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.

# Spectrum Analyzer Algorithm

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 required. I have an 8x8 LED matrix so chose 256 as it creates 8 bins. I used a for loop to collect 128 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 the analog read introduces an almost necessary delay for the output to be visually pleasing. After sampling I do a few operations before sampling again and redrawing to the LED. This allows the previous drawing to remain displayed a brief period longer when sampling for the next display. When using the interrupt routine the sample loop was executing so quickly that the LEDs would flicker rather than remain solid long enough to notice. It seems to be equally accurate either way, but for my purposes it just looks better without the interrupt routine. 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 wave 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. 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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 value and subtracts the minimum decibel value. The minimum decibel value is just a value I created so the bottom rows of lights are illuminated. Essentially this gives the full magnitude of the frequency bin value. Then I divide the result by the step size. The step size is a scalar for the size of the LED matrix calculated by subtracted in the minimum dB from the maximum and dividing by the number of rows. I then take the ceiling of the result. Then I take the value and raise it to the power of two, 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 only way I could think to do this was to go one by one through the column array and check if the LED should be lit or not. 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 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

**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 thing I would like to try is scaling the solution for larger matrixes. I used define statements to store the row and column values in hopes that I could eventually replace my small LED matrix with a larger one, but I haven’t been able to test this.

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

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