ECE 167/L: Sensing and Sensor Technologies
Lab 1 Report

Jose Santiago

Table of Contents

1. Lab Introduction and Overview 1
2. Flex Sensor 1
3. Piezoelectric Sensor 7
4. Musical Instrument Redux 8
5. Simple Analog Filtering Analysis 11
6. Conclusion 17

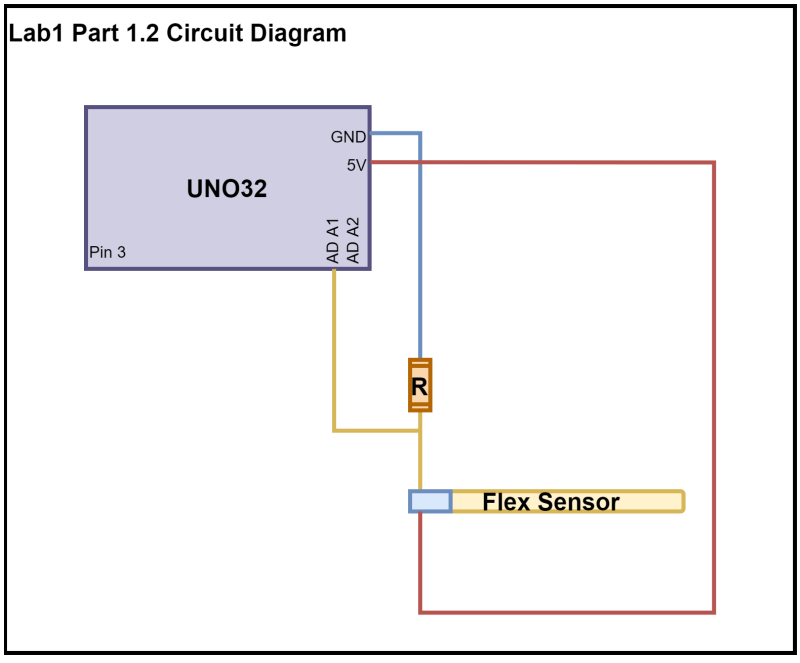
Lab Introduction and Overview

This lab serves as an introduction to resistive sensors. The two sensor types used in this lab are the flex sensor and the piezoelectric sensor. The flex sensor is a sensor that changes resistance depending on the amount of bend on the sensor. The change in the bend of the flex sensor is converted to electrical resistance. The more the sensor is bent, the greater the resistance. Piezoelectric sensors can measure deflection, acceleration, or vibration. The piezo sensor in this lab is a time based analog signal that must be captured. Every tap generates a voltage spike that needs to be snubbed to prevent damage to the UNO32. Both the flex and piezo sensors will be used to implement a musical instrument. The flex sensor will determine the tone of the instrument, while the piezo sensor will trigger the tone. The necessary hardware components for this part of the lab are: the UNO32, the speaker, the audio amp, a breadboard, the flex sensor, the piezo tap sensor, and a couple resistors.

This lab also has a simple analog filtering analysis portion aside from the musical instrument implementaiton. In this part of the lab we analyze low-pass, high-pass, and band-pass filters. We analyze them by deriving their transfer functions. We then experimentally validate our analog filtering. The necessary hardware components for this portion of this lab are: resistors, capacitors, and the MCP004 quad OpAmp.

Flex Sensor

The flex sensor needs to be assembled before use. The flex sensor is paired with a blue clinch provided in the ECE 167 lab kit. To assemble the flex sensor using the blue clinch, follow the instructions provided by sparkfun, or the instructions found in the Lab1 folder in the canvas files for ECE 167. The metal pins sticking out of the flex sensor must be trimmed. The only parts to be trimmed are the metal pins, be careful not to trim anything else. Then the flex sensor must be inserted into the blue clinch. The metal pincers of the blue clinch must align with the perforated holes encased in metal. Once they are aligned you must apply significant force onto the blue clinch flap to snap the blue clinch and flex sensor together. You may want to use pliers if you have them for this part.

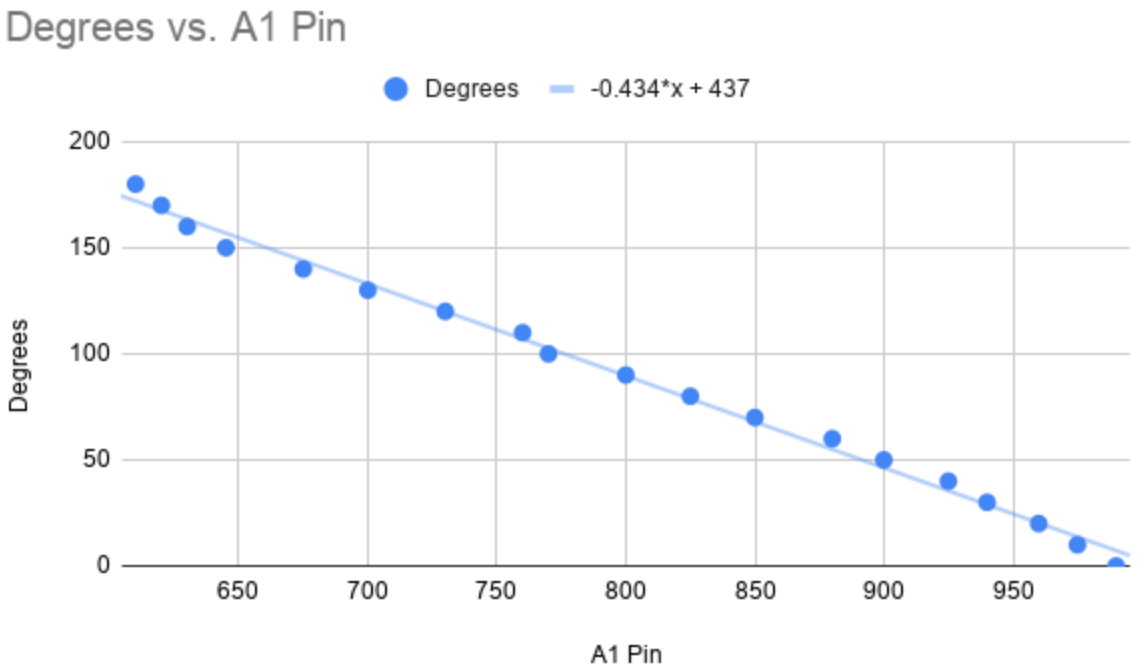


After successfully assembling the flex sensor, we can begin linearizing it. First we wire the flex sensor on a breadboard with a resistor to create a voltage divider circuit. Connect the flex sensor and resistor in series. The flex sensor used in this lab report is a 47kOhm resistor. The node in between the resistor and flex sensor is the node that will supply us the flex sensor reading. Wire the free pin of the flex sensor to the PWR rail on the breadboard and wire the free pin of the 47kOhm resistor to the GND rail of the breadboard. Use a jumper wire to connect the AD A1 pin of the UNO32 to the node between the resistor and the flex sensor. Finally, use jumper wires to connect the PWR rail of the breadboard to the 5V pin on the UNO32 and connect the GND rail of the breadboard to the GND pin on the UNO32. This should conclude the setup for the voltage divider circuit, allowing you to read from the flex sensor via the AD A1 pin on the UNO32. The circuit diagram for this part can be seen above (Lab1 Part 1.2 Circuit Diagram).

Now that the circuit is ready, we can read the AD A1 pin to linearize the flex sensor. The libraries necessary for this are the AD, BOARD, and timers libraries. Include these header files, and initialize the separate components using their respective initialization functions found in the header files. Add the AD A1 pin using the AD_AddPins() function provided in the AD.h file to enable reading from the AD A1 pin. Create a time variable (t), flex sensor reading variable (f), and degree conversion variable (d). The time variable will keep track of time. The flex sensor reading variable will store the reading from the flex sensor. The degree conversion variable stores the conversion of the flex sensor reading to the corresponding angle of bend of the flex sensor in degrees.

Initially the degree variable is unused, since we must first map degree readings measured with a protractor to the flex sensor readings depicted by the AD A1 pin. You can use Google Sheets to keep track of the degrees the flex sensor is bent and the corresponding flex sensor reading. Google Sheets can also be used to linearize this data when the data points are recorded. In this lab report we use 19 data points to measure the flex sensor readings. We bend the flex sensor from 0 to 180 degrees in increments of 10 degrees and record the corresponding flex sensor readings in Google Sheets.

Within an infinite loop, store the time into the t variable using TIMERS_GetMilliSeconds() before anything else. The time is important because the flex sensor readings are noisy. We pause the program for a second at the bottom of our infinite loop to reliably read the flex sensor value with the given angle of bend. After storing time we check to see if any data is ready to read from the AD A1 pin using an if statement in combination with the AD_IsNewDataReady() function provided in the AD.h file. If new data is ready, we store the value from the AD_A1 pin into the f variable with the AD_ReadADPin() function provided in the AD.h file. Outside of our if loop we can print the data of interest using printf() statements. We print the flex sensor value and the wait time to the terminal. Using CoolTerm to establish a terminal connection we can receive the data we print here. After our print statements, we wait for one second. Using the t variable, which is storing milliseconds, we can count seconds using modulus. When t modulus 1000 is zero, a second has passed. So we use a while loop to loop until a second has passed, updating the t variable within the while loop during every iteration. This way we have successfully set up a method for reading data from the flex sensor without too much noise. This concludes the programming portion for the linearization.



After this we must establish a serial connection using CoolTerm and program our device to begin sampling the flex sensor. We sample the flex sensor from a bend angle of 0 degrees to 180 degrees. Once all 19 data points have been recorded in Google Sheets we can use the insert chart function in Google Sheets to turn the data into a plot. Highlight the columns of data and click on the Insert menu on the toolbar of Google Sheets. Click on Chart. Make sure the degrees are mapped onto the y axis and the flex sensor readings are mapped to the x axis. We now have a scatter plot of our data. We can use the Customize tab to find a trend line that best fits our data points. Under the Customize tab there is a Series tab where you can check a box labeled Trendline, to find a trend line that best fits the data. Then there are other options for the trend line, such as adding a Label to the chart. This option opens a drop down menu where you can find the option “equation.” Select this to find the equation for this trend line. This is the equation used to map the flex sensor reading to degrees. In this lab report the trend line we found is described by the following equation : -0.434*x + 437, where x is the flex sensor reading. The data plot used for this linearization can be seen above (Degrees vs. A1 Pin).

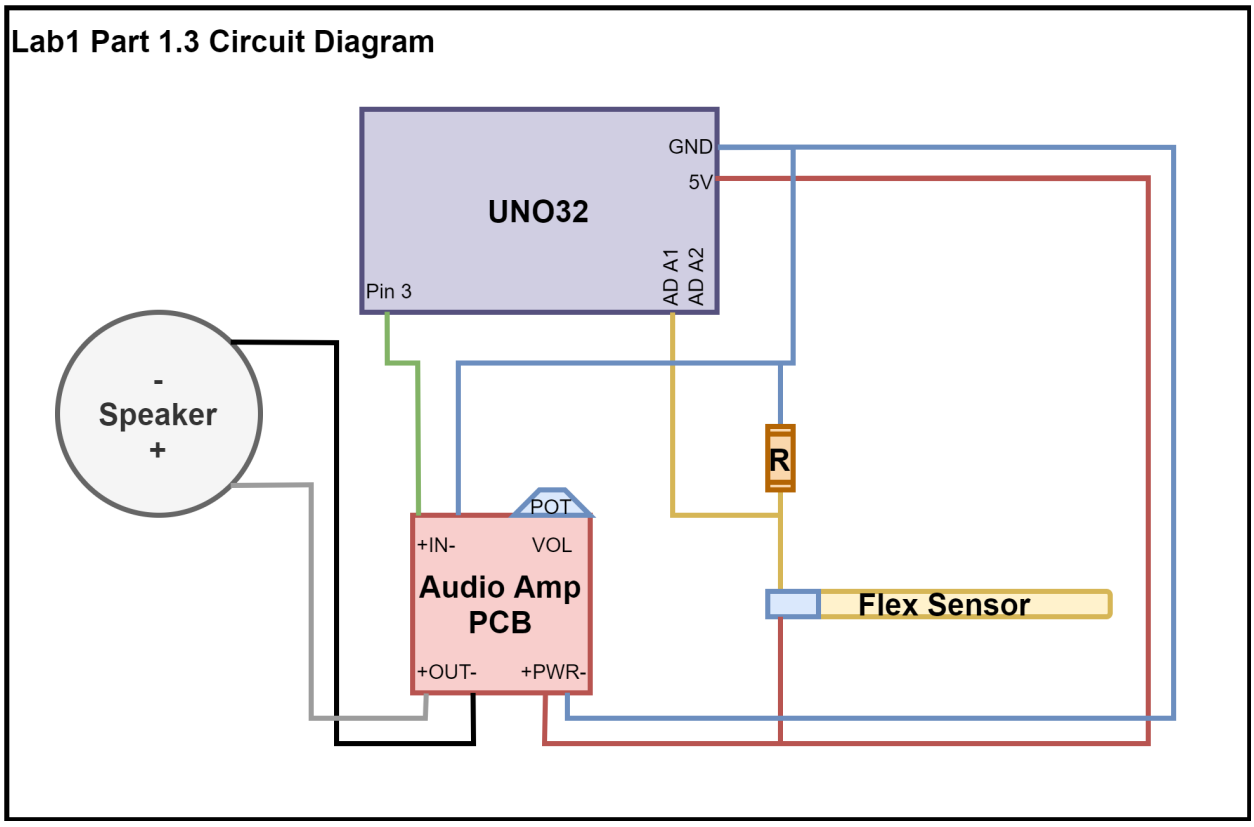
We can use this equation to convert the flex sensor reading to degrees. We can now add this conversion to our code, and print out the degrees corresponding to the flex sensor reading to the CoolTerm window. Add a line after recording the flex sensor reading where we store the conversion from flex sensor reading to degrees into our degree variable. Programming our device, our terminal should now print out flex sensor readings and the corresponding angle of bend in degrees of the flex sensor. This concludes part 1.2 of the lab.

Pseudocode:

```
int main(void){
   //initialize BOARD, AD and TIMERS
   INIT_ALL();

   //Add AD_A1 pin to read form
   AD_AddPins(AD_A1);

   //infinite loop
   while(1){
      t = TIME; //store time in t

      //check flex sensor reading
      if(AD_NewDataReady()){
         f = AD_A1(); //read from AD_A1 pin
      }

      //convert flex sensor to degrees
      d = (-0.434*f) + 437;

      //print data to CoolTerm
      print("flex sensor value: ",f);
      print("degrees: ", d);

      //wait a second
      while(t != 1_sec){
         t = TIME;
      }
   }
}
```

The next part of this lab is to create a tone with a frequency depending on the reading of the flex sensor. To do this we must wire the Audio Amp PCB board, speaker, and potentiometer to the breadboard. We did this in Lab0, so we will not go over the wiring process for this. All we need to know about how to hook up the circuit is depicted in the Lab1 Part 1.3 Circuit Diagram below.

The first thing we should add to our code is a way to map the flex sensor angle of bend in degrees to the frequency range, 1 to 1000Hz. Before this we will modify the code from the previous part slightly. We will be deleting our print statements to our serial connection and instead print our data to the OLED. This means we must include the OLED.h header file in our main program. We will also be including the ToneGeneration.h header file to enable tone generation through Pin 3 on the UNO32. We will be using the three libraries from the previous part again, BOARD.h, AD.h, and timers.h. We first initialize all the components at the top of our



main function. Then we add the AD A1 pin to read flex sensor values. Immediately after we turn the tone generation on. We then read the values from the flex sensor and store them in our variable, f. We convert the flex sensor reading to degrees as described in the previous section, using our linearization equation.

We can then map the degrees to frequency with a new variable, h. Since our degree linearization ranges from 0 to 180 degrees, we can use an equation similar to one we used in Lab0 to convert our linearized degrees to frequency in Hertz as follows:

h = ((d*999)/180 + 1)

Then we can use this frequency in Hertz stored in variable h to perform some software filtering.

The software filtering used here is identical to the software filtering used in Lab0. We define an int array of size 100 (h_buf) to store 100 different Hertz values, h. These values are averaged to filter the different frequencies converted from the flex sensor readings. Immediately after this we check to make sure the average is not greater than our ceiling 1000Hz, or smaller than the floor 1Hz. If they are, we set the current average equal to the max (1000Hz) or minimum (1Hz), respectively.

This software filtering wasn't enough as the flex sensor readings are very noisy. An additional software filtering must be added to output a stable tone from the speaker. This filtering compares the previous average from the Hertz buffer to the current average from the Hertz buffer. If the absolute value of the difference between the two is not greater than 5, then we define the current average as the previous average. This way the readings from our flex sensor are not taken into account until the deviation from the previous average Hertz conversion

is significant enough (>5Hz). This eliminates the constant noise, making the frequency more stable.

After the software filtering we can set the frequency of the tone by using the current average of the Hertz buffer. Following this we can print our data pertinent to this part to the OLED. We print the flex sensor value, corresponding degrees, hertz, and seconds passed. This is useful for debugging purposes and for knowing what is going on in the system. At the bottom of main we wait 25 milliseconds before starting another iteration of our infinite while loop. This concludes the programming portion of Part 1.3.

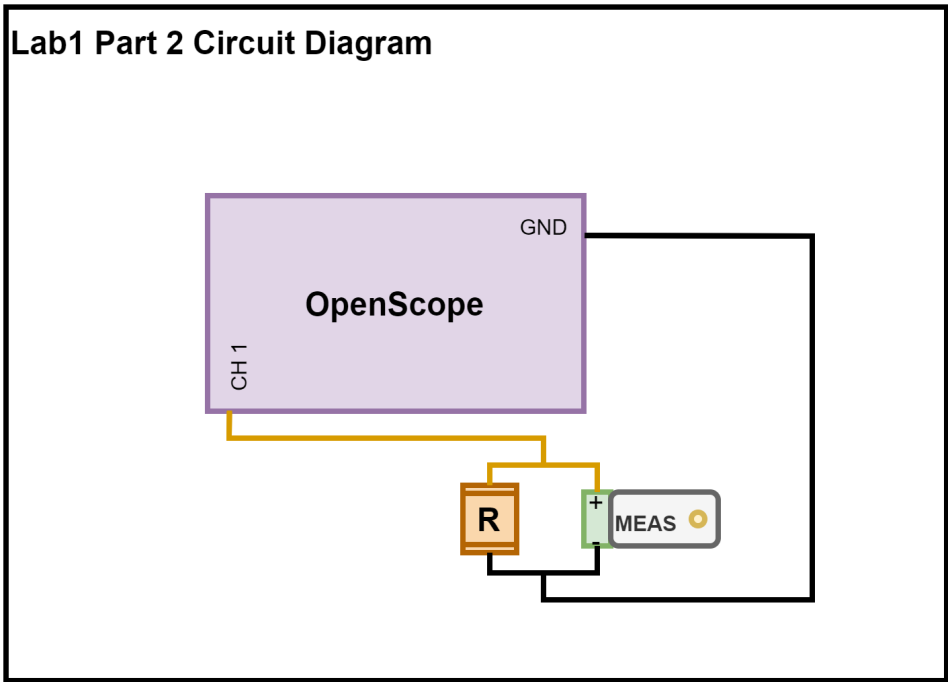
Pseudocode:

```
int main(void){
   //initialize BOARD, AD ,TIMERS, ToneGen, & OLED
   INIT_ALL();

   //Add AD_A1 pin to read from & turn tone on
   AD_AddPins(AD_A1);
   ToneON();

   //infinite loop
   while(1){
      t = TIME; //store time in t

      //check flex sensor reading
      if(AD_NewDataReady()){
         f = AD_A1(); //read from AD_A1 pin
      }

      //convert flex sensor to degrees & to Hz
      d = (-0.434*f) + 437;
      h = ((d*999)/180) + 1;

      //software filtering
      h_buff[h_i%100] = h; //store current hz
      h_i++;
      for(i = 0; i < 100; i++){ //take the average
         avg = avg + h_buff[i];
      }
      avg = avg/100;

      //check for Hz ceiling and floor
      if(avg > 1000){ avg = 1000; }
      if(avg < 1){ avg = 1; }
```

```
      //more software filtering
      if(abs(avg - p_avg) < 5){
         avg = p_avg;
      }
      p_avg = avg;

      ToneGeneration_Frequency(avg);

      //wait 25 milliseconds
      while(t != 25_milliseconds){
         t = TIME;
      }
   }
}
```

Piezoelectric Sensor

To use the piezoelectric sensor we first must understand the magnitude of the voltage spikes that this sensor can generate. To do this we read values from the OpenScope. We must use a snubbing circuit to ensure the piezoelectric sensor does not damage the UNO32 AD pin. The snubbing circuit is a simple parallel circuit with the piezo sensor parallel to a 1MegaOhm resistor. The negative pin of the piezo sensor is grounded along with one end of the resistor. The positive pin of the piezo sensor is connected to the other end of the resistor. This node is also connected to the OpenScope Channel 1 jumper wire. This way we can read voltage spikes as desired for this part of the lab. The circuit diagram can be seen below.

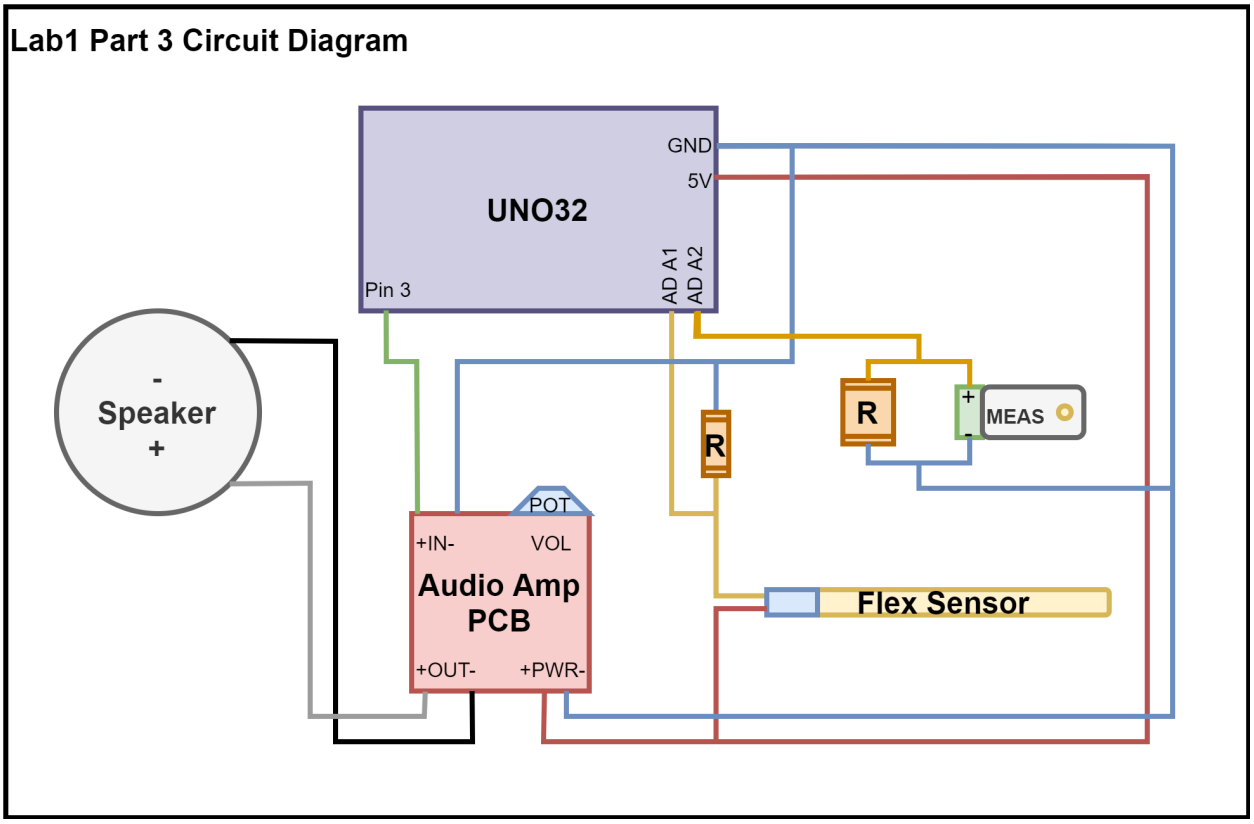


Now that the circuit is set up, we can start sampling the voltage spikes from tapping the piezo sensor. We need to make sure to record the peak-to-peak voltage as we are interested in the voltage spike. We tap the piezo sensor with varying force to get a wide array of voltages. We will be taking 20 data points to find the average, maximum, and minimum voltage spike of the piezo sensor. In this lab we report an average of 8.52V, a minimum of 1.32V, and a maximum of 22.9V. The data points recorded are shown in the table below.

P-P Voltage
10.53
22.76
2.61
1.323
2.375
22.9
4.762
7.236
6.894
10.52
7.732
7.226
9.603
5.41
12.73
5.015
12.35
5.785
7.168
5.444

Musical Instrument Redux

In this part of the lab we combine the flex sensor and piezo sensor to create a musical instrument. The first order of business is to wire the components together. We wire the piezo sensor as we did in the Piezo Sensor section. Except for hooking up the positive end of the parallel configuration to the OpenSchope, we hook it up to the AD A2 pin to use the data in our instrument. The flex sensor and audio components are connected the same way they were in the 1.3 Circuit Diagram. We can see the new configuration of our circuit in the Lab1 Part 3 Circuit Diagram below.



In this part we will be using the flex sensor to set the frequency of our tone, while the piezo sensor will be used to trigger the tone itself. We will reuse the code from Part 1.3 previously mentioned in the Flex Sensor section. Some modifications must be made. We include the same libraries as previously mentioned in the Part 1.3 implementation. We add pin AD A2 to our program to read from the piezo sensor, using AD_AddPins(). We add a new variable to store the piezo sensor reading into, p. We add a variable that defines the duration of our tone, td. All the other variables are repeated from the previous implementation. At the top of our infinite while loop, we clear our OLED, turn our tone off, and store our time in milliseconds into variable t. We then check if new AD data is ready with an if statement and AD.h function AD_IsNewDataReady(). We store the reading from the AD A1 pin into the flex sensor variable, f, and store the reading from the AD A2 pin into the piezo sensor variable, p. We then convert the flex sensor reading to degrees, and then from degrees to Hertz. We then implement the software filtering, identical to how we did previously for our Hertz converted from flex sensor readings.

After the software filtering we get into new coding for the piezo implementation. We check for the piezo sensor trigger. If our piezo sensor variable, p, is reading values higher than 100, we are certain the piezo has been triggered. We find this out through printing the value of the piezo sensor as it is at rest without being tapped. We find that at rest the piezo sensor reading sits at about 20. When it is tapped it shoots up to the 300-500 range and higher, depending on how hard you tap it. It will settle back to the 20-30 range within a second of being tapped. Knowing this we set the threshold of 100 as a trigger for turning tone generation on. If

the reading from the piezo is larger than 100 we set the frequency to the average Hertz of the current while loop iteration. We then turn the tone generation on. We start a while loop to wait for a second to pass (while(t%td != 0)). The tone duration triggered by the piezo sensor chosen in this lab report is one second. Within this while loop we need to check if the piezo sensor has been tapped again. If it has, we restart the tone duration. This is what the tone duration variable, td, is used for. It is set to 1000 by default, since 1000 milliseconds is one second. But if the piezo sensor is triggered again before this duration is over, we must redefine td so that the tone is prolonged by one second the moment it is triggered. Therefore when the piezo is triggered within this time duration (while(t%td != 0)) we store the current time in td with 1000 added to it. Therefore our while loop will exit when the current time plus 1 second (1000 milliseconds) has passed. We store the current time back into our time variable at the bottom of our while loop. Right outside the while loop we redefine the time duration, td, as 1000 milliseconds, so that the next tim the piezo is triggered, the tone only plays for 1 second. This concludes the Musical Instrument Redux portion.

Pseudocode:

```
int main(void){
   //initialize BOARD, AD ,TIMERS, ToneGen, & OLED
   INIT_ALL();

   //Add AD_A1 pin to read from & turn tone off
   AD_AddPins(AD_A1);

   //assign tone duration
   td = 1_second;

   //infinite loop
   while(1){
      ToneOFF(); //turn tone off
      t = TIME; //store time in t

      //check flex & piezo sensor reading
      if(AD_NewDataReady()){
         f = AD_A1(); //read from AD_A1 pin
         p = AD_A2(); //read from AD_A2 pin
      }

      //convert flex sensor to degrees & to Hz
      d = (-0.434*f) + 437;
      h = ((d*999)/180) + 1;

      //software filtering
```

```
      h_buff[h_i%100] = h; //store current hz
      h_i++;
      for(i = 0; i < 100; i++){ //take the average
         avg = avg + h_buff[i];
      }
      avg = avg/100;

      //check for Hz ceiling and floor
      if(avg > 1000){ avg = 1000; }
      if(avg < 1){ avg = 1; }

      //more software filtering
      if(abs(avg - p_avg) < 5){
         avg = p_avg;
      }
      p_avg = avg;

      //check for piezo sensor trigger
      if(p > threshold){
         ToneGeneration_Frequency(avg);
         ToneON();
         while(t%td != 0){ //keep tone on for td
            if(){
               p = AD_ReadADPin(AD_A2);
               if(p > threshold){
                  td = TIME + 1_second;
               }
            }
            t = TIME;
         }
         td = 1_second;
      }


   }
}
```

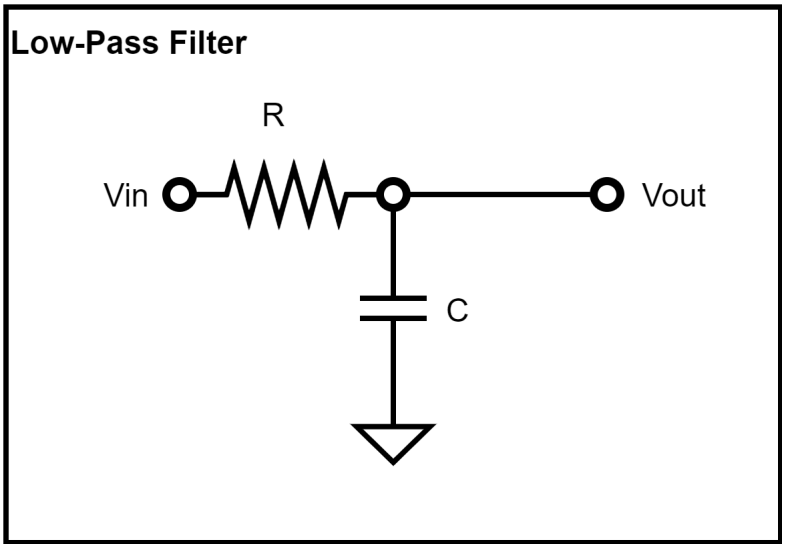
Simple Analog Filtering Analysis

In this part of the lab we will analyze different kinds of filters by deriving their transfer functions and plotting their magnitude vs. frequency graphs. We will then analyze the filters with the OpenScope. In this lab we are using resistors of 47 kOhms and capacitors of 1 nFarad. This means our corner frequency is about 3,388Hz.

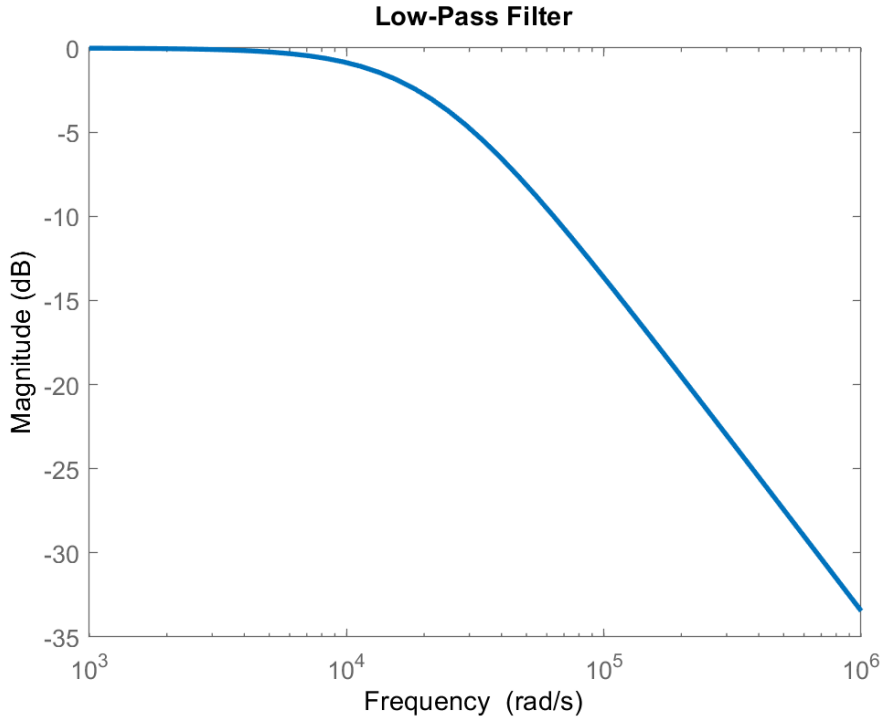
The first filter we are going to analyze is the RC low-pass filter. To find the transfer function we use KCL and solve for Vout/Vin. The result of this derivation is the following:

Vout/Vin = 1/(1 + sRC)

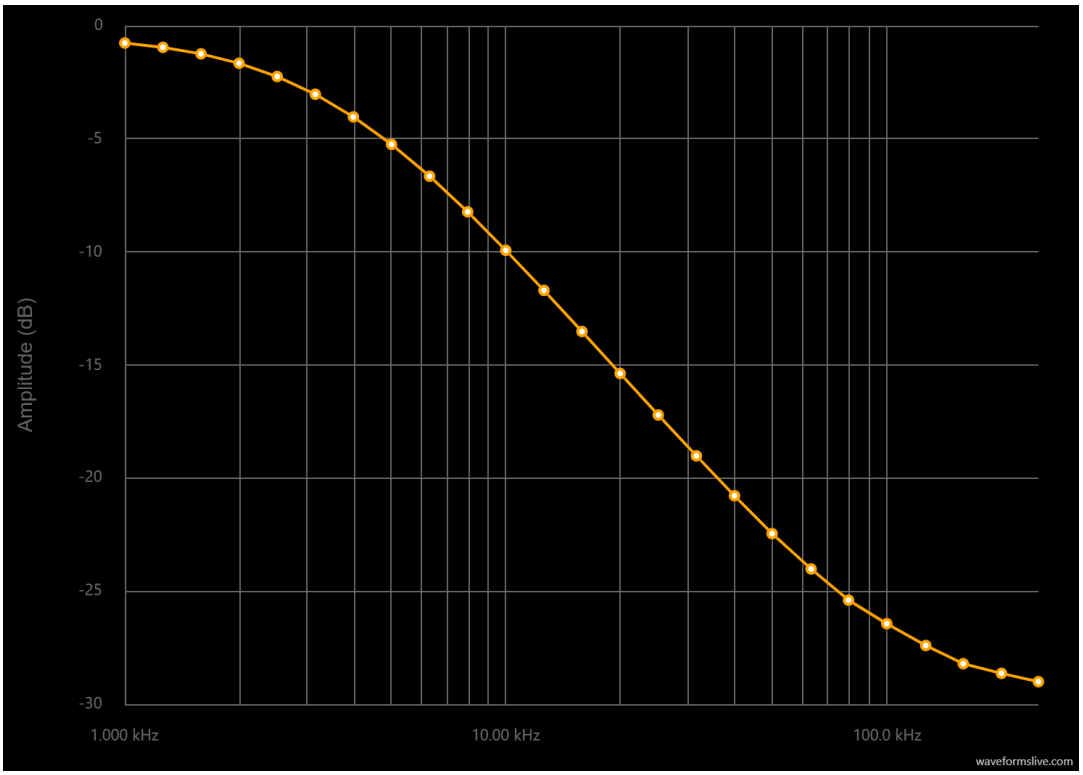
The circuit diagram for this filter is as follows:



We confirm the validity of our transfer function by plotting the magnitude vs. frequency in MATLAB, and comparing the plot to the analog reading from an oscilloscope. The plot from MATLAB can be seen here:



The plot from the OpenScope can be seen here:

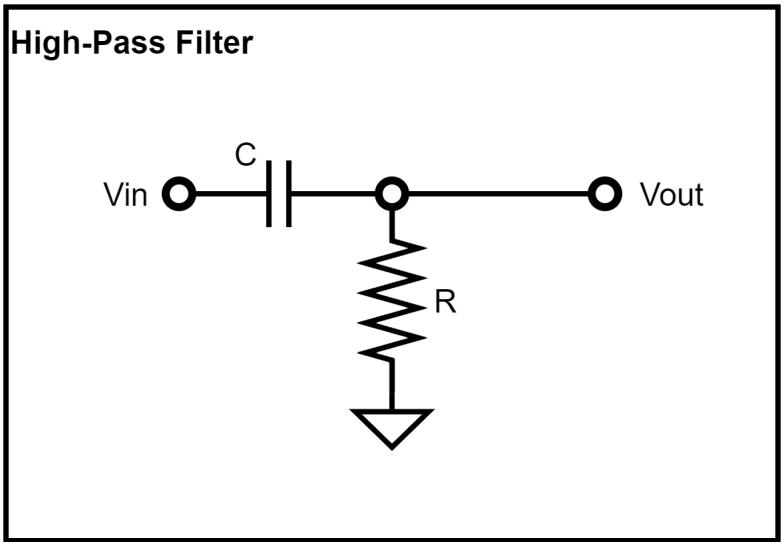


The theoretical magnitude vs. frequency plot look similar to the analog measurement, but they are not 1-to-1 replicas. We can see here that they look like they do begin to decay around the corner frequency of 3,388Hz.

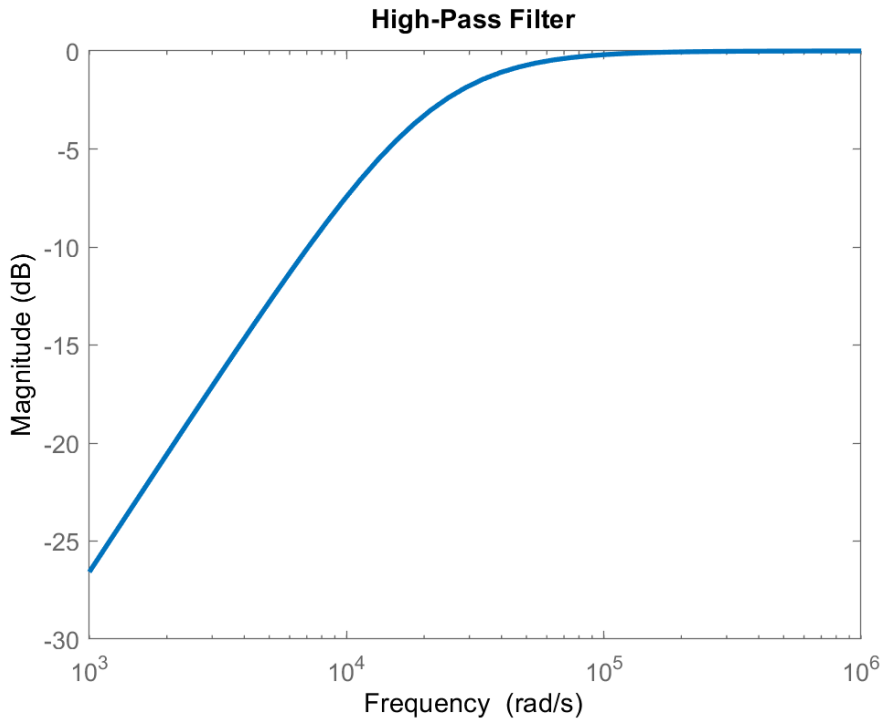
The second filter we analyze is the RC high-pass filter. To find the transfer function we use KCL and solve for Vout/Vin as we did for the low-pass filter. The result of this derivation is the following:

Vout/Vin = sRC/(1 + sRC)

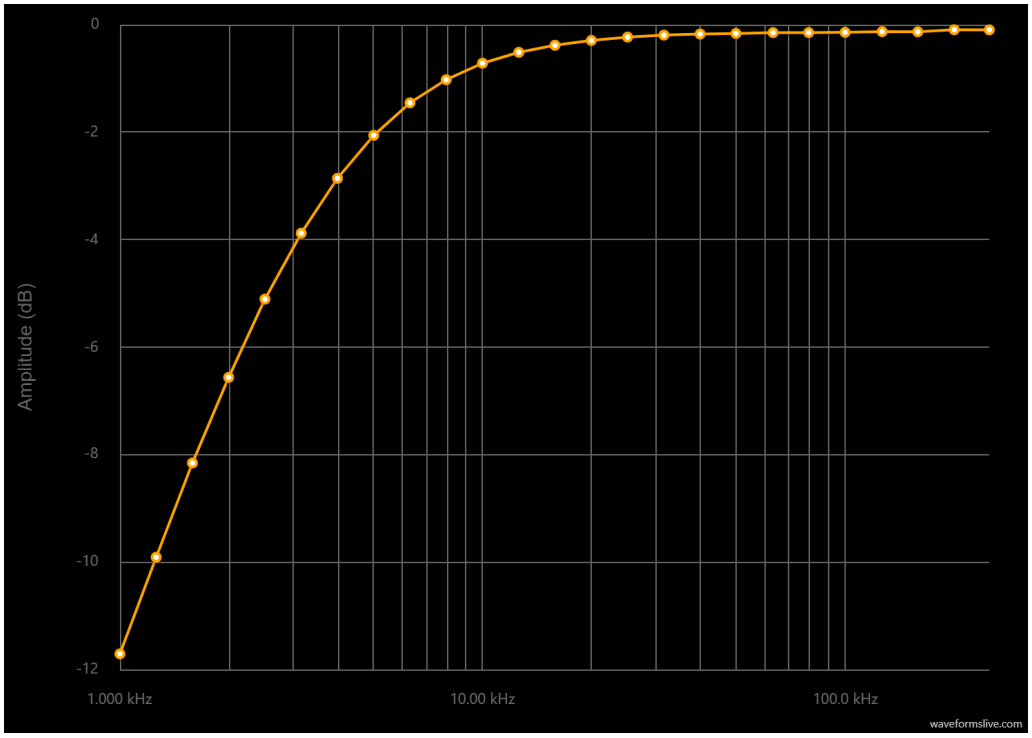
The circuit diagram for this filter is as follows:



We confirm the validity of our transfer function by plotting the magnitude vs. frequency plot of this filter both in MATLAB and with analog measurements using the OpenScope. The MATLAB plot can be seen here:



The OpenScope plot can be seen here:

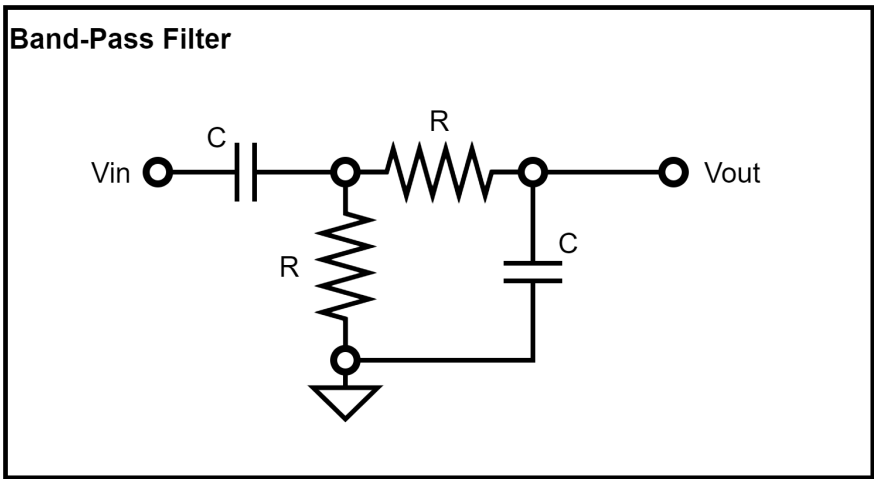


From this we can see that the plots are similar, but they are not exactly the same. Again we can see a ramp up close to the corner frequency of 3,388Hz.

The third filter we analyze is the band-pass filter, which is the high-pass filter cascaded with the low-pass filter. The transfer function obtained from convoluting the two previous filters is as follows:

Vout/Vin = (sRC) / (1 + 2sRC + (sRC)^2)

The circuit diagram of the band-pass filter is as follows:



If we use KCL on the circuit to find the transfer function we get a different answer, assuming resistors and capacitors of differing values:

Vout/Vin = (sR1C1) / (1 + s(R1C1 + R1C2 + R2C2) + s^2(R1R2C1C2))

If we assume the resistors and capacitors are equal we get:

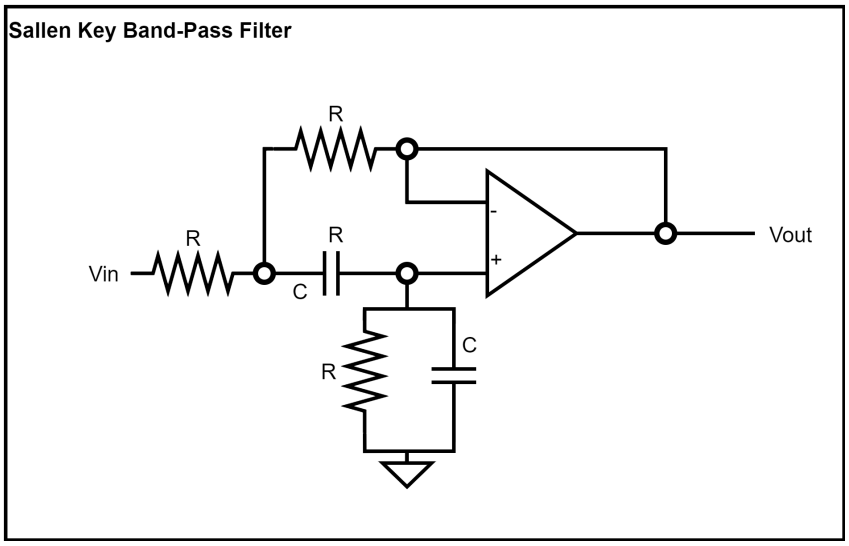
Vout/Vin = (sRC) / (1 + 3sRC + (sRC)^2)

This differs from the convolution we derived previously by multiplying the transfer functions.

The fourth and final filter we analyze is the sallen key band-pass filter. This filter uses an Op-Amp to eliminate downstream loading. The transfer function found from KCL on this circuit is as follows:

Vout/Vin = (sRC) / (2 + 4sRC + (sRC)^2)

The sallen key band-pass filter looks as follows:



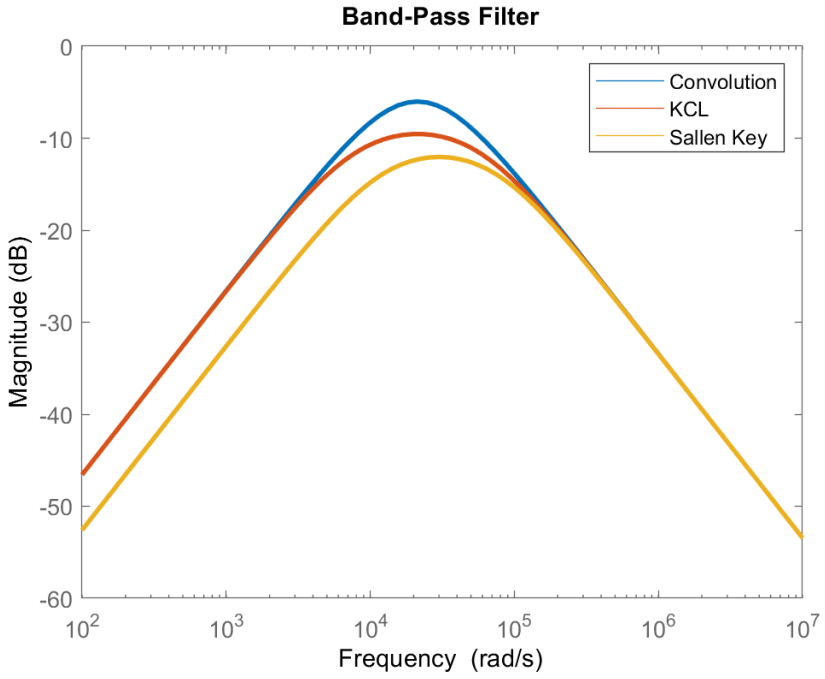
The transfer function of the sallen key band-pass filter if the resistors and capacitors are all different is as follows:

Vout/Vin = (sR2R3C1) / (R1 + R2 + s(R1R3C2 + R1R2C1 + R2R3C2 + R2R3C1) + s^2(R1R2R3C1C2))

If the resistors are equal, and the capacitors are equal, the transfer function is as follows:

Vout/Vin = (sRC) / (1 + s(4RC) + s^2(R^2C^2))

A comparison of the three transfer functions (Convolution, KCL, Sallen Key) found from the two band-pass filters can be seen plotted via MATLAB here:



Here we can see a difference between the three transfer functions we derived with the band-pass and sallen key band-pass filters.

Conclusion

In this lab we learned to implement the flex sensor and piezoelectric sensor in circuit designs. We started by linearizing the flex sensor and reading the values as smoothly as possible with software filtering. We learned to output a tone dependent on the flex sensor in Part 1.

We then learned how to use the piezoelectric sensor to trigger an event. This sensor can be dangerous to components of our circuit if we do not add a snubbing circuit. We learned this by examining the voltage peaks of the piezoelectric sensor in Part 2 of the lab.

Finally we put both sensors together to implement a musical instrument whose frequency is dependent on the bend of the flex sensor and tone activation is dependent on the piezoelectric sensor.

After finishing up the programming we moved on to analyze filters by deriving their transfer functions. We plotted the magnitude vs. frequency graphs in MATLAB and confirmed the validity of them via analog readings on an oscilloscope. Building the filters and seeing the magnitude vs. frequency confirmed on our oscilloscope was satisfying after suffering through the math to find the transfer equations.

A bump in the road during Part 1 I would like to acknowledge is the difficulty of assembling the flex sensor with the clinch without pliers. It was impossible for me to try and close the clinch with the flex sensor in place. I thought I was messing something up, but the reality is that the clinch needs significant force to snap into place. I recommend anyone who does this lab in the future use pliers.

Another bump in the road during Part 2 was my misinterpretation of linearization of the flex sensor. I thought we were supposed to read the resistance from the voltage divider circuit, which seemed impossible to me. After rereading the manual a couple of times it finally dawned on me that the resistance was not of note to use. We just wanted to observe the flex sensor values output from the voltage divider dependent on the flex of the sensor. Make sure to read the lab manual carefully.

Finally, I think this was a pretty fun lab where we get to work with two different but interesting sensors. The musical instrument redux was my favorite part of the lab. The analog filter analysis was probably the roughest for me because it had been so long since I had solved for transfer functions from circuits.

All in all this was a satisfying lab. I enjoyed learning about the sensors and filters examined in this lab. The incremental structure of the parts made things easily digestible.