Hanchel Cheng
Sevy Harris

Tempo Estimation and Manipulation

I. Introduction

This project was inspired by the idea of a smart conducting baton which could change the sound of audio in real time using gestures, like a conductor does for a live orchestra. While there are numerous ways to shape music, this project focuses exclusively on the tempo, or speed at which a piece of music is played.

The project is divided into two parts: tempo estimation and tempo manipulation. The tempo estimation algorithm is based on the algorithm by Scheirer [1] which uses a series of first order comb filters to try to resonate the input signal at its tempo. The implementation was done in almost real-time, sacrificing accuracy for speed while compromising for lack of computational power and memory on the DSP Shield.

The objective of the tempo manipulation is to alter the speed of the song while keeping the same fundamental frequencies of the audio. This was achieved by spacing pitch peaks corresponding to the desired tempo. Pitch peak detection can be done through algorithms such as autocorrelation in the time-domain and the Cepstrum method in frequency-domain, but our implementation is based on the pitch synchronous overlap and add (PSOLA) method for speech manipulation. Implementing the algorithm in real-time was not an option given the limited memory on the DSP Shield.

II. Timeline

Our timeline of project milestones is detailed below (with the projected milestone dates in parentheses). While we managed to stay slightly ahead of our proposed schedule for the first half of the project, we did not accurately estimate the time it would take to debug the implementations on the DSP Shield. The final week of the project was much more rushed than we anticipated.

February 14: Finalized project proposal (week of 2/14/15)
February 23: Successfully implemented beat detection from [1] in Matlab (week of 2/23/15)
February 24: Midpoint Check demonstration of Matlab implementation successfully detecting click track beat (week of 3/2/15)
March 6: Successfully implemented beat change in Matlab while avoiding dramatic pitch changes (week of 3/9/15)
March 9: Finalized beat detection on DSP Shield (3/11/15)
March 11: Demonstration in-class showing real-time beat detection on the DSP Shield and the beat modification in Matlab (3/11/15)
March 12: Finalize beat change for both doubling and halving tempo on DSP Shield (3/11/15)
March 13: Final demonstration to verify total functionality (3/13/15)

III. Class Concepts

Downsampling

The audio input to the DSP shield (sampled at 48 KHz) was downsampled by a factor of 128 to allow us to analyze longer time signals without exceeding the Shield's limited storage capacity. When we downsampled the signal, we essentially discarded the frequencies above ~200 Hz (we did not include a low pass anti-aliasing filter due to Shield processing constraints and this certainly affected the accuracy). We were able to ignore these frequencies and still get results because the fastest tempos used in music (~4 Hz) are much lower than the frequencies being discarded.

Filtering

The tempo was estimated using IIR comb filters that feed the signal back at precisely the delay corresponding to one possible tempo. If the delay of the comb filter matched the tempo of the music, the output of the filter would have a much bigger response, as seen in the figure below.

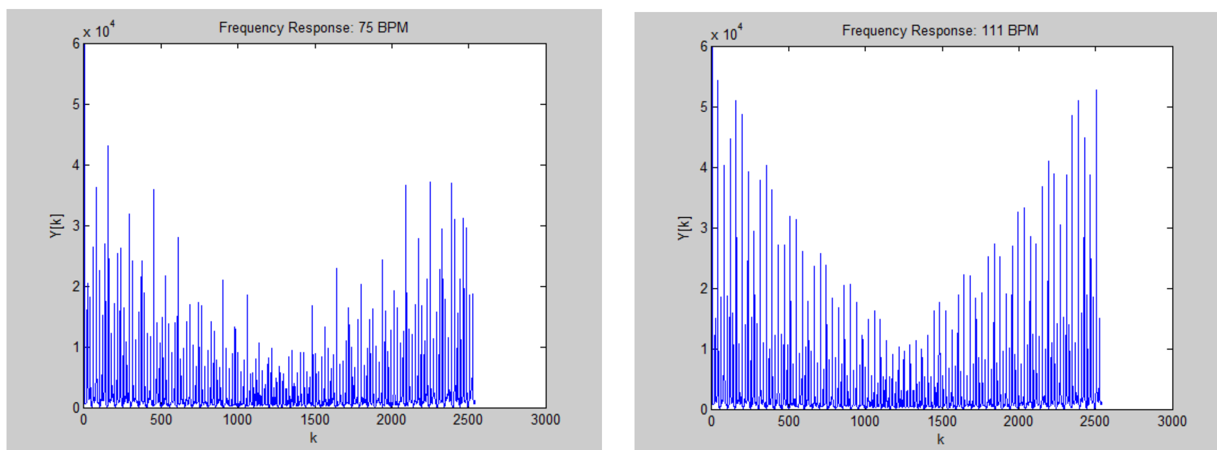

Figure 1: Shows the output of a mismatched comb filter (left) and the output of a resonating comb filter (right) where the filter delay matches the tempo of the audio input.

FFT

We used the FFT to analyze the signal in the discrete frequency domain. This was critical to identifying repetitions in the signal using fewer computations. We used the fact that (circular) convolution in the time domain is equivalent to multiplication in the (discrete) frequency domain to reduce the total number of computations. The comb filtering was all performed in the discrete frequency domain.

Windowing

Before being overlapped and added, the sampled audio segments were smoothed by a Hamming window so that resulting audio would not have discontinuities during playback.

Quantization

The beat modification processing implemented on the DSP Shield in Q15 was fed an input from Matlab that originally took values with magnitude greater than 1. Q15 can represent values no greater than 1 and less than -1. Therefore, all the input from Matlab was first normalized using the maximum value so that it could be represented in Q15. Both the window coefficients obtained from Matlab's Hamming function and the inputs were multiplied by 2^{15}

IV. Implementation

Beat Estimation

The beat estimating algorithm implemented on the Shield eliminated many of the initial steps described in Scheirer [1] in order to speed up the processing to approximately real time. The first step in the estimation was to downsample by a factor of 128. This factor was selected specifically because it was the largest power of 2 that did not break our Matlab model when tested on a click track.

After downsampling, we calculated the difference between consecutive samples to estimate the derivative. This derivative was then half-wave rectified so that any negative values were set to zero. The next step was to take the FFT of the 1024 sample half wave rectified signal. This signal was then passed through a plethora of comb filters to see which resonated the most. The comb filters were designed in Matlab according to the system function

$$H(z) = \frac{1-\alpha}{1-\alpha z^k}$$

with the feedback gain parameter $\alpha = 0.5$ and the number of samples to delay $k = \frac{Tf_s}{M}$, where T is the beat period, $f_s = 48000$ (the sampling rate) and $M = 128$ (the downsampling factor). For example, a comb filter resonating at 120 beats per minute corresponds to a beat period of 0.5 seconds, requiring a delay of $\frac{0.5 \cdot 48000}{128} = 188$ samples (k is rounded to the nearest integer). The FFT of the filters' coefficients were calculated in Matlab and then multiplied by 2^{14} to convert to the largest possible 16 bit fixed point representation. These frequency domain filter coefficients were then copied directly into the DSP Shield code.

For the final steps, we took the output of the comb filter and summed over all frequencies. The resonant frequency of the comb filter with the largest sum was chosen as the estimated tempo and output to the OLED.

Beat Modification

The basic procedure involves windowing the signal periodically, e.g. every N samples, and spacing these windowed results either closer, e.g. N/4 apart, or farther, e.g. 2N apart. The choice of window width N and the spacing is dependent on the detected pitch of the signal. However, pitch detection was not a feature of this project and can become complicated for a single signal with varying pitch. Beat modification on the DSP Shield was reduced to halving and doubling the beat after verifying the correct spacing in Matlab.

A significant limitation to changing the beat was the amount of memory available in the DSP Shield. The maximum buffer size is 2048, while a song is an average of 3 minutes, sampled at 48 kHz, resulting in millions of samples. The compromise was a Matlab-fed implementation instead of using the DSP Shield's audio codec for real-time beat changing.

Matlab sampled the audio and sent data in buffers of size 400 to the DSP Shield. The manner at which data is sent depended on whether or not the beat was being doubled or halved.

Doubling the Beat

For beat doubling, Matlab sends two buffers of sampled audio and expects one buffer in return. The final two buffers are sent with a differing command number to indicate to the DSP Shield to process it without expecting another pair of buffers as a part of the output. The final two buffers are followed by two outputs instead of one: the output that was from the previous procedure and an output that only has previous buffer. For example, in Figure 2b, if the yellow buffer were the last input, the DSP Shield would know to expect an output only consisting of the yellow buffer's information and not another input. This procedure implies that an even number of buffers are used. An odd number of buffers can be accommodated with another conditional and expected output result in the Shield's sketch. However, a beat increase cannot be done in real-time regardless of DSP capabilities because any increase in beat will result in a shorter duration of audio than what is given, which cannot be without post-processing since on-demand beat-quickening requires future input.

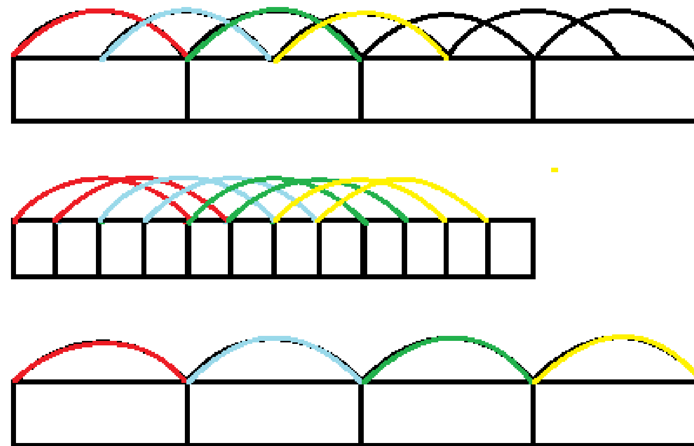

Figure 2: (a) *Top.* Original signal, windowing in color. (b) *Middle.* Signal with beat doubled (c) *Bottom.* Signal with beat halved.

The concept of doubling the beat is creating double the pitch peaks: the input buffers are repeated every fourth of their overall length, effectively creating another beat between originally existing beats. This is shown in Figure 2b, where the center of each window corresponds to a potential pitch peak. Once again, pitch detection was not done in this project so these windows

are chosen solely based on an implementable size in the DSP Shield, i.e. a buffer length less than 2048.

Halving the Beat

To slow down the song by a factor of two, Matlab will send one buffer of data and expect a return of one block for the first buffer of data and two blocks of data for subsequent buffers of data due to the overlapping buffer between two buffers. In the DSP Shield, each input buffer is windowed and sent back to Matlab to store back-to-back in a longer buffer. The DSP Shield will save a copy of the previous input such that when it receives the next input buffer, it will concatenate the last half of the previous input with the first half of the current into to create an overlapping buffer to window. With the exception of the very first input buffer, the DSP Shield will return both the windowed overlap and input to Matlab.

The concept behind halving the beat involves spacing the existing pitch peaks twice as far apart, which is done by placing each window a whole window length apart from one another. Recall that doubling the beat involved spacing of a quarter-window.

The values of the original audio after being sampled in Matlab ranged from values greater than 1 and less than -1, so the audio was pre-processed by normalizing to obtain Q15 representable values. Furthermore, to avoid overflow, the DSP Shield stored the windowed inputs and final overlapped and added outputs to buffers of type long.

V. Results

Beat Estimation

The accuracy of the beat estimator was verified using a metronome. The DSP implementation is less accurate on real songs where the audio input has multiple instruments and melodic lines. The estimated tempo versus actual tempo for the metronome is shown in the figure below.

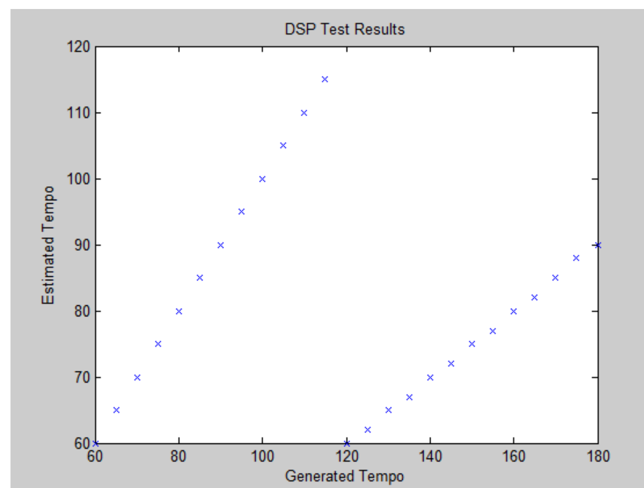

Figure 3: Shows the Estimated Tempo versus the tempo generated by a metronome

These results are accurate for the click track until the tempo exceeds 120 beats per minute, because then the algorithm picks the harmonic at half tempo instead of the true tempo. This ambiguity could have been removed by analyzing the FFT of the FFT for the periodicity of the harmonics.

The implementation on the shield was developed as a tradeoff between processing time, estimation accuracy, and size in memory. We tinkered with the downsampling factor, the length of the comb filters, the number of comb filters, and the number of samples used for each round of analysis. We found that increasing the length of the captured time signal increased the accuracy, but the DSP could only take the FFT of a size 2048 buffer, and this larger buffer had a much longer computation time. The comb filters took up a significant amount of space in memory, but we found that 121 comb filters ranging from 60-180 beats per minute could be included at a filter length of 128 complex elements.

Downsampling by M allowed us to capture samples over a longer range in time, but increasing M to more than 200 severely degraded the accuracy of the algorithm. We ultimately settled on a downsampling factor of $M = 128$ and a buffer size of 1024, which corresponds to roughly 0.36 seconds of audio per analysis. The Shield then takes approximately three seconds to compute and display the beat.

Beat Modification

The proposal set out to modify the beat in real-time using the DSP Shield's audio codec. However, the maximum available buffer size on the hardware was not large enough to implement this. Halving the beat requires either one buffer to store the entire song or dynamically allocating a new buffer for each input, since each input block requires storage of that block windowed and the windowed overlap between it and the previous block. Meanwhile, there is only one buffer freed for each output. Doubling the beat cannot be done in real-time since there is no way for the buffer to predict what will it will be fed in the future to play the input at time, for example, 10 seconds, at time 5 seconds unless the DSP Shield has all the audio at once.

Because of this limitation, the DSP Shield's role is to process data with input and output flow controlled by Matlab. The calculations done on the DSP Shield were verified by Matlab to produce the graphs in figures 4 and 5 below.

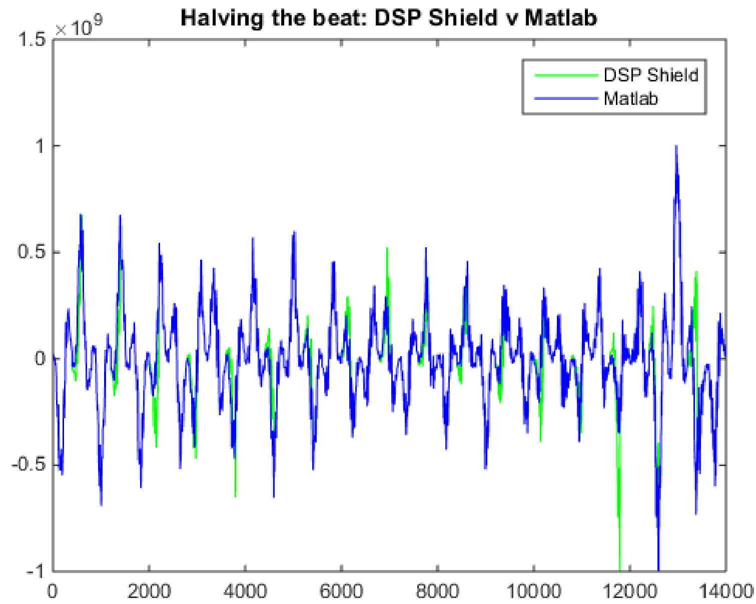

Figure 4: Beat Halving processed via Matlab vs DSP Shield *(X-axis in samples & Y-axis in Q15)*

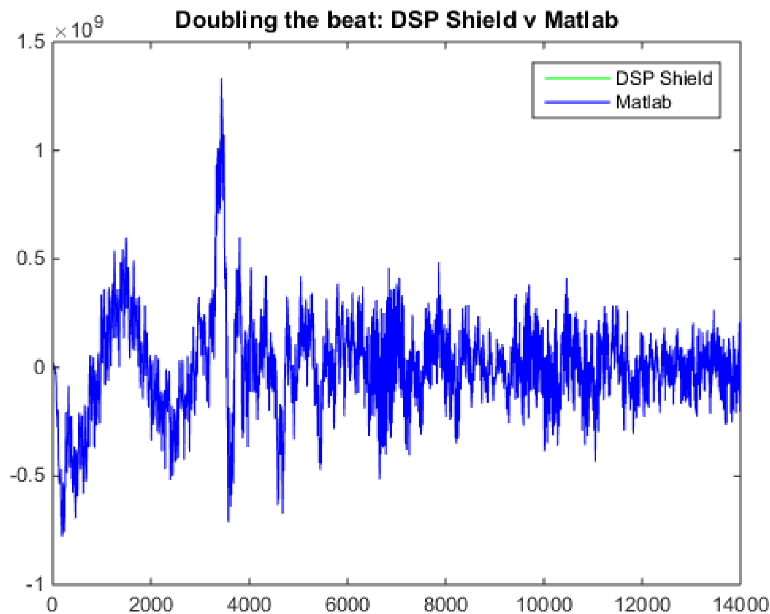

Figure 5: Beat Doubling processed via Matlab vs DSP Shield *(X-axis in samples & Y-axis in Q15)*

VI. Future Work

There are many ways in which we intend to improve and extend this project. The beat manipulation implementation on the shield is currently limited to speeding up or slowing down the audio by a factor of two. For future work, we will change this fixed factor of two to a flexible

input that determines the spacing of overlapped windows. The accuracy of the tempo estimation algorithm can be improved by adding in the initial filter bank recommended by Scheirer. One way to free up memory for additional filters would be to store the numerous comb filter coefficients on the SD card and read them in during run time.

There are two interesting projects that follow directly as an extension of this one. The first is identifying and annotating the phase of the tempo, or the specific location of each beat. This step is critical to matching music to the gestures of a conductor. The second and likely most challenging part of the project will be processing accelerometer data to extract the tempo control from physical gestures. We are excited for the opportunity to continue working with the DSP Shield.

References

[1] E. D. Scheirer, "Tempo and beat analysis of acoustic music signals," Acoustical Society of America, Cambridge, 1998.

Appendix A – Tempo Estimation Code API

List of Functions with Brief Explanations

Function	Explanation
dmaIsr()	*Manage ping-pong input/output buffers. Calls processData() on input/output buffers.*
extract_beat()	*Helper function to execute the comb filtering step. Calls get_energy() for every comb filter.*
get_energy()	*Multiply complex vectors X[k] and H[k] and sum up the product over every frequency k.*
loop()	*Main application loop: Check if the downsampled audio buffer is full and ready to process*. If so, process the data and display to the OLED.
parse()	*Interpret Matlab commands and send back arrays for verification.*
processData()	*Main audio processing function: Copies input audio buffer to output audio buffer. Downsamples the input audio, takes the difference, and half-wave rectifies the result.*
setup()	*Allocate memory for input/output arrays, set buffer initial values to zero, initialize the OLED and Audio Codec.*

List of Functions with Detailed Explanations

dmaIsr – DMA Interrupt Service Routine

Function	interrupt void dmaIsr(void)
Arguments	None
Description	The DMA Interrupt Service Routine interrupts the main loop to process the input and audio buffers. The input data from the Audio Codec is copied to the input[] buffer and the data in the output[] buffer is copied to the Audio Codec output. When the codec has filled up the input[] array, processData() is called.

extract_beat – Helper function to execute all the comb filters

Function	void extract_beat()
Arguments	None
Description	This function runs all 121 of the comb filters stored in "filters.h" and saves the result in the Power[] array. The comb filters range in resonant frequency from 60-180 beats per minute. This function runs the comb filters by calling get_energy().

get_energy – Helper function to return $\sum_k |X[k]H[k]|$

Function	long get_energy(int* X, int*H, int nx)
Arguments	X[nx] – The pointer to the complex input signal array (after downsampling, differencing, and half-wave rectifying) in the frequency domain. The real values of X are stored in sequential even positions and the imaginary values of X are stored in sequential odd indices. H[nx] – The pointer to the array with the complex coefficients of the comb filter in the discrete frequency domain. The real values of H are stored in sequential even positions and the imaginary values of H are stored in sequential odd indices. nx – The length of the input arrays to multiply. Twice the length of the number of complex elements in the filter.
Description	This function takes two signals in the frequency domain, multiplies them together and sums up the absolute value of the result over all frequencies. It performs the operation $\sum_k |X[k]H[k]|$, but scales the result down by a total facto of 2^{14} during intermediate calculations to prevent overflow. This function returns the resulting sum of products as a 32 bit long.

loop – Main Application Loop

Function	void loop()
Arguments	None
Description	This function checks to see if the buffer containing the downsampled, differenced, and half-wave rectified audio input is full. If so, it processes the audio by taking the FFT of the signal and calling extract_beat() to multiply the comb filters. When the outputs of the comb filters have been determined by get_energy() and stored in the Power[] array, this function finds the index of the comb filter with the maximum output. The resonant frequencies of the comb filters range from 60-180, so 60 is added to the index of the maximum to get the estimated tempo of the audio. The estimated tempo is then displayed on the screen.

parse – Interpret and execute Matlab commands

Function	void parse(int c)
Arguments	c – the Matlab command received by the getCmd() function.
Description	This function uses a switch statement to determine which course of action to take depending on the command received. The commands can send the arrays of the audio input and intermediate calculations to

Matlab to verify the functionality of the intermediate steps in the algorithm. The code fails to connect to Matlab when the comb filtering step is included, so this portion of the code is not executed. However, it was very useful for the initial stages of debugging.

processData – Main audio processing function

Function	void processData(const int *inputLeft, const int *inputRight, int *outputLeft, int *outputRight)
Arguments	inputLeft[BufferLength] – pointer to the left audio input signal inputRight[BufferLength] – pointer to the right audio input signal outputLeft[BufferLength] – pointer to the left audio output signal outputRight[BufferLength] – pointer to the right audio output signal
Description	This function copies the input buffers to the output buffers for the play of constant audio input. This function also contains the initial processing steps of the beat estimation algorithm. If the downsampled input buffer is not yet full, this function grabs every 128th sample, takes the difference of consecutive downsampled inputs, clips all negative values to zero, and stores the result in the dec[] array. The overall effect is that dec[] is the downsampled, differenced, and half-wave rectified output of the audio input.

setup – initialization function

Function	void setup()
Arguments	None
Description	This function allocates memory for all the global arrays and initializes the values to zero. It also initializes the board’s display and audio codec. When everything has been initialized, the board prints “Process ON” to the display.

Appendix B – Tempo Manipulation Code API

List of Functions with Brief Explanations

fillInputs()	*Process the data for increasing the tempo by 2*
halfBeat()	*Process the data for slowing the tempo by 2*
loop()	*Main processing function*
processEvenOutputs()	*Overlap and save the even windows*
processOddOutputs()	*Overlap and save the odd windows*
setup()	*Initialize buffers, OLED, and serial connection*

List of Functions with Detailed Explanations

fillInputs – Buffer filling for beat doubling

Function: void fillInputs(const int *input)

Arguments: input[dataLength] – pointer to the input signal from Matlab

Description: For doubling the beat (halving the beat has a different method of filling inputs due to the fewer amount of buffers required to store for each output). Fills in appropriate buffer i.e. will fill in oldest buffer with the newest data. At start of code, will fill in1, then next time Matlab sends a buffer, will fill in2, so on and so forth.

halfBeat – Buffer filling for halving the beat

Function: void halfBeat(int command, const int *input)

Arguments: command – the most recent Matlab command (10 for regular data transmission, 11 to indicate this is the first buffer being sent)

input[BufferLength] – pointer to the input signal from Matlab

Description: This function not only fills in the output buffers but will also output a windowed copy of the input and overlap. If the command is 11, this indicates the very first input from Matlab which doesn't have an overlap to window and output.

loop – Main processing function

Function: void loop()

Arguments: None

Description: This function uses serial commands to receive the input signal from Matlab in buffers 400 samples long. If the halve-the-beat flag is raised, the function will slow the tempo by 2. Otherwise it will increase the tempo by 2. The result is then sent back to Matlab.

processEvenOutputs – function to overlap and save even windows

Function	void processOddOutputs(int dataLength, int command)
Arguments	dataLength – the length of one buffer of the audio signal (400 for this implementation) command – the most recent Matlab command (10 for regular data transmission, 11 to indicate this is the first buffer being sent)
Description	Similar to processOddOutputs with the exception of the algorithm accounting for a different order of the windowed inputs during the overlap and add.

processOddOutputs – function to overlap and save odd windows

Function	void processOddOutputs(int dataLength, int command)
Arguments	dataLength – the length of one buffer of the audio signal (400 for this implementation) command – the most recent Matlab command (10 for regular data transmission, 11 to indicate this is the first buffer being sent)
Description	Sends the 1st, 3rd, 5th, etc outputs to Matlab along with the length of the output, dataLength. The command is necessary to indicate whether or not the last input is being fed into the DSP Shield because the resulting output should not expect future inputs overlapped in that case. pin3 and pin4 are initialized to zero so the first output, which doesn't include previous input, can still use the same overlap-add algorithm to produce appropriate output.

setup – Function to initialize buffers, OLED, and connect to Matlab

Function	void setup()
Arguments	None
Description	This function allocates memory for the buffers and initializes all their values to zero. It also initializes the OLED and connects to Matlab.