Performing the Spectrogram on the DSP Shield

EE264 Digital Signal Processing Final Report

Christopher Ling
Department of Electrical Engineering
Stanford University
Stanford, CA, US
x24ling@stanford.edu

***Abstract*—This report briefly describes the theory and implementation of a spectrogram of an audio file performed by a DSP shield that sends the spectrogram data over a serial link to MATLAB. A brief analysis of the different spectrogram data will also be discussed.**

I. INTRODUCTION

The Discrete Fourier Transform (DFT) is a great way for a microprocessor to determine the frequency content of a signal, but like most interesting signals that engineers would like to analyze vary over time. This means that the frequency content of a signal will change over time. Furthermore, taking the DFT of a very long signal with a lot of samples can be computationally expensive but not very information because you do not know when certain bands of frequencies are present at a certain point of time. In previous labs, the DSP shield has been used for spectrum analysis for an audio signal that is 512 samples long at a sample rate of 48kHz, meaning the sample is 10ms.

To capture the frequency content of a signal as a function of time as well, I programmed the DSP to perform DFT on a specific time window of the signal, move the window, and perform the DFT again. Then I plot how the frequencies change over how far the window is shifted in time. This process is known as the spectrogram.

The goal of this project is to use the DSP shield record and perform a spectrogram on a 2 seconds long audio file with a programmable window length, window type and block skipping length. This spectrogram will then be sent over the serial port to a PC computer via MATLAB for visual analysis.

II. TIMELINE

Before following through with the project, the timeline of the project was established to get a good sense of how much time I will need to take for the project. The biggest change I had to make was that I was unable to get the real-time spectrogram to work because the serial port was too slow. I also had to take some time to develop code that writes and reads audio files to the SD card.

A. Anticipated Timeline

Week 1: Get input audio and perform one window of the spectrogram to be sent over MATLAB
Week 2: Perform full spectrogram with all slices together.
Week 3: Vary parameters like window length/type and sample rate for different spectrograms.
Week 4: Play audio while spectrogram plays, maybe to some processing like speech or note recognition.

B. Actual Timeline

Week 1: Get input audio and perform one window of the spectrogram to be sent over MATLAB
Week 2: Get full Input Audio by storing the data onto an SD card. Read the data from the SD card.
Week 3: Perform full spectrogram and send spectrogram and audio to MATLAB
Week 4: Vary parameters like window length/type and sample rate for different spectrograms.

III. CONCEPTS USED

To perform the spectrogram, we need to first understand the DFT, Windowing, and the Time-Dependent Fourier Transform.

A. Discrete Fourier Transform

To get the Fourier transform of a discrete signal, which is the only possible choice of getting a signal as opposed to a digital signal, we can only perform the Discrete Time Fourier Transform, which is defined as:

$$X(e^{jw}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \qquad (1)$$

But because we are sampling our signal, the DFTF will be a periodic signal that is a function of the sampling frequency. To ensure that there is no aliasing of the signal, we assume that the sampling frequency is twice the maximum frequency of our sampled signal.

However, the DSP cannot directly store the Fourier

Transform of a signal in the DSP because it is a continuous signal; therefore, the DFTF needs to be discretized to N-points, meaning, we need to store the DFT, which is defined as:

$$X[k] = X\left(e^{\frac{2\pi k}{N}}\right) \qquad (2)$$

The DSP shield has a built-in Fast Fourier Transform (FFT) function that I will use to determine the DFT, and it's the most efficient when the number of points used is a power of 2.

For an N-point DFT of a sampling frequency of f_s, the band of frequencies we get range from 0 to $f_s/2$ with N/2 frequencies available to use for viewing. This is because the frequencies from $f_s/2$ to f_s are flipped duplicates of the frequencies from 0 to $f_s/2$ because of the periodic nature of the DFT. This means that a longer DFT will give us more frequency resolution of our signal.

B. Windowing

The idea of the spectrogram is to perform the DFT on a specific section of the audio sample, and to do this we need to get a window of the sample.

If we were to just simply copy a segment of the signal to be processed by the DFT, we are essentially multiplying the signal in the time domain by a rectangular function, which means we are convoluting our signal in the frequency domain by a sinc function. This greatly distorted our frequency response because the sinc function has lower magnitudes in the high frequencies, meaning there will be distortion in the signal.

Ideally, we want to get a rectangular window of the signal in the frequency domain, so the best way to accomplish this is to multiply the time-signal by a window function who's frequency response looks more like a rectangle. There are several ways to do this, but the most common way is to use the Hann, Hamming, or Bartlett window. The Bartlett window is easiest to compute because it's simply a linear function. The sinc function could work, but it's computationally complicated, so the Hann and Hamming windows are better alternatives.

As seen in Figure 2, the frequency response of the Hann and Hamming windows are very close to rectangular, meaning the frequency response will be less distorted than convoluting with the rectangular window (which isn't rectangular in frequency response) and the Bartlett Window (which has some odd behavior approaching the edges of the windows).

Fig. 1. Different Window in Time-Domain

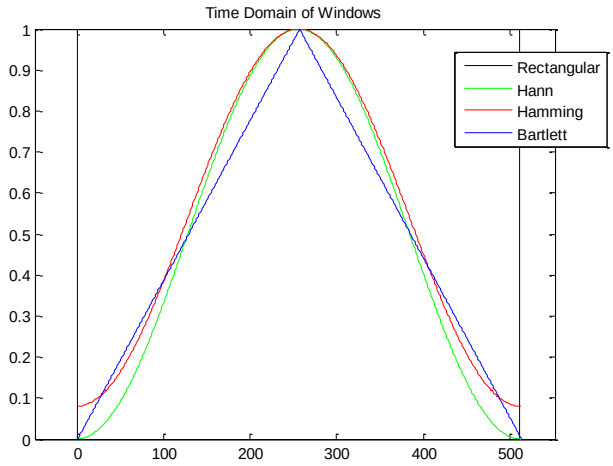
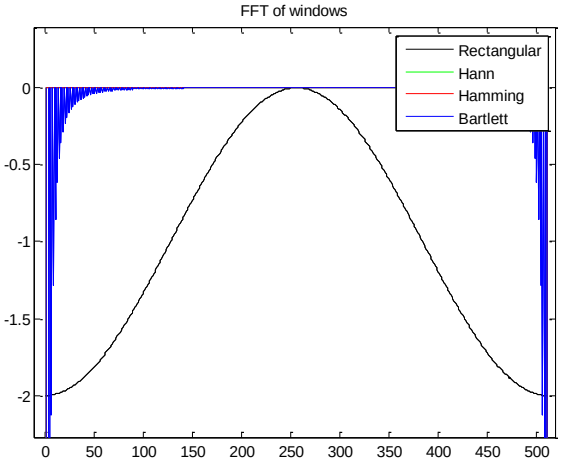


Fig. 2. Different Windows in Frequency-Domain (Hann and Hamming are basically rectangular, it's hard to determine from the graph)



C. Time-Dependent Fourier Tranform

Finally to put everything together, we will now discuss how to perform the Time-Dependent Fourier Transform. An overall description is that we simply need to multiply our signal by the window that is time-shifted to the portion of the signal we wish to analyze and then perform the DFT on that section. The overall equation is:

$$X[n,k] = \sum_{m=0}^{L-1} x[nR+m]w[m]e^{-j\left(\frac{2\pi k}{N}\right)m} \qquad (3)$$

The Window w[m] has a length L, n is which block of samples to get, k is the frequency in the DFT, and R is how many samples to skip to get the next window of samples.

IV. Implementation

The three main functionalities implement were Audio sample capture, spectrogram computation, and MATLAB control of DSP.

A. Acquired Data

In the beginning, my first attempt to acquire the audio data was to copy data from the Audio buffers every single a new buffer is filled, and keep filling them until the required window length is acquired. This was really difficult to implement because depending on how many samples I skip per block and the length of the window, the number of samples to copy from the audio buffer complicated and there needed to be a place to store the other samples.

Afterwards, I decided that the simplest way was to store the entire audio file onto an SD card, very much like the recorder lab that was done in the beginning of the quarter. This way, the audio data will always be available and I only need to parse through the .wav file stored in the SD card. I implemented a recording function that get the audio from the DSP input for a specified amount of time, and then it store the audio file into the SD card. Furthermore, I also implemented a function that reads the audio file from the SD card and sends it over to MATLAB for viewing. Because the serial channel is limited by the number of bytes it can send, I split the audio file into multiple chunks and concatenate the data together.

B. Spectrogram Computation

The spectrogram was computed from equation (3) using the built-in FFT library commands. The two options I had in storing the spectrogram data was either in the SD as a text file or to send it over back through MATLAB. I ended up choosing the latter because I was hesitant on requiring that the user would take out the SD of the DSP shield to analyze the spectrogram data, so I had the data be sent over MATLAB. Once again, because the maximum serial data length was 1024, I simply made the maximum length of the window to be 512 to take account of complex values. That way, I send one spectrogram window in one serial send and I keep sending the spectrogram windows until the entire spectrogram has been processed.

If I wanted to, I could've also split the spectrogram data into multiple serial sends, but this can be done in future iterations.

C. MATLAB Control

The DSP shield depends on MATLAB to determine what to do. Drawing inspiration from previous labs, I implemented a command system that MATLAB can send to the DSP shield.

TABLE I. MATLAB Commands

Number	Command
10	Record DSP Input
11	Send Audio File
20	Compute and Send Spectrogram
30	Send Window
31	Block Skipping Size
32	Set Record Time

The test MATLAB script I wrote was to get record 2 seconds of audio, send a window that I design before-hand, get the audio file, and then get the spectrogram data. As mentioned before, all of the data has to be sent in bursts. As a result, the MATLAB script has to take into account of how many bursts of data the DSP needs to send and how many get.

D. Other Comments

As mentioned earlier, I had attempted to perform a real-time spectrogram of the input audio to the DSP, but the biggest bottleneck I faced was the speed of which the data is sent over to MATLAB that I ended up not implementing this part into the DSP shield. One thing to try in the future is to reduce the amount of data that is sent over through serial.

V. Results

I compare the results of my spectrogram between the spectrogram developed by MATLAB. I will be using the spectrogram I develop to analyze the audio signals I have while varying parameters of the spectrogram.

The audio input signal I processed was me saying "sha" repeatedly. The beginning "sh" has a lot of high-frequency content in the audio while the "a" has fuller low-frequency content. The spectrogram output I anticipate is one that has high frequency content in the beginning then low frequency content towards the end.

All frequency values in the y-axis are all normalized frequencies, where I divide the actual frequencies by the sampling rate and multiply them by 2π, meaning the maximum normalized frequency seen in the spectrogram plots should be π.

A. Comparison to MATLAB

Figure 3 shows the comparison between the spectrogram computed by the DSP shield vs the spectrogram computed by MATLAB. I use a Hann window of length 512 samples, I have a block skipping rate of 256 samples, a record length of 2 seconds, and a sampling rate of 24kHz. The two spectrogram are very similar to each other when the same window was used.

The time-domain audio signal lines up quite well with the spectrogram output because it portion of the audio where the frequency is higher, the spectrogram will show a higher value in higher frequencies.

B. Different Window Types

Figure 4 shows the comparison between different windows used to multiply the selected portions of the audio signal. I used a Hann window for the first section and a Rectangular window for the second part.

Because the Rectangular window does not have a very clear high-frequency cut-off behavior, there is some strange transitions spikes that are a result of using the rectangular window that are less prominent than the ones in the Hann window.

Fig. 3. Spectrogram Comparison between DSP and MATLAB

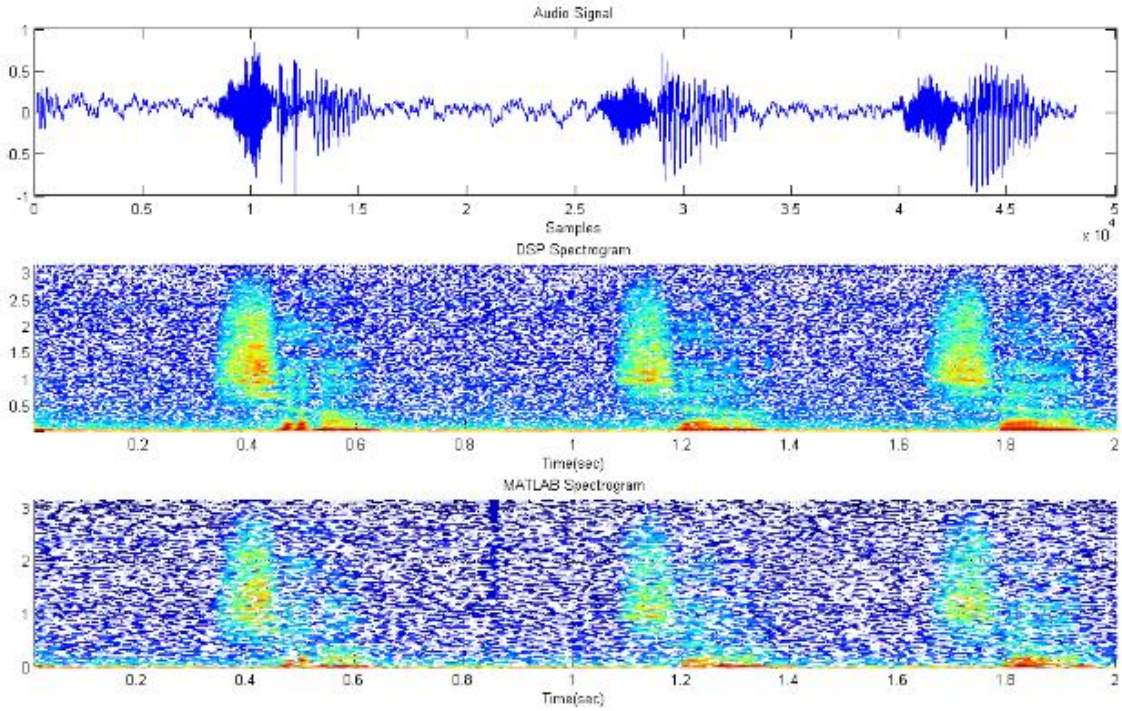


Fig. 4. Spectrogram Comparison of Window Types

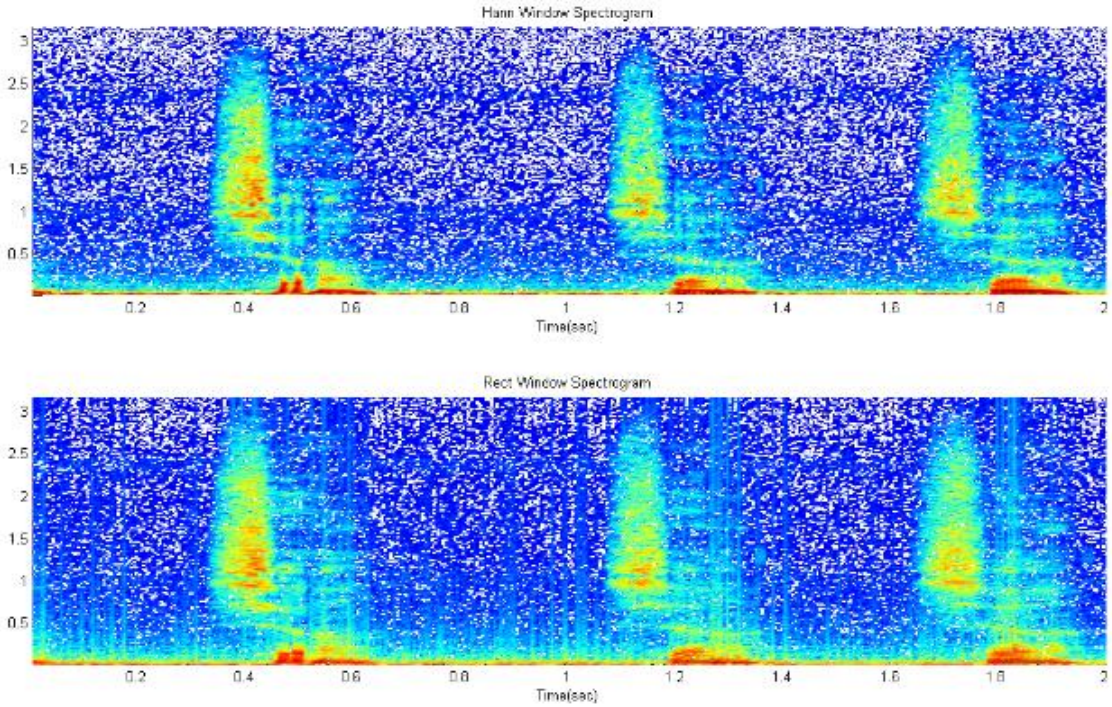


Fig. 5. Spectrogram Comparison of Window Lengths

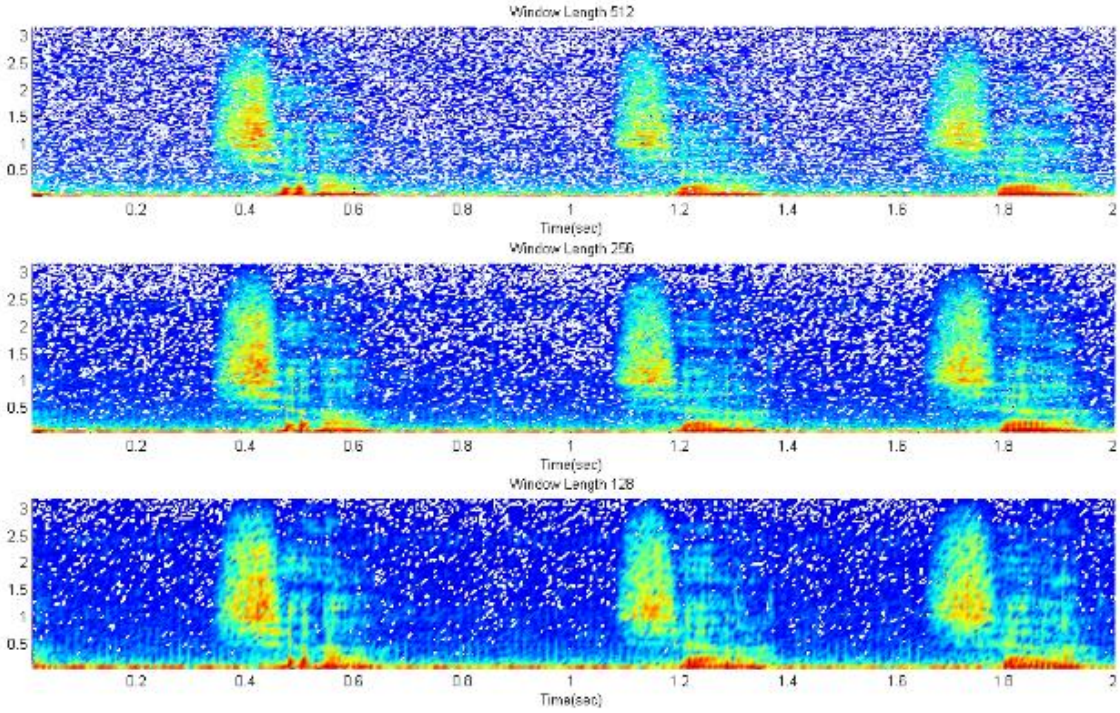
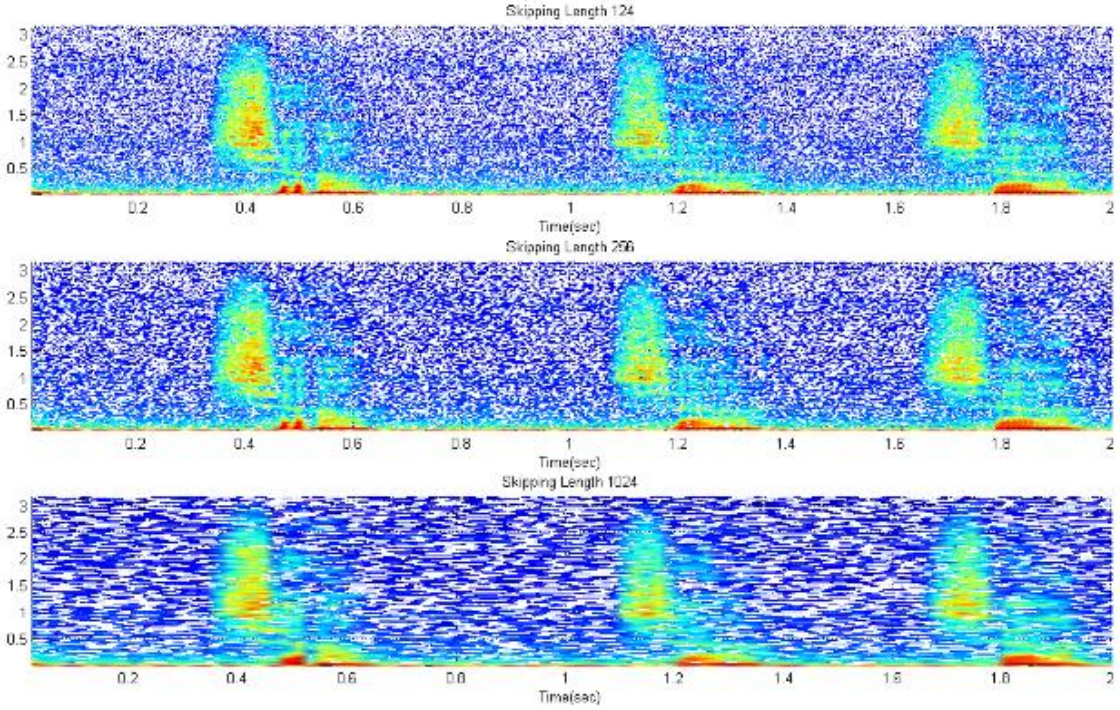


Fig. 6. Spectrogram Comparison of Block Skipping Amounts



C. *Different Window Lengths*

Figure 5 depicts using different window lengths for the Hann filter, and as expected, the frequency resolution is less for shorter windows. The lengths I use are 512, 256, and 128. This is because there are fewer frequencies that are shown in the spectrogram. The spectrogram bands for shorter windows are thicker than those of the spectrogram bands in longer windows.

D. *Different Skip Values*

Finally, Figure 6 compares the spectrograms of different block skipping values. With window lengths of 512, I use the values 128, 256, and 1024 (which has no overlap). The spectrogram with a larger block skipping amount shows that the frequency values are thicker over time which makes sense because we are sampling fewer windows over the same amount of time.

VI. Future Work

After developing a spectrogram, the next step would be using the data from the spectrogram to determine various qualities of the signal. An example of a potential application of the spectrogram is to do some audio and speech processing. We know that various speech patterns and music tones have different frequency contents in the sound, so by performing the spectrogram on the signal, one can predict what that sound was. A naïve speech or tonal recognition program can be developed from this spectrogram in the future.

VII. Appendix

List of critical files in compressed folder:

- Spectrogram.ino – Source file for DSP Shield
- Spectrogram_MATLAB_Comp. – compares the MATLAB and DSP shield spectrogram outputs
- Spectrogram_Skip_Comp.m – compares the spectrogram outputs at different block skipping values
- Spectrogram_Window_Length_Comp.m - compares the spectrogram outputs at different window lengths
- Spectrogram_Window_Type_Comp.m – compares the spectrogram outputs at different window types.