrays(FrequencyPeak peaks1[], FrequencyPeak peaks2[], unsigned int numberOfPeaks, double
tolerance);
void setup() {
  Serial.begin(115200); // Start serial communication
  pinMode(micPin, INPUT);
  pinMode(passwordPin, INPUT_PULLUP);
  pinMode(inputPin, INPUT_PULLUP);
   pinMode(ledGreen, OUTPUT);
  pinMode(ledRed, OUTPUT);
  servo1.attach(6);
}
void loop() {
  if (digitalRead(passwordPin) == LOW) {
    // Record and process audio for password setting
    delay(100); // Simple debounce
    recordAndProcessAudio(passwordPeaks);
    Serial.println("Password set with the following frequencies:");
    for (int i = 0; i < 5; i++) {
      Serial.print(passwordPeaks[i].frequency);
      Serial.println(" Hz");
    }
    if (passwordPeaks[0].frequency > 20) { // Assuming the first peak frequency > 20 Hz indicates password is set
      servo1.write(90); // Rotate servo to 90 degrees to unlock
    }
  }
  if (digitalRead(inputPin) == LOW) {
    // Record and process audio for input verification
    delay(100); // Simple debounce
    recordAndProcessAudio(inputPeaks);
    Serial.println("Input recorded with the following frequencies:");
    for (int i = 0; i < 5; i++) {
      Serial.print(inputPeaks[i].frequency);
      Serial.println(" Hz");
    }
    // Compare the input with the stored password
    if (compareFrequencyArrays(passwordPeaks, inputPeaks, 5, 20.0 /* tolerance in Hz */)) {
      Serial.println("Password Match");
      digitalWrite(ledGreen, HIGH);
      delay(1000); // Show match for 1 second
      digitalWrite(ledGreen, LOW);
      delay(500);
      servo1.write(180);
    } else {
      Serial.println("Password Mismatch");
      digitalWrite(ledRed, HIGH);
      delay(1000); // Show mismatch for 1 second
      digitalWrite(ledRed, LOW);
```

```
    }
  }
}
void recordAndProcessAudio(FrequencyPeak peaks[5]) {
  for (unsigned int i = 0; i < samples; i++) {
    vImag[i] = 0; // Reset the imaginary part
    vReal[i] = analogRead(micPin); // Collect samples
    delayMicroseconds(1000000 / samplingFrequency); // Wait for the next sample period
  }
  FFT.Windowing(vReal, samples, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
  FFT.Compute(vReal, vImag, samples, FFT_FORWARD);
  FFT.ComplexToMagnitude(vReal, vImag, samples);
  findTopFrequencies(vReal, vImag, samples, samplingFrequency, peaks, 5);
}
void findTopFrequencies(double* vReal, double* vImag, unsigned int samples, double samplingFrequency, FrequencyPeak
peaks[], unsigned int numberOfPeaks) {
  // Determine the index corresponding to 20 Hz
  unsigned int minIndex = (20 * samples) / samplingFrequency;
  for (unsigned int i = 0; i < numberOfPeaks; i++) {
    peaks[i].magnitude = 0; // Initialize
    peaks[i].frequency = 0;
  }
  for (unsigned int i = minIndex; i < samples / 2; i++) { // Start search from 20 Hz onwards
    double frequency = (i * samplingFrequency) / samples;
    double magnitude = vReal[i]; // Magnitude from FFT result
    for (unsigned int j = 0; j < numberOfPeaks; j++) {
      if (magnitude > peaks[j].magnitude) {
        // Shift peaks down in the array
        for (unsigned int k = numberOfPeaks - 1; k > j; k--) {
          peaks[k] = peaks[k - 1];
        }
        // Insert new peak
        peaks[j].frequency = frequency;
        peaks[j].magnitude = magnitude;
        break;
      }
    }
  }
}
bool compareFrequencyArrays(FrequencyPeak peaks1[], FrequencyPeak peaks2[], unsigned int numberOfPeaks, double
tolerance) {
  unsigned int matchCount = 0;
  for (unsigned int i = 0; i < numberOfPeaks; i++) {
    for (unsigned int j = 0; j < numberOfPeaks; j++) {
      if (abs(peaks1[i].frequency - peaks2[j].frequency) <= tolerance) {
        matchCount++;
        break; // Found a match for peaks1[i], move to the next peak
      }
    }
  }
  return matchCount > 3;
}
```