Lab 3: Spectrum Analyzer

# Introduction

The second lab builds off the hardware from the first lab, but demonstrates a different kind of processing, spectral analysis. Spectral analysis in audio is the analysis of the frequency content of sound. Many stereos have a spectrum analyzer built in, we recognize it as the colored lights that dance to the beat of the music. I used this opportunity to build a spectrum analyzer of my own.

# Sampling

There are three main methods to the spectrum algorithm: sampleInput, populateLedMatrix, and drawLed. The sample input method is exactly how it sounds. First, I clear the fht\_input array, which is an array that stores all the samples to run the Fast Hartley transform upon. I can do this quickly without looping using the memset function, applying zeros to all indexes. Next, I take 256 samples as defined by the FHT\_N variable. The two choices here are 128 or 256, which determine how many output bins are will be created. I have an 8x8 LED matrix so chose 256 as it creates 8 bins [1]. I used a for loop to collect 256 samples from the analog input zero pin in which I connected my audio input. I had two choices here; I could use the interrupt routine to collect the samples or a simple analog read. The analog read isn’t as optimized as the interrupt routine so it takes a bit longer. Usually, this would be a disadvantage, but I found when using the interrupt routine the LED output wasn’t as effective. Using interrupts means the code constantly stores context and executes the interrupt routine. The algorithm I created samples first, then processes those samples so there is no reason to continue executing the interrupt the routine in the middle of the populateLEDMatrix or drawLED methods. It actually ends up slowing down the loop and the LED output lags. For this reason I chose to just sample using the analogRead method.

After the sampling is complete there are a series of FHT functions to prepare the data and then process the data. The final function fht\_mag\_octave populates the fht\_oct\_out array with the processed data in bins. There are other types of FHT magnitude operations, but I chose octave because it corresponds with my goal of representing audio visually. The octave function places the frequency data into bins that are reflective of how humans perceive sound [1]. This is why many spectrum analyzers use a logarithmic scale when representing frequencies. Powerful low frequencies are put into their own bin for the most part, while weaker high frequency content is grouped together. Using FHT\_N of 128 the bins are:

[0, 1, 2:4, 5:8, 9:16, 17:32, 33:64, 65:128]

**Populate LED Matrix**

After sorting the frequencies into bins I could easily create a spectrum analyzer with a few operations and the set\_column method provided by the LED Control library. In fact I did that initially and have a video attached with what that looks like. The issue was using the set column function and writing frequencies from low to high in a loop creates a distracting left to right animation. It’s very easy to see each column written one by one and there is a noticeable delay in the always-changing high frequency content. Therefore I created some operations to convert the column data into row data and then write the rows from the bottom up. This gives a much better visual representation of the sound without a noticeable delay.

The LED control has a few nice functions to write rows and columns. The function takes three parameters: the address of the LED controller, the row/column index, and an integer value [3]. The value is then represented as a binary value with the LEDs. For example, 0 would have no LEDs illuminated, 3 would have the first two, 4 would have only the 3rd LED illuminated, etc. Knowing this, I want to convert the values I receive from the fht\_mag\_output array into the closest binary value: 0, 1, 3, 7, 15, 31, 63, or 127. I choose these because they mean all LEDs are illuminated beneath.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 127 | 63 | 31 | 15 | 7 | 3 | 1 | 0 |
| 127 | 127 | 127 | 127 | 127 | 127 | 127 | 127 |
| 63 | 63 | 63 | 63 | 63 | 63 | 63 | 63 |
| 31 | 31 | 31 | 31 | 31 | 31 | 31 | 31 |
| 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

The calculate column value to get this binary representations takes the octave magnitude value, multiplies it by the number of rows, and divides by a maximum magnitude value. The maximum magnitude is assigned as the largest value given by the FHT octave out. I used this so that the spectrum can scale to any size matrix and have maximum resolution. All columns are scaled as a percentage of the maximum sized column. To prevent outliers and problems caused by a change of input signal amplitude I consistently decrement this maximum magnitude value in the populate LED matrix method. This way the max magnitude will consistently be updated. Then I take two and raise it to the power of the value, which I expect to be between 0 and the number of rows. This gives a value such as 0, 2, 4, 8, 16, 32, 64, or 128, which would all only result in one LED illuminated. I then subtract 1 to get to my goal values and illuminate all LEDs.

The next step is to translate these column values into row values so I can use the setRow command. The way I do this is one by one through the column array and check if the LED should be lit or not and assemble a row value that represents the columns. I do this with the bitRead method, which takes an integer and a bit position and returns a 1 or 0. For example given the integer 4 and bit position 3 the result would be one. Therefore, the loop goes through reading each bit, multiplying the value by two, raising it to the power of the bit position, and adding it to the row integer value being assembled. The only exception to this is on bit position zero, because raising anything to the power of zero is 1, but we must represent both zero and 1 in this position. This is essentially just creating an integer value by doing a binary to decimal conversion, going across the columns.

After the row array is assembled, I simply loop through the array and call the set row method with the integer value. I have a few counters: rowIndex and colIndex in these two methods to control the order the LEDs are written in, giving the matrix the proper orientation.

The following is a shortened version of the column to row conversion:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Column 0 = 3 | Column 1 = 0 | Column 2 = 3 |
| Row 2 =1 |  |  |  |
| Row 1 = 5 |  |  |  |
| Row 0 = 5 |  |  |  |

For example, calculating row 1:

1. The bitRead of column 0 for row 1 would return 1. Row 1 = 1
2. The bitRead of column 1 for row 1 would return 0. Row 1 = 1 + (0\*2)^1 = 1
3. The bitRead of column 2 for row 1 would return 1. Row 1 = 1 + (1\*2)^2 = 5
4. Call setRow(0, 1, 5)

**Results**

I’ve attached a video explaining how the spectrum works and demonstrating it. The end result works quite well with a few improvement opportunities; the first being the quality of the input circuit. The spectrum seems to track noise in the input circuitry so that LEDs are displayed even without playing audio. I might be able to fix this using an ADC offset taken at the beginning and removed from future samples. Another objective I would like to try is scaling the solution for larger matrixes. I used define statements to store the row and column numbers in hopes that I could eventually replace my small LED matrix with a larger one, but I haven’t been able to test this.

One interesting result I found in this lab is that the FHT operation shows low frequencies are stronger than high frequencies. As I said, I used the octave magnitude function to sort the frequencies in bins. The lowest frequencies, the first two bins, always have the highest magnitude. After doing some research I found this is known as the pink spectrum, or pink noise. “The frequency spectrum of [pink noise](https://en.wikipedia.org/wiki/Pink_noise) is linear in [logarithmic space](https://en.wikipedia.org/wiki/Logarithmic_scale)” [2]. So this means the energy between 100 and 200 Hz is equivalent to the energy between 1000 and 2000Hz, or 10kHz and 20kHz. The octave magnitude ordering creates this linear relationship with logarithmic scaling.

Finally, I must point out this was far from my original intention with this lab. Before working with the Arduino I had very high hopes and ambitious objectives. Those were crushed after I struggled mightily with the first filtering experiment. I ended up learning more about the Arduino and using more advanced functions than I originally planned. I learned early that the built in Arduino libraries, while very intuitive, don’t offer the performance required for some tasks. I was able to experiment with interrupts and bit manipulation on the Atmel core itself to work around some issues, but it wasn’t quite enough to create my full spectrum with filtering. I feel I accomplished quite a bit in these three labs; however, much of the knowledge was gained before I started the first lab. Getting into the Arduino is as easy as opening up a project book and imitating the projects. I spent a few weeks doing this before I felt comfortable moving on to something larger like these labs. Attempting real time operations can be done with some tweaks; however, that’s not a strong point for the Arduino. All in all the Arduino is a great prototyping tool for something with fairly limited scope. The libraries are well documented and the Atmel manual is easy to pick up and read.

**Works Cited**

1. Arduino FHT Library. (2013, March 13). Retrieved December 8, 2015, from <http://wiki.openmusiclabs.com/>

2. Colors of noise. (2015, November 4). In Wikipedia, The Free Encyclopedia. Retrieved 04:07, December 9, 2015, from <https://en.wikipedia.org/w/index.php?title=Colors_of_noise&oldid=688971460>

3. LedControl. (2007, June 23). Retrieved December 15, 2015, from http://playground.arduino.cc/Main/LedControl