2021 JD Accelerated Project

Abdulla AlMannai

Kevin Kott

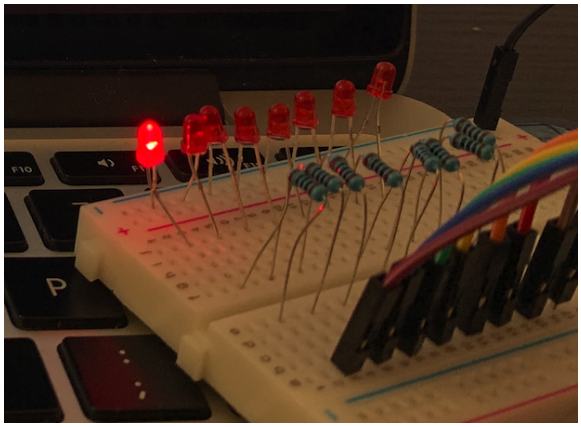
Anton Liakhovitch

Dhruv Rengarajan

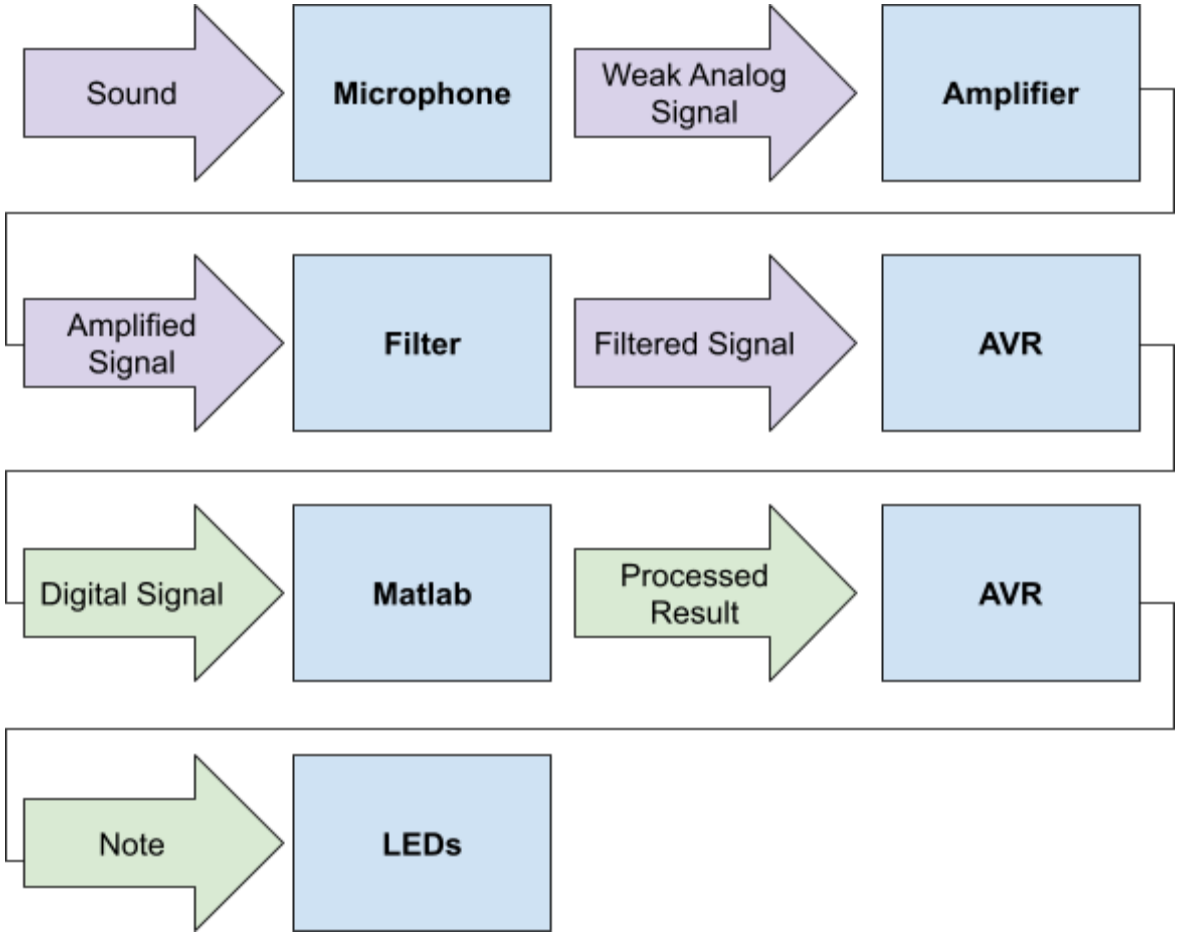
Contents

Top Level Design

The goal of the project is to create a system which can recognize musical notes. The black-box diagram below illustrates the overall function of the device. The input is a sound travelling through the air, and the output is a visual indicator LED which corresponds to the musical note recognized by the device.



There are a total of eight LEDs, corresponding to the 'white key' notes in the range C4 to C5. The picture above shows a prototype. From left to right, the LEDs represent the notes C5, B4, A4, G4, F4, E4, D4, and C4. These LEDs are connected to output pins D2 to D9 on the Arduino. Only one LED will be illuminated while a note is being played, as shown in the picture above.



The block diagram above represents the signal flow through the various components of the system. The analog components - microphone, amplifier, and filter - process the input signal into a voltage and frequency range suitable for analog-to-digital conversion. The AVR microcontroller samples this analog signal and sends it to a computer in a digital format, where it is processed. The computer continuously sends its results back to the AVR, which displays the recognized note on a set of LEDs.

A brief summary of each component's function:

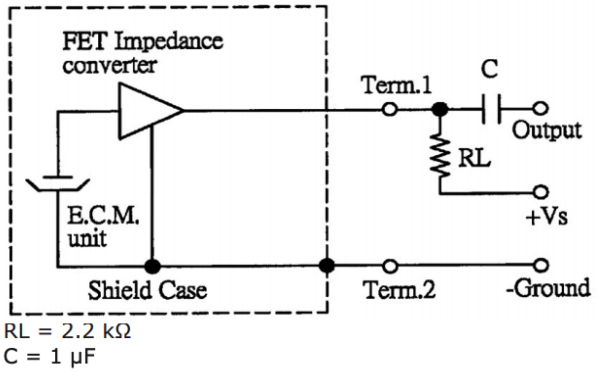
- Microphone: Sense sound and output it as a weak analog signal.
- Amplifier: Amplify the weak signal to a larger voltage range which is suitable for the AVR's analog-to-digital converter.
- Filter: Further amplify the signal, remove frequency components above and below the input range, and bias the output signal to make it readable by the AVR's ADC.
- AVR: Sample the signal and send it in digital format to Matlab, then receive an LED lighting command from Matlab and light the corresponding LED.
- Matlab: Receive the signal in digital format from the AVR, recognize the note being played, and send the AVR a command to light the corresponding LED. Matlab also displays additional analysis results once the program is stopped.

Hardware Design

Microphone

The microphone circuit was built using the electret mic provided in the AP kit, and the circuit was built using the schematic provided as a reference source for the mic as shown below. it was first soldered to connecting pins for convenient connection on the breadboard. A 20k resistor, biasing the mic to 1.4V operating at lower voltage for less distortion and power supply noise, is connected to the mic before the signal is passed through a Low pass filter using a voltage divider cutting the input voltage to 2.5V from 5V and a frequency cutoff which can be calculated as follows.

$fc1 = 12\pi(100k\Omega)(330nF) = 4.823Hz$



Amplifier

The amplifier circuit was also used with only components provided in the AP kit and is connected to a 5V power supply. The op amp used is the LMC6032, a dual amp, but only pins 5, 6, and 7 are used while following the schematic provided in the microphone amplifier reference. Different values were set for R1 (the resistor that biases the mic), R4, and C2, which is a LPF and used to set the gain for the circuit. A 100K ohm potentiometer was used as R4 to range the gain between minimal and maximum depending on user desire. The output voltage is biased to 2.7V before it goes through a HPF to set the output to 0.2V before the signal goes to our 2nd order filter circuit for further filtering and amplifying of the signal. This HPF has a cutoff frequency of 4.823Hz. There are multiple HPF and LPF through the circuit setting frequency ranges and voltage gains as shown in the diagram before the signal is passed on to the filter stage for further filtration and amplification of the AC signal, therefore high amplification with minimum noise and distortion can be achieved .

References :

LMC6032 CMOS Dual Operational Amplifier
https://www.ti.com/lit/gpn/lmc6032

Microphone Amplifier:

https://www.ti.com/lit/pdf/sboa290

Filter

The filter design for the system uses multiple op amps in series to limit the input to a proper frequency range. In the end, a band-pass filter was constructed to achieve 2nd order filtering along with a minimal amount of voltage gain.

Design

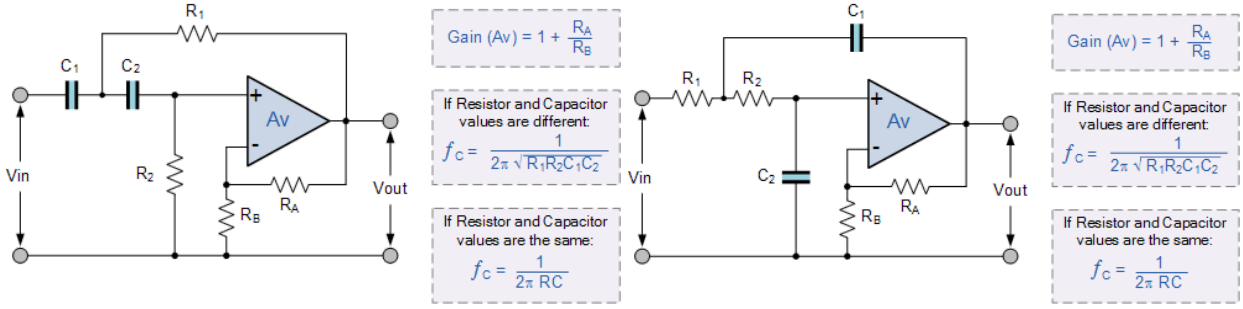
Since the design requirements list the input range as being the notes between middle C (261.626 Hz) and high C (523.251 Hz) a band-pass filter was decided on to meet the specifications. However, even though the range of notes lie between those frequencies, a separate requirement states that 3 harmonics should also be allowed through to the Arduino. This puts the input range between 261.626 Hz on the low end and 1569.753 Hz on the high end. With these frequencies known, a band-pass filter can be designed to operate within the general frequency range.

With the frequency range of the input calculated, the next task was to determine what type of filter to use. Given that operational amplifiers were provided, it was decided that they would be incorporated into the circuit design. After a little bit of research, it was discovered that the LMC6032 8-DIP operational amplifier was actually a dual operational amplifier. That is, the integrated circuit was a combination of multiple operational amplifiers sharing the same positive and negative voltage rails. This design was used to our advantage as now, instead of designing a band-pass filter using a single operational amplifier, a low-pass filter and high-pass filter could be combined in series to achieve a band-pass filter with a larger slope of decay.

With a general design decided upon, there was still the decision of what filter topology to use. The best topology for the operational amplifier provided was decided to be Sallen-Key. These amplifiers were chosen for a multitude of reasons; they have an extremely large input impedance as well as minute output impedance, the second order characteristics inherent with design was a bonus, as was their ability to be cascaded in series.

Below is pictured both the high-pass filter and low-pass filter designs used in the final system.

High-Pass Filter **Equations Used** **Low-Pass Filter** **Equations Used**



The final step in the design process for the Sallen-Key filter was to calculate the values for the elements used. Given the equations listed on the previous page, the fact that the gain resistors

A and B would be equal, as well as the fact that the filter's resistors and capacitors values would be equivalent as well simplified the arithmetic. Manipulating the equations of the previous page, we can solve for either the resistors or the capacitors in the circuit. Given that our selection of capacitors is limited, a capacitor value will be chosen that matches one of the values provided while the resistor's value will be solved for. Arbitrarily, a value of 1μF was chosen for the capacitor. With this is mind we can now rearrange the frequency cut-off equation to solve for the required resistance, as is shown below:

$$R = \frac{1}{2\pi f_c C}$$

Filter calculations

Knowing our frequency range, 261.626 Hz to 1569.753 Hz, we can plug these values into the equations above. Which, after entering the maximum frequency, gave us:

$$R = \frac{1}{2\pi * 261.626Hz * 1\mu F} = 608.33\Omega$$

This resistor value is not one provided. Plugging in the next closest resistor value found in our kit, 1kΩ, to the cut-off frequency equation listed on the previous page gave us:

$$f_c = \frac{1}{2\pi * 1k\Omega * 1\mu F} = 159.155Hz$$

This is an acceptable cut-off frequency for our high-pass filter. Similarly for the low-pass filter:

$$R = \frac{1}{2\pi * 1569.753Hz * 330nF} = 307.238\Omega$$

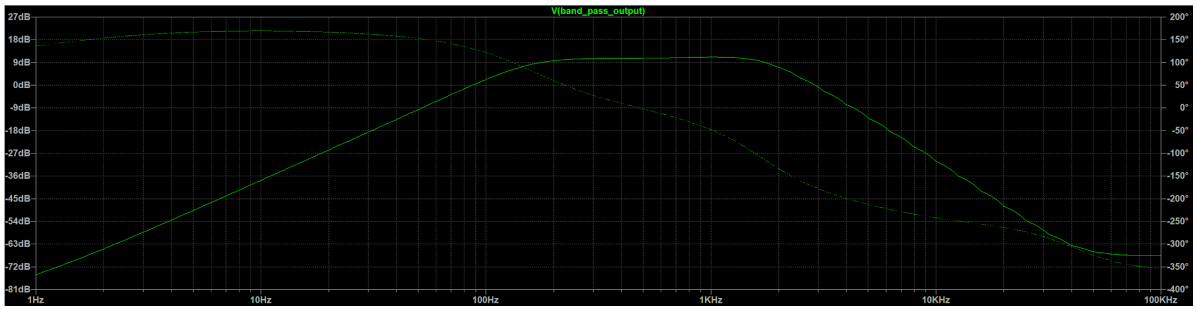
This isn't an acceptable resistor value. Smaller capacitors were provided. After switching the capacitor out with a 45nF capacitor we get this:

$$R = \frac{1}{2\pi * 1569.753Hz * 45nF} = 2253.08\Omega$$

We can change the resistor's value to 2.2kΩ since this is a resistor provided. Finally, for our low-pass filter, we get:

$$f_c = \frac{1}{2\pi * 2.2k\Omega * 45nF} = 1607.63Hz$$

The final schematic can be referred to in the design files document. Lastly, an AC signal analysis was performed to confirm the functionality of the schematic. This is shown below:


V(band_pass_output)
27dB
18dB
9dB
0dB
-9dB
-18dB
-27dB
-36dB
-45dB
-54dB
-63dB
-72dB
-81dB
200°
150°
100°
50°
0°
-50°
-100°
-150°
-200°
-250°
-300°
-350°
-400°
1Hz
10Hz
100Hz
1KHz
10KHz
100KHz

Software Design

Summary

In this design, the AVR microcontroller and Matlab PC software work together to collect data, perform analysis, and display results.

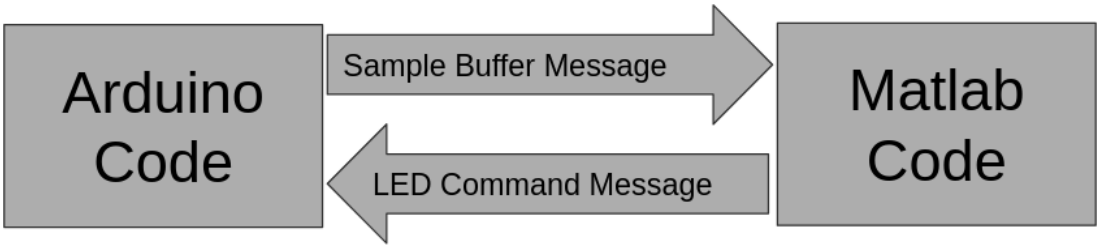
Top level algorithm:

1. AVR samples analog microphone data into a buffer
2. When the buffer is full, the AVR sends data to a Matlab script running on a PC
3. Matlab analyzes the data to determine which note is being played
4. Matlab sends the result back to the AVR
5. The AVR lights up an LED corresponding to the detected note
6. Execution returns to step 1

In step 1, the AVR collects microphone amplitude samples at a frequency of 11Khz. Instead of sending and processing these samples continuously, samples are first collected into a buffer and then processed once the buffer is full. This ensures that only step 1 of the algorithm needs to run in real time at the 11Khz sampling frequency. The other steps still need to run relatively fast for acceptable performance, but the program will continue to function if a section unexpectedly takes too long to execute.

The specification requires processing of input frequencies as low as 260Hz, and requires that at least three periods of the signal are collected at a time. The buffer size was chosen to be 170 samples, which is enough to store four full periods of a 260Hz signal when sampled at 11Khz. The 11KHz sampling frequency was chosen because it is high enough to record 20 samples in one period of the highest valid input frequency (523Hz), as required by the project specification.

Once the AVR is finished collecting data, it sends the buffer to Matlab via an RS-232 serial connection. Matlab processes the 170 sample buffer via Fourier analysis to determine which note is being played. It then sends the Arduino a message encoding the recognized note (LED Command Message), and the Arduino lights up an LED corresponding to the note.



This entire process (data collection, processing, display) repeats until the user presses a "Stop" button in the Matlab program. At this point, Matlab halts the program loop and performs final analysis steps on the last sample buffer collected.

Serial Protocol

The AVR firmware and Matlab script communicate by means of an RS-232 serial connection. A higher level protocol dictates the format of the messages between them.

There are two types of message in the protocol, the Sample Buffer Message and the LED Command Message.

Sample Buffer Message

The sample buffer message is always sent from the Arduino to the Matlab script, and it serves two purposes:

- Transfer the latest sample buffer to Matlab
- Let Matlab know that the Arduino is ready to collect more data and is waiting for a response

The message is encoded as ASCII text. It consists of the following:

1. 170 comma-separated decimals integers containing the raw values sampled from the AVR's ADC. Numbers are not padded with zeros on the left side, and there is no comma before the first number or after the last number.
2. A line-feed character to signify the end of the message.

Example message:

```
143,146,149,153,158,163,168,173,177,182,187,192,197,1,5,10,15,19,25,29,34,39,44,48,53,58,62,67
,72,77,82,86,91,96,101,105,110,115,120,125,130,134,139,144,148,153,158,163,168,172,177,182,187
,191,196,1,5,10,15,19,24,29,34,39,43,48,53,57,62,67,72,76,81,86,91,96,100,105,110,115,120,124,
129,134,139,144,148,153,158,163,167,172,177,182,187,191,196,0,5,10,14,19,24,28,34,38,43,48,53,
57,62,67,72,76,81,86,91,96,100,105,110,114,119,124,129,134,138,143,148,153,158,163,167,172,177
,182,186,191,196,0,5,10,14,19,24,28,34,38,43,48,53,57,62,67,71,76,81,86,91,95,100,105,109,114,
119,124,129,134,138,143[LF]
```

Here, [LF] represents a line feed character.

LED Command Message

The LED command message is always sent from Matlab to the Arduino, and serves two purposes:

- Tell the Arduino which LED to turn on
- Let the Arduino know that Matlab is ready to accept more data and is waiting for a Sample Buffer Message

The message is encoded as ASCII text. It consists of the following:

1. A single character representing the LED that must be lit. A number from 0 to 7 represents an LED index, while any other character is a "turn off all LEDs" command.
2. A line-feed character to signify the end of the message.

For example, "`2[LF]`" means "Light the third LED". Alternatively, "`8[LF]`" means "Turn off all LEDs".

Design Justification

Several protocol variations were initially considered, each with their own strengths and drawbacks. This particular CSV-over-ASCII approach has the issue of low efficiency. It would be possible to transmit data much faster by sending it raw, with no ascii encoding - resulting in smaller, fixed-length packets. ASCII encoding wastes data, because there are many messages which are possible but invalid. However, none of the advantages of raw data encoding are actually needed with the sample-buffering design. Because there is no hard time constraint on transmission time, it is acceptable for messages to be long and variable length. Furthermore, ASCII encoding provides the advantage of being easy to work with during both development and testing - making it the best fit for this system.

AVR Firmware

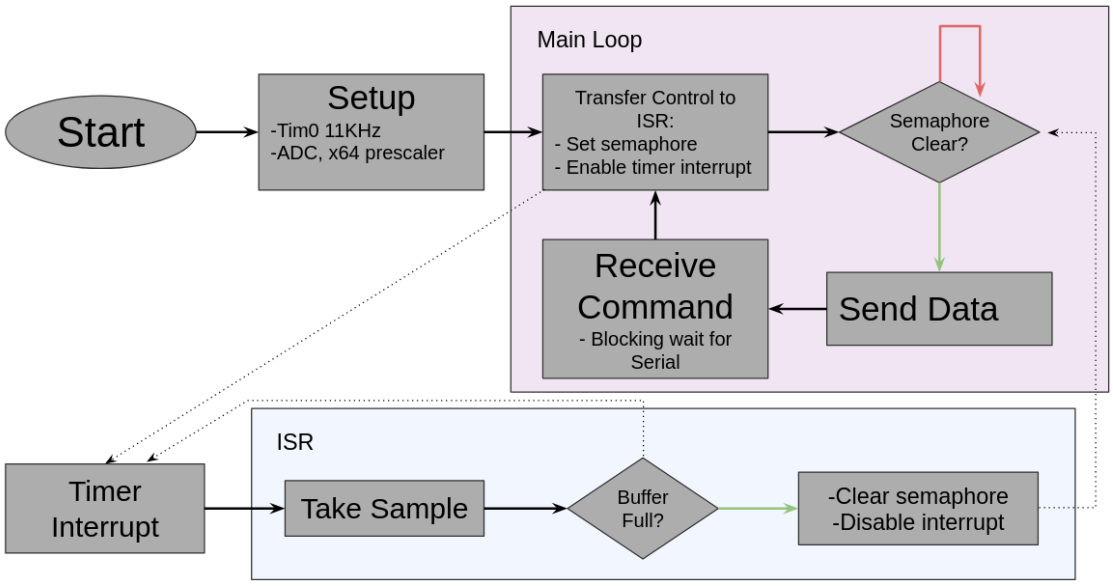
The goal of the Arduino firmware is to collect data, send it to the Matlab program, and then execute an LED command from Matlab.

Top level algorithm:

1. Collect ADC data into a buffer, in real time
2. Send data to Matlab
3. Wait for Matlab to respond with an LED command
4. Execute LED command
5. Return to step 1

Step 1 must run in real time, collecting data at a specific sampling frequency. This is achieved via the AVR's TIM0 8-bit timer peripheral, configured to trigger an interrupt at approximately 11Khz. While 11Khz is not actually possible, the timer can oscillate at 10989Hz when set to divide the 16Mhz I/O clock by 182. Every time the timer triggers, an interrupt service routine reads an ADC sample and adds it to the buffer. The ADC itself is configured with an x64 clock prescaler, which allows it to sample within the 11Khz time frame while retaining high accuracy.

The block diagram below illustrates the overall program flow.



Dark lines represent actual function calls and flow. To make this easier to read, grey dotted lines are added to represent the order in which the program normally executes when the timer interrupt is considered.

A semaphore flag and the sample buffer form a crude mutex. When the main thread reaches the "Transfer Control to ISR" step, it sets the flag to signify that the ISR may mutate the sample buffer and the main thread may not. Once the ISR has triggered enough to fill the sample buffer, it clears the flag to let the main thread know that it can take control of the sample buffer resource.

After the ISR returns control to the main thread, the program is no longer bound by the real-time constraint of the sampling frequency. In other words - the rest of the main loop simply needs to execute quickly (on a human scale), but does not need a guaranteed execution time. Thus, the "Send Data" portion of the protocol can afford to spend time converting the acquired data to ASCII and sending it in CSV format.

The "Receive Command" section simply waits until data is available in the serial buffer, then interprets the first available ASCII character as an LED command and discards the rest. After lighting the requested LED, the program again returns control to the ISR and starts the main loop over again.

The resulting firmware collects data quickly and reliably, while being easy to test and troubleshoot through a simple ASCII RS-232 console.

Matlab

Matlab performs several functions in this design. These include reading in data from the Arduino, performing graphical analysis, and computing specific values required by the specification such as Signal-to-noise ratio and frequency distribution.

This can be summarized in the below algorithm:

1. Send Arduino an LED command as a signal that it is ready to receive data
2. Receive the sample buffer
3. Processes buffer using a fast Fourier transform
4. Extract the fundamental frequency from the Fourier transform output
5. Match recognized frequency with a constant corresponding to a musical note
6. Prepare to send the recognized note to the Arduino as the next LED command, and return to step 1

This process starts with Matlab sending the AVR an LED message to signal that Matlab is ready to accept data. The Arduino Nano sends a buffer of data composed of ASCII text. Each buffer contains 170 decimal numbers representing samples of data. Matlab is responsible for reading through a full line of the buffer, only stopping once an end-of-line character is detected. A serial object reads a line of the buffer, splitting it using comma delimiters and storing it in an array.

Once the serial driver has parsed through the buffer, the program performs a Fourier transform using the built-in Matlab function. The software calculates the magnitudes of the Fourier outputs (discarding the phase component) and interprets the largest magnitude frequency bin as the base frequency. Next, the resulting frequency is matched to the frequency of the closest musical note. Finally, the number of the LED it corresponds to is noted down, and sent back to the Arduino through the serial connection. This process repeats until the user presses a "stop" button. At this point the program closes the serial connection, and the values stored in the buffer are used to calculate the SNR and confirm that it is above 20. The magnitude versus time and magnitude versus frequency graphs are created, the latter with the appropriate harmonics.

The process of receiving data, transforming it, matching it and then sending it runs in a constant loop. However, in order to exit the loop and graph the current data, a special button was added to the UI window in Matlab that does exactly that. When pressed, the button causes the program to break out of the main loop and move on to the graphing functions. Therefore, the loop will run indefinitely until the user presses the button, and the data currently stored in the buffer is graphed.

Results

260Hz and 520Hz signal input

We tested our circuit by getting the data through pin A0 on the Arduino nano and using our software design, the mic was tested using multiple frequencies to make sure that it is working to the requirements specified. And as expected, the results were to standards required. First test was while playing a continuous Signal of 260 Hz to make sure that the corresponding LED which is LED 0 was turning on. The sine wave did not have any distortions with Peak to PEak output of almost 3V, and the SNR was indeed above 20 and was registered at 28.33 as shown in Figure 1. The second test conducted was using a continuous 520Hz signal, and the result was an undistorted sine wave with 2Vpp and LED 7 was lighting up with an SNR value of 38.86 as shown in figure 2 .

As stated above and shown in the result figures, while stimulated, the signal output was within the required voltage range having more than 1 volt of amplitude, and having an SNR above 20.

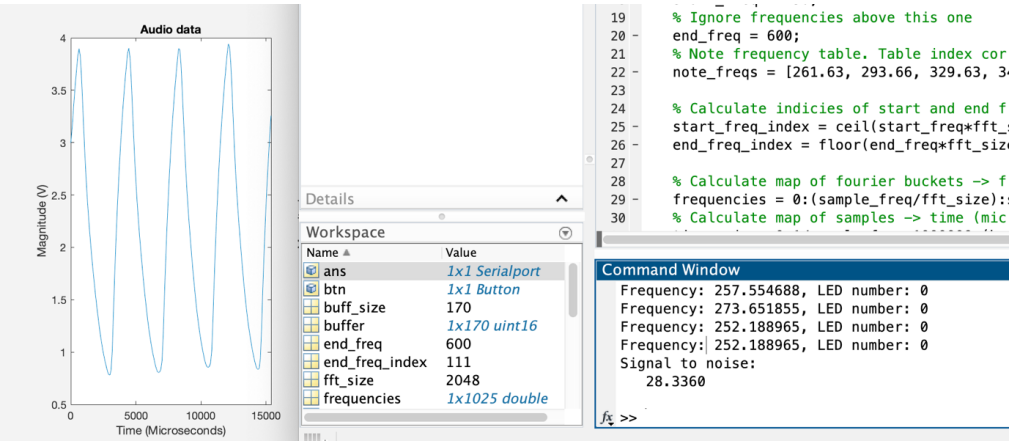
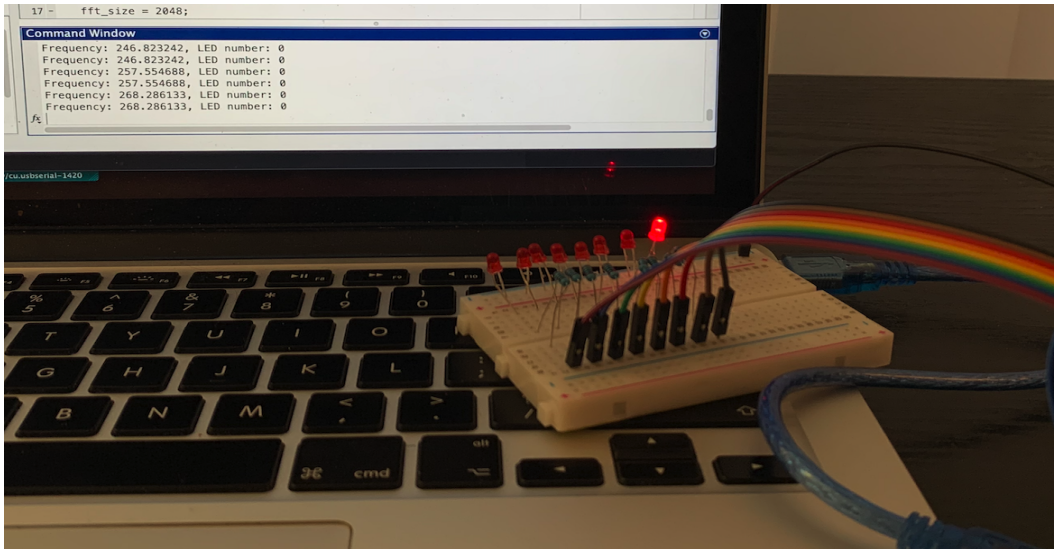


Figure 1



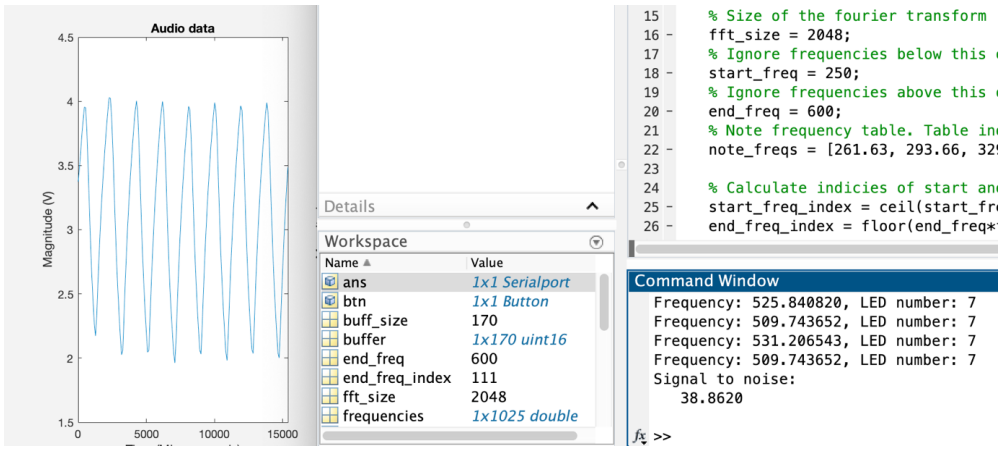
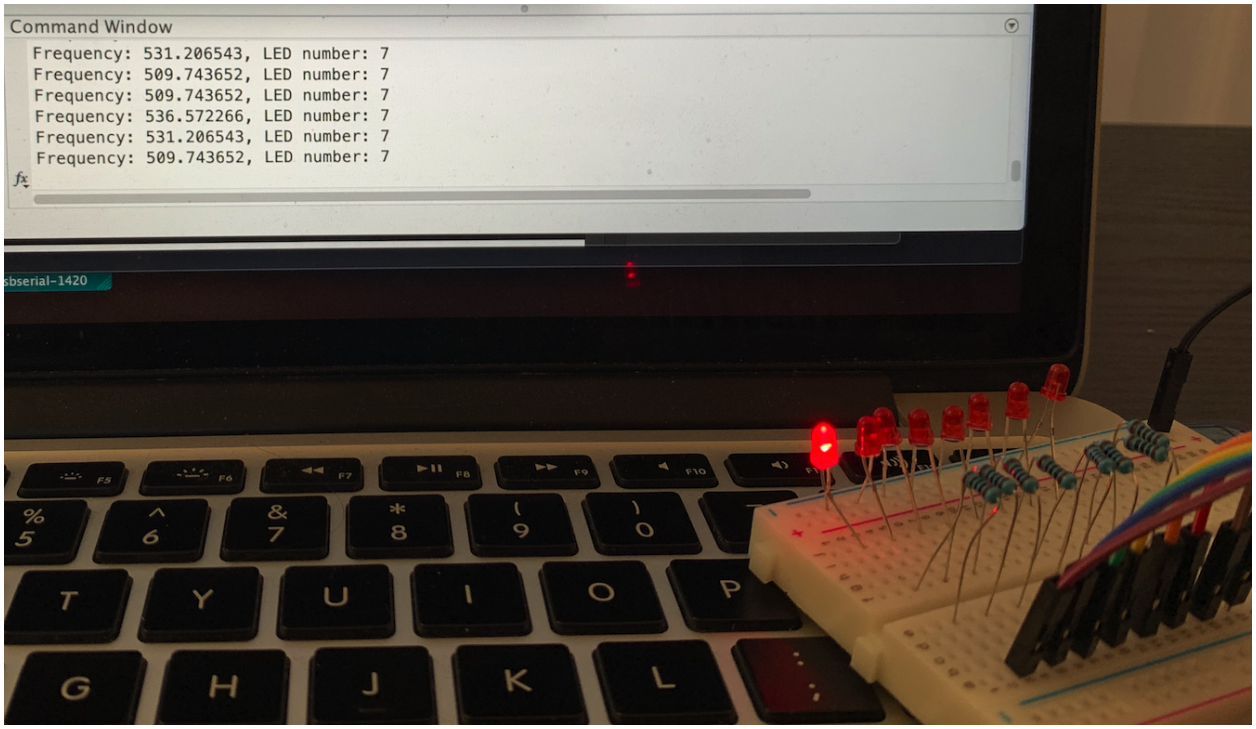

Audio data
Magnitude (V)
Details
Workspace
Name
Value
ans
1x1 Serialport
btn
1x1 Button
buff_size
170
buffer
1x170 uint16
end_freq
600
end_freq_index
111
fft_size
2048
frequencies
1x1025 double
% Size of the fourier transform
fft_size = 2048;
% Ignore frequencies below this
start_freq = 250;
% Ignore frequencies above this
end_freq = 600;
% Note frequency table. Table in
note_freqs = [261.63, 293.66, 32
% Calculate indicies of start an
start_freq_index = ceil(start_fr
end_freq_index = floor(end_freq*
Command Window
Frequency: 525.840820, LED number: 7
Frequency: 509.743652, LED number: 7
Frequency: 531.206543, LED number: 7
Frequency: 509.743652, LED number: 7
Signal to noise:
38.8620
>>

Figure 2


Command Window
Frequency: 531.206543, LED number: 7
Frequency: 509.743652, LED number: 7
Frequency: 509.743652, LED number: 7
Frequency: 536.572266, LED number: 7
Frequency: 531.206543, LED number: 7
Frequency: 509.743652, LED number: 7

No signal input

The last test was with no signal input, where the mic was not stimulated and the signal output had about 0.2Vpp magnitude and SNR value of 25.64 as shown in figure 3. These results also comply with the project requirements where it states that if no signal was played as input or mic was not stimulated, then output must have less than 0.2V amplitude while also having an SNR of more than 20 for all cases.