Lab 1: Audio Filter

# Introduction

As my first Arduino lab experiment I attempted to design an audio filter. The learning objective in these labs is how I can use microcontrollers to sample, process, and output processed data. This is relevant to my field where we are collecting large amounts of data from industrial machines. Sometimes it’s beneficial to reduce the amount of data if it is not necessary. I am simulating this data using an audio signal and processing with a filtering library. I spent quite a bit of time struggling with this experiment due to hardware constraints and ended up having to modify the experiment some. There are four main tasks in creating the filter, audio input, audio sampling, data processing, and audio output.

# Audio Input

One of the greatest challenges with this lab was preprocessing the audio to prepare it for sampling. This must be done with hardware. I was able to find some similar projects as Instructables by Amanda Ghassaei, which I used as a base for my own projects [2].

First I had to understand how audio is represented as an electrical signal. Using an oscilloscope I was able to view the electrical signal of audio. Audio is represented by a low voltage, approximately 200mV depending on the source, A/C signal oscillating around 0V. This immediately creates an issue with the Arduino analog to digital converter, or ADC. The Arduino ADC reads voltages between zero and its source voltage, 5V. Immediately we are losing half of the information as the audio signal oscillates below 0V. Additionally, the resolution of the reading will be quite low. Considering the value read will be converted to a digital value between 0 and 1023 a signal around 200mV would be converted to the value 41. In summary, without preprocessing the signal we would sample half of the signal with digital values ranging from 0-41.

The first step is to apply gain to the signal. Ideally, I want the signal to swing across the full voltage spectrum, from 0 to 5V. In this first step I want the signal to swing 5V while still oscillating around 0V, from 2.5V to -2.5V. I do this with an operational amplifier. Operational amplifiers are integrated circuits that take in a signal and output a multiple of that signal depending a resistor configuration. In this case I have a signal with an amplitude of 400mV (+/- 200mV) so I want a gain of 25.

After I have a signal oscillating from 2.5V to -2.5V I need to offset the voltage to oscillate around 2.5V. This way I capture all data with the ADC. This is done simply with a voltage divider resistor combination and a capacitor. I tied the output of the op amp to the positive terminal of a capacitor and the negative end to the voltage divider. The voltage divider is a very common way to get a specific voltage as a fraction of the source voltage. A voltage divider is two or more resistors in series to ground. The output voltage from the divider is calculated by

My source voltage was 5V so I use two 100K Ohm resistors in series to ground. After the first resistor the voltage has dropped to 5V\*100K/200K or 2.5V, which was my goal output. Now the signal oscillates around 2.5V and has an amplitude of 5V, so it is ready for sampling.

# Sampling and Filtering

Sampling on the Arduino from the ADC is relatively straightforward. An analog pin can be read using the analogRead(pin\_num) command. As I stated earlier, the value read will be a value between 0 and 1023. The value 1023 is represented with a 10 bit binary value 1111111111B. Looking ahead I’ll be outputting the data using the digital out pins to an external 8-bit Digital to analog converter, so at this point I’ll reduce the data size to 8 bits. The highest value that can be represented by 8 bits is 255 so I can simply divide the sampled value by 4 to get within this range.

Filtering is done directly on the value at the analog input pin. I used a filter library from Arduino playground contributor JonHub. To use these filters I first create an instance of a one-pole filter with the keyword for the type of filter and a frequency to apply the filter at. Rather than hard coding a frequency into the filter I decided to control the frequency with a potentiometer. A potentiometer is a great example of a voltage divider; it is essentially a variable resistor that can be adjusted by a knurled knob. I translate the result of the voltage division by the potentiometer to a value between 0 and 4800Hz. I used 4800Hz because the sampling rate of a single analog input is 9600Hz; therefore, isn’t possible to convert signals higher than 4800Hz. This is known as the Nyquist theorem, which states the sampling rate must be at least twice the highest analog frequency component [4]. Attempting to convert signals faster than 4800Hz will yield a distortion known as aliasing. Generally, humans can hear frequencies from 20Hz to ~20000Hz [3]. 9600Hz is well within the hearing range for a human so I can prove my filter is working simply by listening to the output.

I created three different filtering sketches for this lab: lowpass, bandpass, and highpass. The lowpass and high pass filters are quite similar. I use one filter and declare a cutoff frequency using the potentiometer reading. In the lowpass filter all frequencies below the cutoff frequency come through while any frequencies higher are attenuated. The highpass is just the opposite. All frequencies above the cutoff frequency pass while the lower frequencies are attenuated. The band pass allows a band of frequencies to pass through so I use two filters: a highpass and a lowpass. I also use two potentiometers, one to control the center band frequency and another to control the band width. The following two commands set these values:

FilterOnePole lowpassFilter(LOWPASS, (frequency + bandwidth/2));

FilterOnePole highpassFilter(HIGHPASS, (frequency - bandwidth/2));

The frequency read by the analog input 1 is the center frequency. I offset the cutoff frequency of each the lowpass and highpass filter from that center frequency and half of the bandwidth read on analog pin 2. In order to pass a band I first filter the incoming signal with a lowpass filter, removing high frequency content and the cutoff frequency. Then I take the output from that filter and use it as the input to the highpass frequency, which has a lower cutoff than the lowpass filter, did. That effectively attenuates low frequencies leaving only the center band of frequencies present:

incomingAudio = analogRead(A0)/4 - 127;//filter to include the negative component of audio signal

lowpassFilter.input(incomingAudio);//lowpass

highpassFilter.input(lowpassFilter.output()); //filter the lowpassed signal

outgoingAudio = highpassFilter.output() + 127;//final output is a bandpass, add back 127 for output to DAC

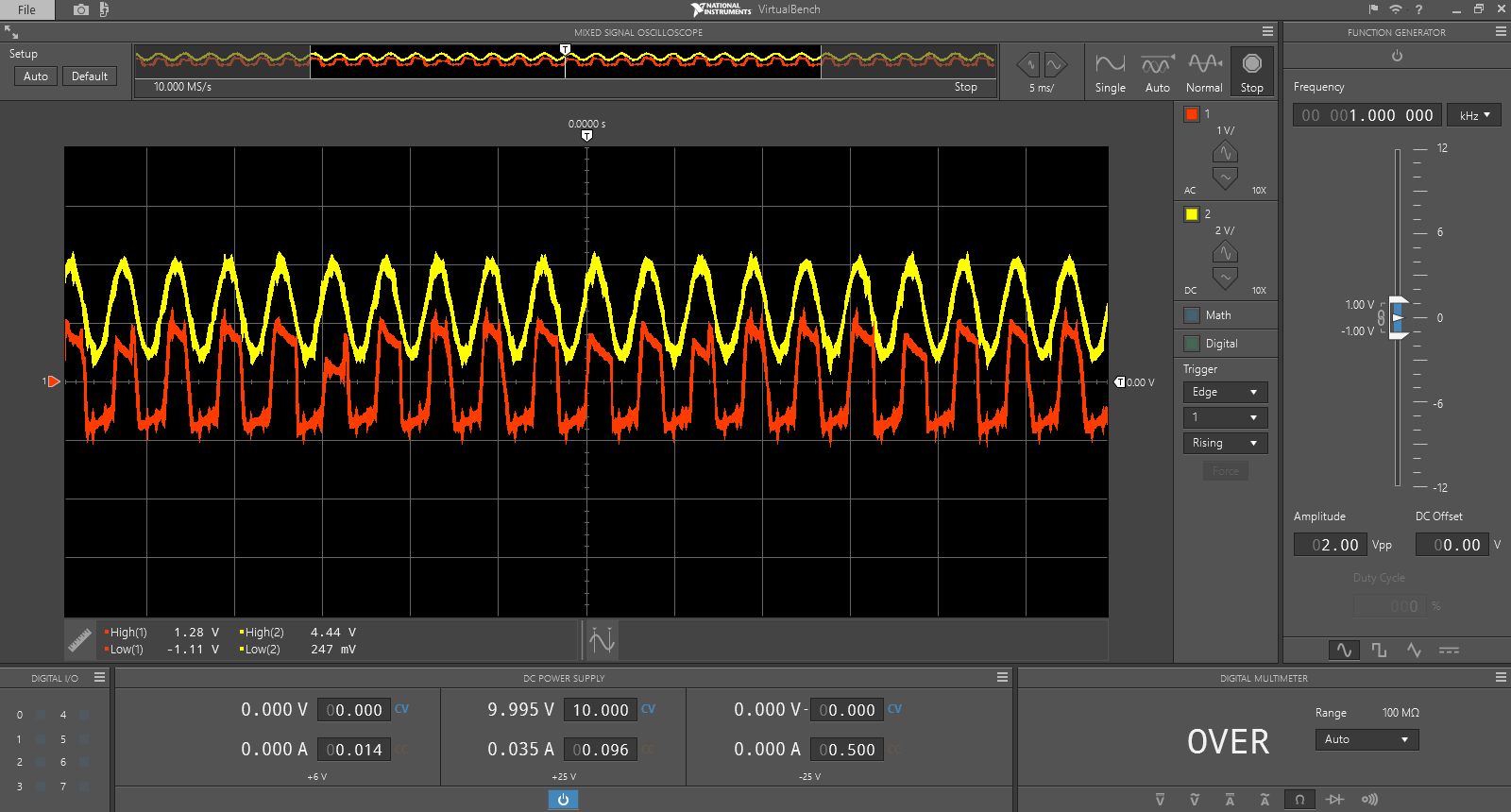
# Audio Output

Outputting the audio is just as tricky as inputting the audio. It’s a very similar process; however, now I have a digital signal that I want to turn into an analog audio signal. The conversion requires the use of a digital to analog converter. DACs can be created using a resistor ladder; however, there are also integrated circuits created for this specific purpose. I purchased one from Texas instruments because they offer better performance than something I could create on my own.

After the DAC is a series of operational amplifiers. The first two have the purpose of smoothing the output of the DAC [2]. Since the Arduino is outputting a value that holds until the next value is written. This results in the signal appearing as a series of steps with the length at the Arduino’s write frequency. The first operational amplifier acts as a low pass filter to smooth out these steps. The next op amp is configured as a buffer to protect the filter from loads downstream [2]. The final two operational amplifiers are configured as voltage followers. Voltage followers do not change the signal, but just increase the available current to drive a speaker. Finally, I offset the value again so it oscillates around 0V and include a potentiometer to control the volume, which is just adjusting the amplitude of the final signal.

# Results

As a test I used a tone generator to generate a pure sine wave to use as an input to my device. A sine wave is the easiest signal to visualize on an oscilloscope as well as detect distortion. Due to my hardware there was a considerable amount of distortion introduced to the signal. I’ve included an oscilloscope screenshot to demonstrate this.



The yellow signal is the output of the DAC from the input of a pure sine wave. As you can already see there is a bit of distortion from the sampling process. The voltage of the yellow signal is 4.44V to 247mV, which is well within the range of 0-5V. The red signal is the output after filtering and the DC offset. The red signal follows the yellow signal; however, it is evident the hardware degraded the signal.

The degradation manifests itself as a harsher sound than the pure sine wave with additional harmonics. This is where I started to notice hardware limiting my progress. Just creating a sketch that passes through audio from the analog in, to the digital out, without any processing, was already heavily distorted by the sampling process. I noticed this also created some interesting results of the filter. The most interesting was that after several attempts and code changes, the filtering library flat out didn’t work for this sampled signal. I was able to run through the filter example with good results; however, it was a very contrived example. The example given generates a sine wave in the code and runs its filtering algorithm on it. The issue I found is filtering operations involve heavy floating-point math, which the Arduino, specifically the Uno, isn’t very effective calculating. The Arduino has an 8-bit processor, while floating point numbers are 32 bit. Additionally, the Arduino lacks a FPU, or floating-point unit, which is a dedicated portion of the processor to deal with floating point numbers [1]. Between sampling the input, filtering it, and outputting it to the digital out pins the Arduino just couldn’t keep up to be effective. I’ve included a video in which I attempt a filtering operation and try to output audio in a loop to no avail. I didn’t even use the filtered output to source the audio, but merely having the operation as part of the loop causes major issues. After this lesson I decided to pivot my goal of filtering to something more achievable and moved on to less math intensive audio effects such as phasing, reverb, and delay.

**Works Cited**

1. [2] Atmel 8-Bit Microcontroller with 4/8/16/32KBytes In-System Programmable Flash Datasheet(J ed.). (2015). Atmel.
2. Ghassaei, A. (n.d.). Arduino Vocal Effects Box. Retrieved October 15, 2015, from <http://www.instructables.com/id/Arduino-Vocal-Effects-Box/>
3. Hearing range. (2015, December 13). In Wikipedia, The Free Encyclopedia. Retrieved 04:36, December 15, 2015, from <https://en.wikipedia.org/w/index.php?title=Hearing_range&oldid=695093778>
4. Nyquist–Shannon sampling theorem. (2015, December 1). In Wikipedia, The Free Encyclopedia. Retrieved 04:35, December 15, 2015, from <https://en.wikipedia.org/w/index.php?title=Nyquist%E2%80%93Shannon_sampling_theorem&oldid=693311115>