Wireless Real-Time Drum Triggering

A Design Project Report

Presented to the School of Electrical and Computer Engineering

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical Computer Engineering

Submitted by
Curran Sinha
MEng Field Advisor: Bruce Land
Degree Date: December 2017

Abstract

Master of Engineering Program

School of Electrical Computer Engineering

Cornell University

Design Project Report

Project Title: Wireless Real-Time Drum Triggering

Author: Curran Sinha

Abstract: With the rise of electronic music and home production in the music industry, drum triggering is gaining popularity and is now used widely by professionals and amateurs in live and studio environments. Drum triggering allows a musician to sample various sounds using mechanisms to detect a drum strike, which then triggers a midi sample. Typical setups involve sensors or "triggers" for each drum that are wired to a drum module, which either directly triggers samples or generates midi data. There are no current products that allow for cheap wireless triggering, which are portable and easy to setup. This design project aims to fill that gap in the market by developing a system that uses just one microphone and a microcontroller to distinguish between multiple drums.

Executive Summary

Drum triggering is a common tool that allows musicians to trigger arbitrary samples like percussion instruments or melodic textures. Whether it is being using in a live or studio setting, it adds another dimension to the instrument by layering different voices on top of typical drum set sounds. Even though it has been around for decades, in the past few years the rise of electronic music and the ease of producing music without a studio have pushed more drummers to think about exploring triggering. The standard system for triggering requires a separate piece of hardware for each drum that needs triggering ability, and all of those need to be wired to a central drum module to generate the sounds. The setup and convenience of the current technology is less than ideal. New products have come out that try to fix that, but they all are lacking in at least one way that makes them undesirable. Either they cost more than a casual drummer looking to explore triggering is willing to pay, or they do not have the functionality to trigger different drums, or they are inconvenient to transport and setup. This design project aims to fix that.

This project is a proof of concept for a product that uses just a microphone and a microcontroller to trigger multiple drums in real time. The system is small enough to easily carry around and the set up required is just placing the system near the drums and training each drum. By implementing the algorithm that performs the training and classification in Matlab, I was able to prove the feasibility of this design. The algorithm is easily transferable to a microcontroller to execute the same algorithm in real time and trigger sound accordingly.

There are quite a few significant problems with triggering drums using just one microphone. First off, to maintain real time response, the entire system must detect a drum strike and trigger the audio sample within 15 milliseconds. That means only about 10 milliseconds of a sample can be used to analyze, and that short of a sample does not contain a large amount of information about the drum. As well, once a strike is detected and a sample is recorded, the features that uniquely distinguish the sample from other drum samples are used in a K-nearest neighbors algorithm to allow the user to play a drum and trigger any sound.

The final results are extremely promising. Given a training set to train on, the algorithm correctly identified 93% of the test samples accurately. For version 1 of the design, this system has plenty of room to grow and improve, and hopefully will eventually lead to a real product.

1. Introduction

1.1 Motivation

Drum triggering is used widely by musicians all over the world to add electronic and sampled sounds to their playing while using an acoustic drum kit. This is not just a fad. Electronic music that uses non-acoustic sounds is gaining popularity and musicians are expecting these sounds to be recreated in live settings or studio settings. As well, drummers are starting to experiment with creating full songs on their own. This involves triggering the melody, harmony and bass lines, while still playing the drum part. I have been playing drums for 8 years and at Cornell I'm currently part of a jazz combo, jazz big band, and a funk/R&B band, so when I started to think of a final design project, I immediately thought about designing something related to drums or music.

Recently, a product was released called Sensory Percussion (sunhou.se) that uses sensors on a drum to detect different hits depending on where the drum is struck and trigger audio samples accordingly. The sensors can detect many sounds (7+) like center of the head, edge of the head, rim of the drum, rim shot (hitting the rim and drumhead at the same time) at the center of the head, rim shot at the edge of the head, and a few more. Each of these sounds can then be assigned to a sample and it gives the drummer new sounds to create music with. Along with this, the company has software that allows for more advanced features like blending between two different sounds if you hit in between regions.
This product inspired me because it combined my love of drumming with hardware and signal processing. I remember when the product first came out, the drumming community was extremely excited and they still are, but there are a few flaws with the system. The first issue is that these sensors work with just one drum and if the user wants to trigger with more than one drum, they need another sensor. This is limiting because most drummers are used to creating beats using separate sound sources and thus different limbs, which would require more sensors. This is related to another issue, which is that the software and one sensor costs $700. Each additional sensor is $300. For most musicians, this is an expensive part and creates a high barrier to enter the market. The last issue is that setting up the system can take a while because one needs to attach all the sensors and then wire

each sensor up to an audio interface. For recording purposes this is fine, but if someone wants to quickly setup these sensors it is inconvenient. I used these “downsides” as inspiration while developing my project.

1.2. Previous Work

My main goals for this project were to design a portable, easy to setup, reliable drum triggering mechanism that did not cost too much and had no noticeable latency. The initial thought I had was to do this using only a microphone and a microcontroller. Taking a look at some similar products on the market, I discovered Versatrigger: a wireless drum triggering mechanism. Despite meeting a few of my requirements of being wireless and having low latency, the set up process is tedious and laborious. This trigger needs to be installed inside the drum which make it non portable and hard to set up. As well, one trigger and the hub, which receives the data from the trigger, costs around $120.

Figure 1. Versatrigger

Another product that was recently released is called the Electronic Acoustic Module (EAD) from Yamaha. They use a similar principle to this design project by using just one unit, but it does not trigger drums separately. Instead, it processes the sound coming from the microphone(s) with certain sound effects to add another layer of sound. This is a step in the right direction and confirms that there is growing need for a product like the one I am developing.

Figure 2: Yamaha EAD

2. Implementation

2.1 Possible Solutions

Tackling this problem is not simple because it has not been done before and can vary depending on the environment. One approach is to use wireless piezo sensors with a simple radio protocol such as Zigbee, which would transfer the sensor readings to a central hub that generated midi data. This is relatively easy to setup and probably would accomplish the job, but then issues might arise with price because of various different parts and with battery life because radio protocols take up a decent amount of power regardless of how little information is being transferred. Another option is to use two microphones in front of the drum set to gain not only audio information, but also spatial locality of the drum set. This is promising, but a better idea with regards to that is just to place two microphones near the middle of the drum set and point the microphones in opposite direction. This would help determine the direction the sound is coming from, which can be combined with the audio processing to determine the drum. This idea seemed to meet all my requirements, but I decided to start with just one microphone and rely on audio processing to see if that would be possible.

2.2. System Requirements

The current design assumes an environment consisting of only a drumset, where the user is triggering sounds using multiple drums, but not playing cymbals. In order to use triggering

in a live environment (band performance), a trigger per drum is necessary because of all the noise from other instruments. The scope of this project is catered more towards drummers who want to explore new sounds and the possibilities of triggering. As well, the system is not expected to produce 100% accuracy because of the simplicity and cheap price of the design. I would hope for accuracy above 80% though, where most hits are registered and trigger the correct sound, but there will be a few misplayed sounds. More iterations of the design will help increase the accuracy though.

The main approach to this project is to constantly sample audio from the microphone, use a simple power or amplitude threshold to detect the start of a hit, run the audio through an FFT, send the spectrum and the actual sound wave into a classification network, and trigger an audio sample accordingly. The training of the classification network will happen first and it will generate a classification model for the drums.

2.3. Issues

One main issue is the frequency spectrum of a drum is not like a piano or trumpet note, because there is not always a clear primary frequency and if there is there are many other harmonics that appear. As well, depending on how hard the drum is hit, the frequency spectrum can change. On a related note, in order to maintain real time response, the time between the drum hit and the midi trigger must be less than 15 milliseconds because real time audio perception is between 10-20 milliseconds. This means the actual sample that will be used can only be around 8-10 milliseconds because it takes time to process the sample and generate a midi trigger. With such a short sample, the tone of the drum will not be completely present. It instead will be a combination of the tone of the drum and the impact of the stick hitting the head. A comparison of a full drum sound wave to what 10 milliseconds of a drum sound is shown in Figure 3 and 4.

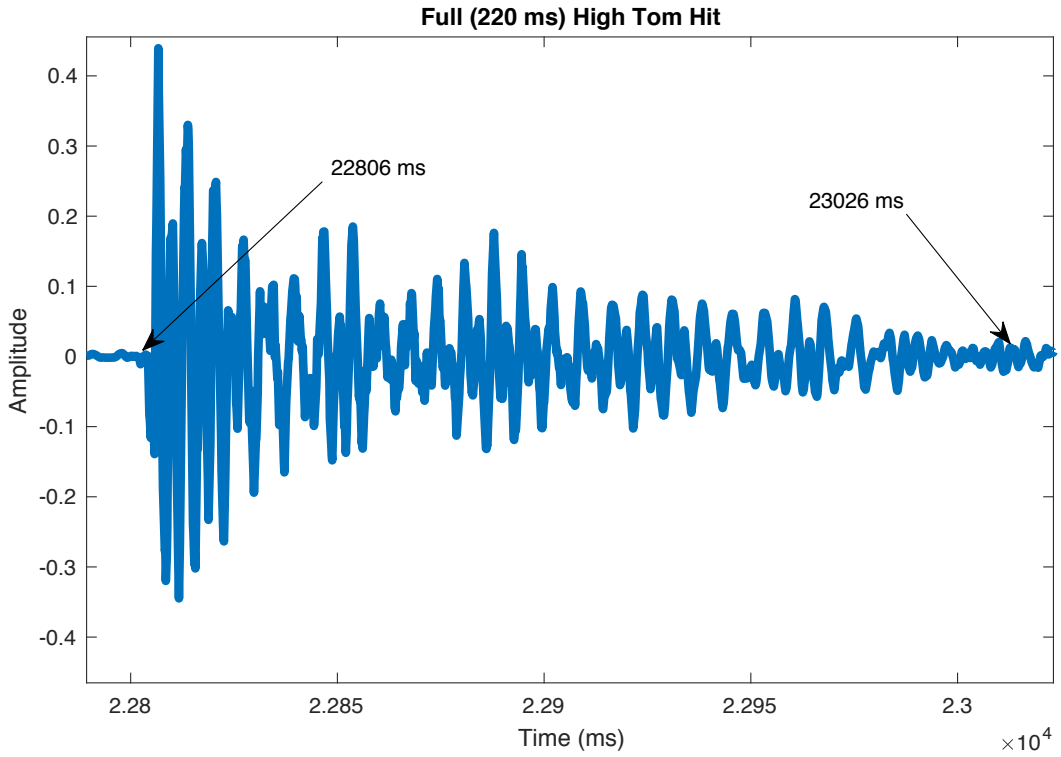

Figure 3. Full Tom Sound Wave

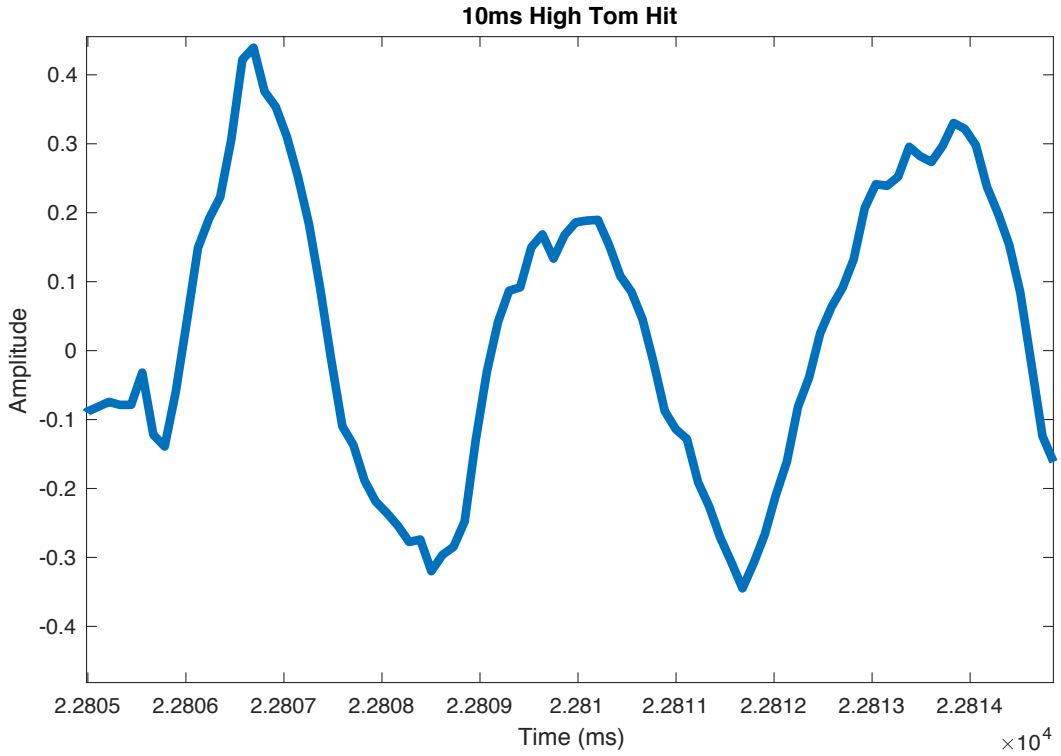

Figure 4. 10ms Tom Sound Wave

Another interesting issue is the classification mechanism. It needs to be robust and able to adapt to various different drums, but it cannot take up too much space or take too long computationally.

2.4. Software Design

At its current state, my project is more of a proof of concept for a real time system that proves the feasibility of using just one microphone to trigger multiple drums. The system is implemented in Matlab, but for real time execution it needs to be run on a microcontroller. It is easy to port the system to one because all the steps are relatively basic computations. In the next sections I will elaborate on each step of the design.

2.4.1. Audio Input

The system samples audio at 8khz because the majority of drum frequencies are present below 1000 kHz. For the prototyping I was doing with Matlab, I ended up using a Zoom H4N microphone, which has two stereo microphones. It samples as 44100, but down sampling that is simple. For the actual microphone circuitry that would be connected to the microcontroller, a high pass and low pass filter will be added along with an amplifier. The low and high pass will help filter out any noise that is present, while the amplifier allows the user to set the gain depending on where the microphone is placed and how loud they are playing. The audio samples are stored in a circular buffer to allow the system to have some previous recordings to take when a drum strike it detected.

2.4.2. Detecting Drum Strike

In order to detect the start of a drum strike, the system needs to detect a sudden increase in amplitude above a certain threshold, but it must not trigger on sporadic noise that might be present in environment. The basic way to do this is to generate an envelope of the signal that tries to outline the amplitude of the signal. There are two ways I tried to approach this: an exponential moving average (1 pole IIR) and a moving average window (. Both approaches are shown on a sample drum strike in Figure XXX. For the exponential moving average, it uses this formula:

$$E(i+1) = \alpha * \big(input(i)\big) + (1-\alpha) * E(i)$$

An n-window moving average uses the follow equation:

$$E(i) = \frac{1}{n}\sum_{i=0}^{n-1} input(i))$$

The input(i) to the formulas can be either the raw microphone inputs or the absolute value of those results. Using the absolute values allows the envelope to respond slightly faster, but for this specific application, it does not affect the result significantly. Also, the alpha term for the exponential moving average can easily be adjusted to change the response time, which can be useful when tuning the system. In terms of implementing an envelope on a microcontroller, the exponential moving average is much simpler, so I chose to work with that.

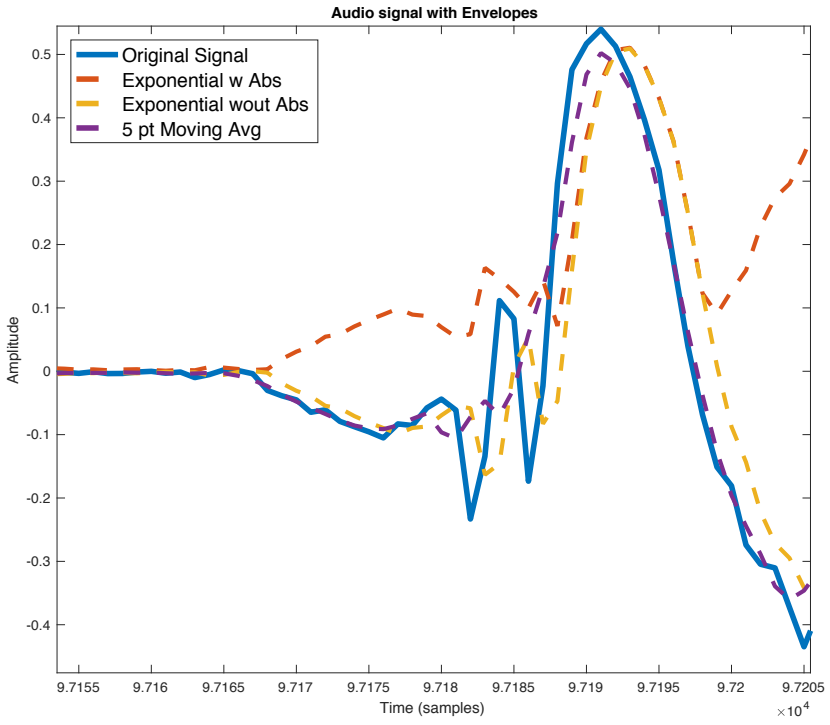

Figure 4. Drum Strike with Filtered Signals

2.4.3. Feature Extraction

After a drum strike is detected, the system takes note of that specific index and saves around 2 milliseconds before the detection as the start of the sample. It then proceeds to collect samples for ~8 more milliseconds, which at 8khz is 64 samples. This produces a 10ms sample to process. These offsets were chosen by analyzing drum strikes and finding the smallest index that was needed to obtain the start of the drum hit for most samples, and then the rest is allocated for after the strike to gain as much information as possible about the drum.

Next, the signal is processed to extract the important features. A large problem with extracting features is that there is no definition for what is important for a specific drum. A useful method to determine this is by visualizing the features as a K-nearest neighbors classification algorithm. In order to do that, I only use three features because it is easy to graph the features for all the drums and see if there is a clear separation between them. I will talk more about the tradeoffs between the features in the classification section. The most obvious feature to extract is the frequency response of the drum since a bass drum clearly has a lower pitch then a small tom drum and it can be applied to drums no matter how big or small they are. To do this the system generates the discrete Fourier transform of the 10-millisecond sample using the fast Fourier transform. The output is a graph that represents the amplitudes of each frequency range. Another factor to consider is whether the filtered signal or the original signal should be passed into the FFT. The original signal seems like the better option comparing Figure 5 and 6 because it has larger peaks, but issues can arise if there is significant noise that adds in some frequency components. However, since the noise will show up in frequency bins separated from the main sample I chose to use the original signal. It is important to note that raw amplitude data is not useful because the system needs to function regardless of the volume of the drum hit. To deal with this, the system uses features based around relative data like the order of peaks or using amplitude ratios relative to the highest peak.

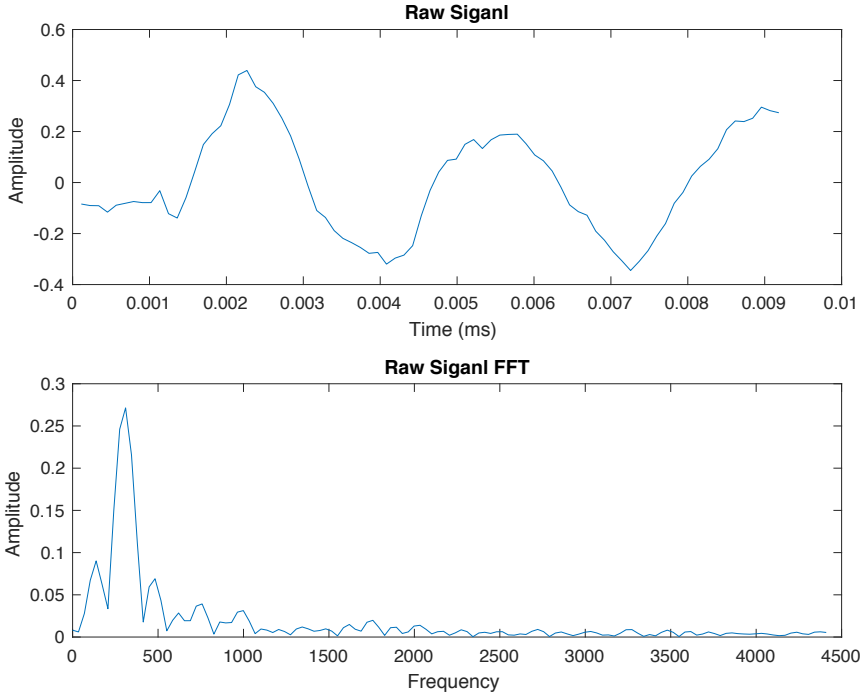

Figure 5. Original Signal Response

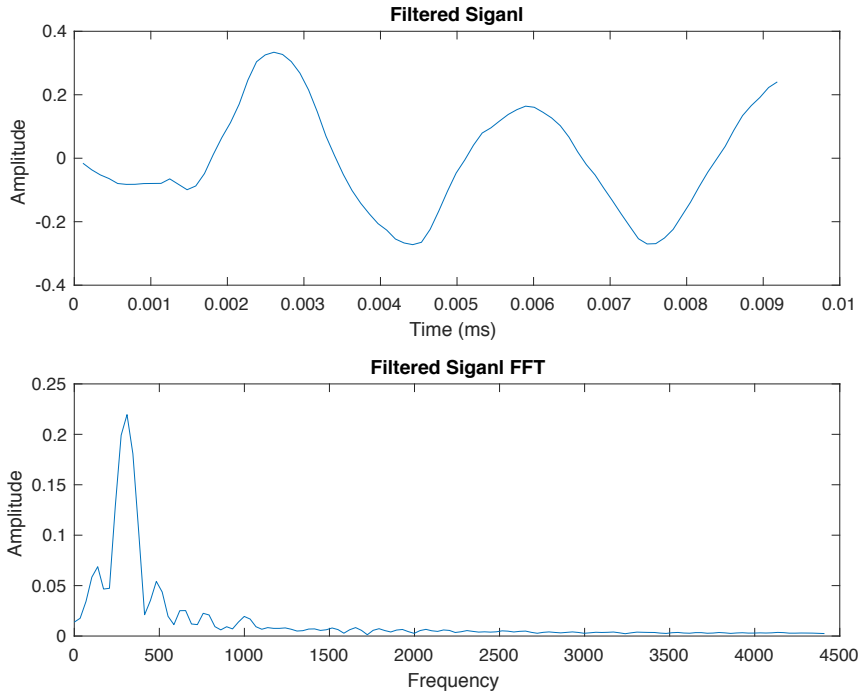

Figure 6. Filtered Signal Response

Once the frequency response is available, there is a large variety of features that can be extracted. A few that I explored included the highest peak in the frequency graph, the number of frequency peaks, the time difference between peaks in the time domain and the ratio of the highest peak to the second highest peak in the time domain. These are just a small selection I thought of by observing the different responses of four drums that can be seen in Figure 7.

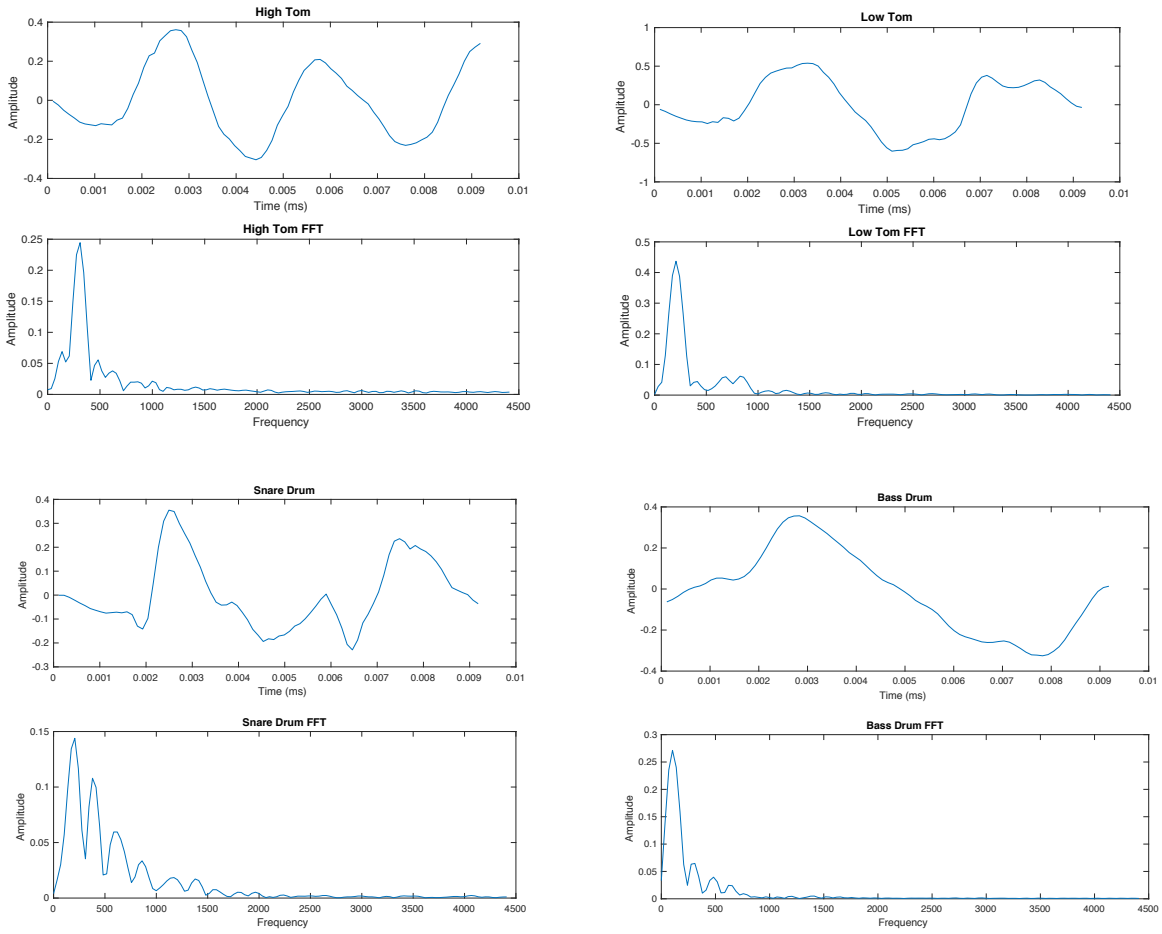

Figure 7. Different Drum Responses

2.4.4. Training and Classification

This system classifies drums using a simple K-nearest neighbors classification algorithm. Initially, during the training, the user plays each drum 10 or 20 times and the system saves the three features for each drum hit. Then, when a new sample is obtained, the three

features are calculated and then the distance to each of the training points is calculated. The sample is classified as the majority of its "k" nearest (in terms of Euclidian distance) neighbors. This is a fairly simple machine-learning algorithm, but that helps in terms of comprehending the reasoning for its classifications. This makes it easier to tune the algorithm so it is catered towards drums. While testing out different features, I would generate graphs of all of the training points and the goal was to separate each drum as much as possible. Figure 8 shows one of the first ones that worked decently. The features are the peak frequency, the number of peaks in the time domain, and the ratio between the maximum peak and the last point in the time domain. For the most part each drum is in a specific region, which is what matters.

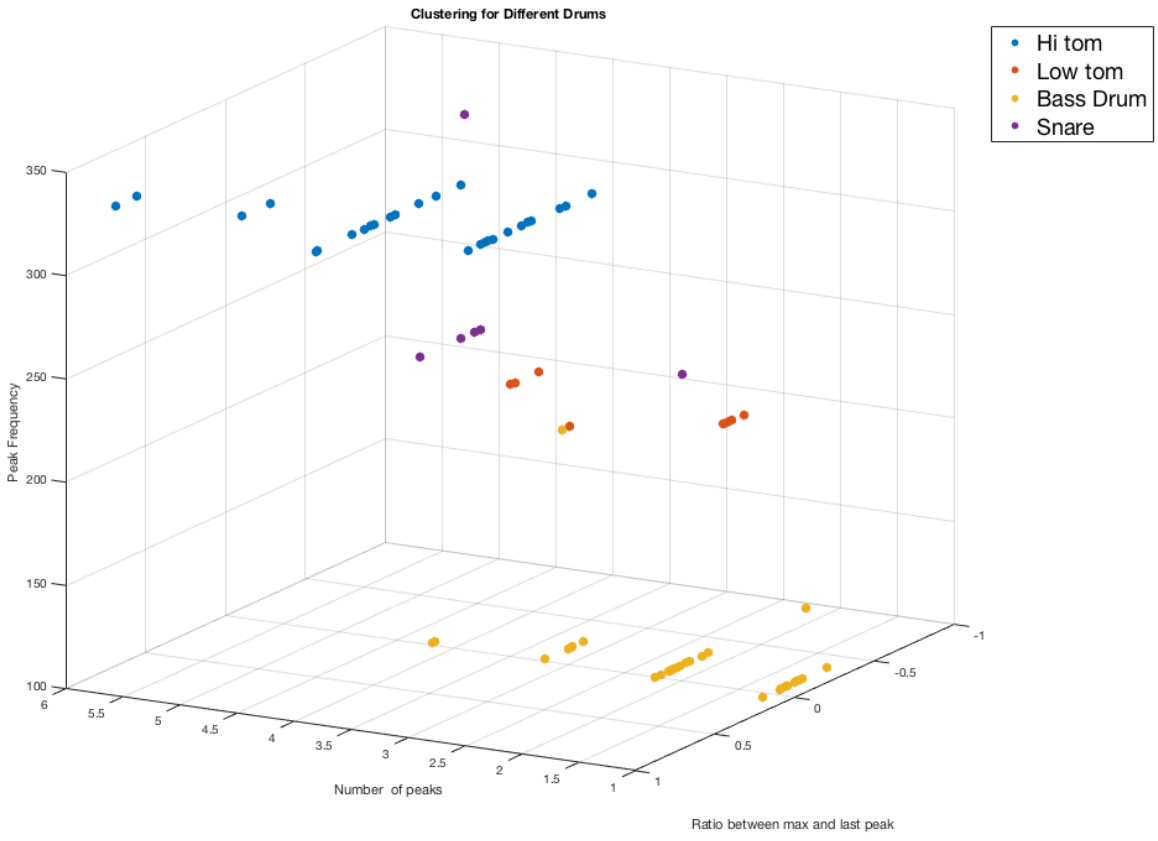

Figure 8. First Working KNN Model

Through more analysis of the different drums' time and frequency responses, and musical intuition of what typical drums sounds like, I generated the KNN graph shown in Figure 9. This is one of the best I obtained. It uses the peak frequency, the second highest frequency, and the ratio of the max peak and the last point. Other than one or two outliers, the high tom and bass drum are extremely separated, and low tom and snare drum are slightly separated. I tried to compensate for that by using a new feature that was fairly specific. The feature was the number of frequency peaks that were equal or greater than half of the highest magnitude peak, which represents spectral density in a way. This is because I noticed the snare drum had a lot of frequency noise and because of that had a much busier frequency spectrum despite having a similar first and second peak. The final KNN is shown in Figure 10. The final system implements the classification using the peak frequency, the second highest frequency, and the number of frequency peaks greater or equal to half the magnitude of the highest peak. Many drum strikes mapped to the same point, so I sized each point according to the number of instances at each point. It is of course possible to keep optimizing for the training set, but that can easily lead to over fitting, which I felt I was starting to do. By analyzing the specific drums in the training set, and trying to think of features that would fit that specific sound wave can lead to results that do not adapt well to other drums.

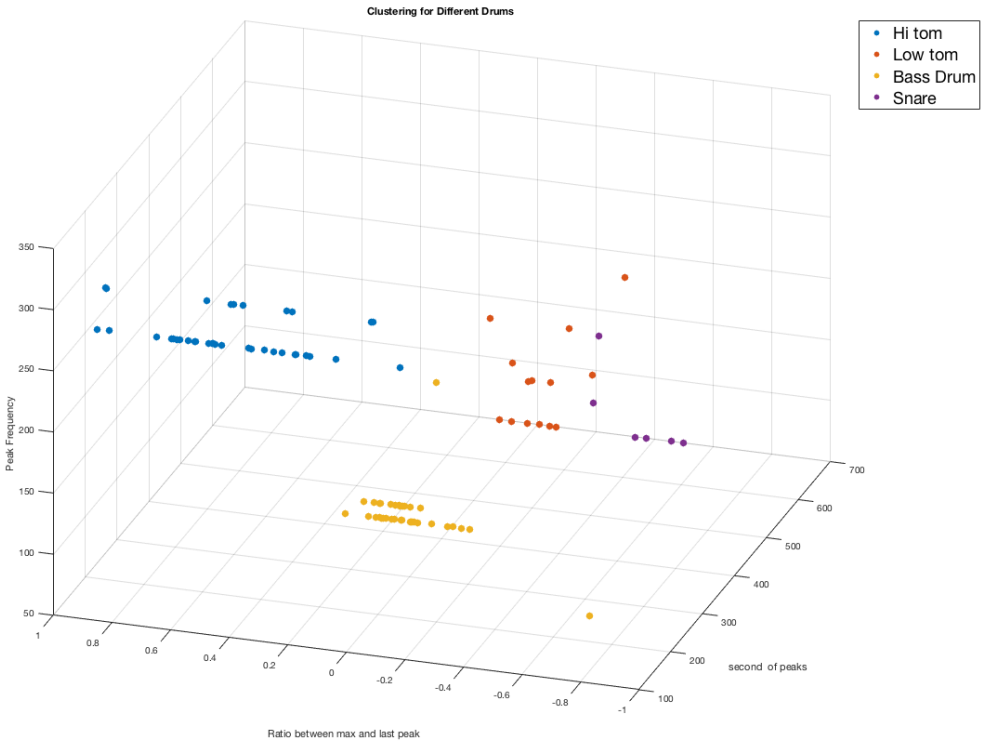

Figure 9. Second Working KNN Model

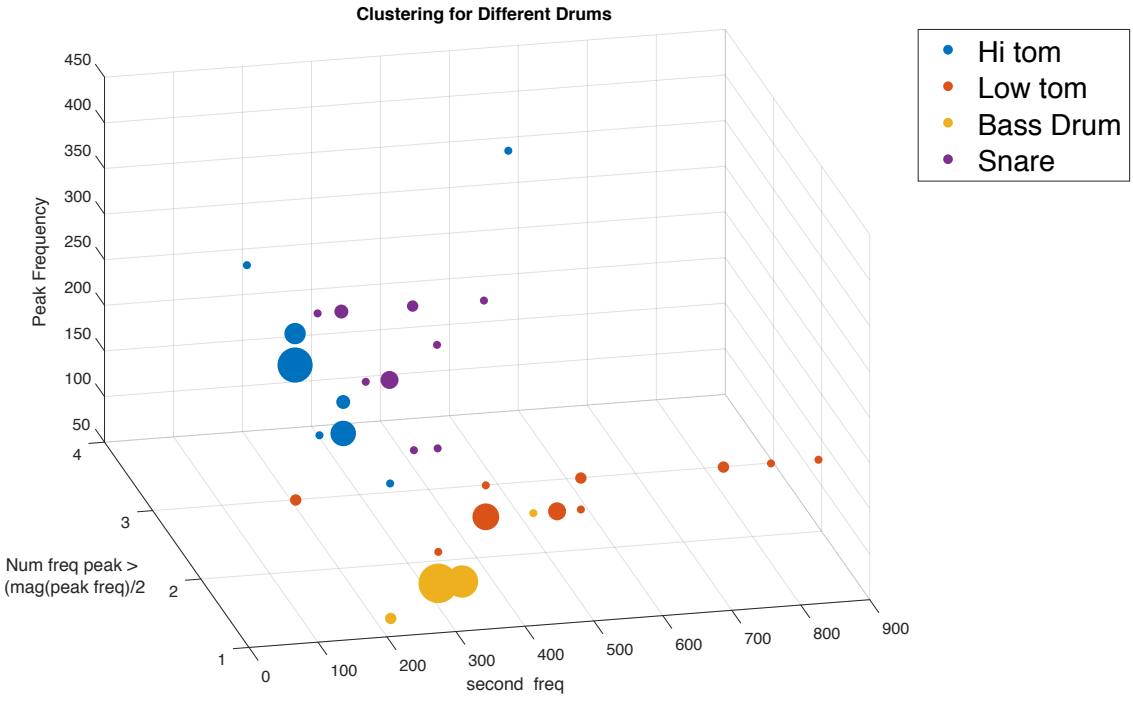

Figure 10. Final KNN Model

3. Results/Discussion

Overall, I was able to obtain decent results that performed well with a given training and testing set. These were processed in Matlab (not in real time), but the way each audio signal is processed simulates a 10ms sample to prove its feasibility on a microcontroller.

Training Set	**Number of Samples**
High Tom	60
Low Tom	20
Bass Drum	203
Snare Drum	91

The number of samples here varies so much because of the way the microphone is positioned; certain drums are louder than others, and since each is using the same threshold, not all of them are registered as drum strikes despite the recording containing similar amounts of each drum. This needs to be addressed somehow. After running them through the KNN algorithm with K = 3, the resubstitution loss, which is how the training set performs (error rate) on a KNN algorithm, was 0.058. A low 6% means that the data has separated the drums fairly well.

Testing Set	**Total Samples Detected**	**Total Classified Correctly**	**Accuracy**
High Tom	86	83	0.965
Low Tom	37	32	0.865
Bass Drum	252	248	0.984
Snare Drum	145	124	0.855
Average	*520*	*487*	*0.937*

Overall, these numbers show that the system works for the most part and achieves accuracy beyond what I expected. In terms of usage for exploring triggers, this type of

accuracy for a first version is a great starting point, but still leaves a lot of room for improvement.

These results can vary quite a bit depending on a few variables that are not set in stone. The main things are the threshold that can change how many drum strikes are detected, the value of K, which greatly affects the accuracy if the model is not well fitted, and lastly the features that are chosen to create the KNN model. While the features for all drums are similar at their core, depending on the environment, tuning of the drums, volume, sticks that are used, and many other factors, these features may not represent those particular drums best. For this reason, this application might be more suited for a neural net that accepts the entire waveform to classify the drums. As well, computationally speaking, KNN is intensive compare to a simple neural network and once the system moves to a microcontroller, this could pose an issue.

4. Conclusion and Future Work

Overall, this project was an interesting and successful exploration of wireless drum triggering using a microphone. The goal of developing a cheap, reliable, easy to setup system was achieved in theory (Matlab) and I would have loved to implement this on the microcontroller, but because of a lack of planning and unforeseen difficulties I was unable to. The accuracy of the system was proven to be high enough to use as a musician, and there are significant improvements that can be made.

The obvious next step is to move this over to a microcontroller so it can actually trigger sounds in real time. I have already started that process. I have the full microphone setup, connected to a PIC32 that also is connected to a Teensy 3.2 for debugging purposes. I already have that code sampling from the microphone, performing an envelope filter, and detecting and capturing 10 milliseconds of a drum strike. The next steps involve putting that sample through an FFT in fixed point so it can execute it in a few milliseconds, and then also implementing some type of KNN or neural network on the microcontroller. This

is not too common because of space constraints, but I'm confident there will be enough space because the audio processing code is pretty memory efficient.

Other steps include implementing the same functionality with two microphones. With this type of special locality, detecting which drum is played becomes much more trivial and can lead to accuracies above 98%. That would be exceptional for even professional musicians and could be developed into a product. A significant amount of work would need to be dedicated to choosing the right classification algorithm that can work for various environments, is not too computationally intensive, and is implementable on a microcontroller with space restraints. The other issue is overlapping or simultaneous drum hits. This is probably solvable using two microphones and some more frequency analysis. Overall, this design project has shown me that a simple solution can achieve the functionality of an accepted standard in a much cheaper way. It as well has inspired me to try to develop this into a product because now I am confident it will work and if I would love to use it, I believe there will be other drummers who would as well.

Appendix

Matlab Code

```
%Import samples
hitom_train_o = importdata('longSamp/train/hitom.mp3');
lotom_train_o = importdata('longSamp/train/lotom.mp3');
bass_train_o = importdata('longSamp/train/bass.mp3');
snare_train_o = importdata('longSamp/train/snare1.mp3');

hitom_test_o = importdata('longSamp/test/hitom.mp3');
lotom_test_o = importdata('longSamp/test/lotom.mp3');
bass_test_o = importdata('longSamp/test/bass.mp3');
snare_test_o = importdata('longSamp/test/snare1.mp3');
all_test_o = importdata('longSamp/test/all.mp3');


Freq = 44100;
Freq_d = 5;
Fs = Freq/Freq_d; % 8.8kHz

%downsample Samples
hitom_train = downsample(hitom_train_o.data(:, 1), Freq_d);
lotom_train = downsample(lotom_train_o.data(:, 1), Freq_d);
bass_train = downsample(bass_train_o.data(:, 1), Freq_d);
snare_train = downsample(snare_train_o.data(:, 1), Freq_d);

hitom_test = downsample(hitom_test_o.data(:, 1), Freq_d);
lotom_test = downsample(lotom_test_o.data(:, 1), Freq_d);
bass_test = downsample(bass_test_o.data(:, 1), Freq_d);
snare_test = downsample(snare_test_o.data(:, 1), Freq_d);
all_test = downsample(all_test_o.data(:, 1), Freq_d);

drum_names = {'hitom', 'lotom', 'bass', 'snare'};
drum_train = {hitom_train, lotom_train, bass_train, snare_train};
drum_test = {hitom_test, lotom_test, bass_test, snare_test};

% Set Constants
tiny = 0.3;
threshold = 0.23;



j = 1;
class = "";
knn_x = [];
knn_y= [];

counter = 0;
total = 0;


for j=1:length(drum_train)
    class = drum_names(j);
    testThing = drum_train{j};  % actual sample
    lengthH = round(length(testThing));
    envelope = zeros(1, lengthH);
%     envelope1 = zeros(1, lengthH);

    %instantiate feature vectors
    numPeaks = [];
    peakRatio = [];
    peakFreq = [];
    secFreq = [];
```

```
    numfPeaks = [];
    total = 0;

    % Create envelope Filter
    i = 1;
    while (i < (lengthH))
        envelope(i+1) = (tiny*testThing(i)+ (1.0-tiny)*envelope(i));
%         envelope(i+1) = (tiny*abs(testThing(i))+ (1.0-tiny)*envelope(i));
        i = i+1;
    end

    i = 1;

    while (i <= (lengthH))
        if (envelope(i) > threshold && counter <= 0)

            start = i - 15;
            last = i + 65;
%             sample = envelope(start:last);
            sample = testThing(start:last);

            % Time Based Features
            [peaks, loc] = findpeaks(sample, 'MinPeakDistance', 10);
            numPeaks = [numPeaks, length(peaks)];
            [sortedPeaks, sortedPeakI] = sort(peaks, 'descend');
            peek = (sample(end)/sortedPeaks(1));
            peakRatio = [peakRatio, peek];

            % Frequency Based Features
            L = length(sample);
            NFFT = 2^nextpow2(L); % Next power of 2 from length of myRecording
            Y = fft(sample,NFFT)/L;
            f = Fs/2*linspace(0,1,NFFT/2+1);
            max_fft = max(abs(Y(1:round(NFFT/2+1))));
            [fpeaks, floc] = findpeaks(abs(Y(1:round(NFFT/2+1))));
            [sortedfPeaks, sortedfPeakI] = sort(fpeaks, 'descend');
            peakFreq = [peakFreq, f(floc(sortedfPeakI(1)))];
            secFreq = [secFreq, f(floc(sortedfPeakI(2)))];
            [nfpeaks, nfloc] = findpeaks(abs(Y(1:round(NFFT/2+1))), 'MinPeakHeight',
max_fft/2);
            numfPeaks = [numfPeaks, length(nfpeaks)];

              % graphing for debuggin
%             figure();
%
%             subplot(2,1,1)
%             plot ((1:length(sample))/Fs, sample);
%             subplot (2, 1, 2);
%             plot(f,2*abs(Y(1:NFFT/2+1)))
%             title(["thing " num2str(i)]);
%              end

            % add feature vector to KNN data
            knn_x = [knn_x; [peek, f(floc(sortedfPeakI(1))),
f(floc(sortedfPeakI(2)))]];
            knn_y = [knn_y, class];
            total = total + 1;
            counter = 100;
            envelope(i) = 0;
        end
        if (counter > 0)
            counter = counter - 1;
        end
        i = i +1;
```

```
    end


    disp (['total training for ' drum_names{j} ' = ' num2str(total)]);
    xx = peakRatio;
    yy = secFreq;
    zz = peakFreq;
    figure(70)
%     scatter3(xx, yy, zz, 60, 'filled');

    %Engine
    [uxy, jnk, idx] = unique([xx.',yy.', zz.'],'rows');
    szscale = histc(idx,unique(idx));
    %Plot Scale of 25 and stars
    scatter3(uxy(:,1),uxy(:,2), uxy(:,3),'o', 'filled', 'sizedata',szscale*25)

    title("Clustering for Different Drums ");
    xlabel('Peak between max and last peak');
    ylabel('Second Frequency peak ');
    zlabel('Peak Frequency ');
    legend({'Hi tom', 'Low tom', 'Bass Drum', 'Snare'}, 'Fontsize', 18);
    hold all;

end

%KNN Model
mdl = fitcknn(knn_x, knn_y);
mdl.NumNeighbors = 3;
rloss = resubLoss(mdl);



% Training

for j=1:length(drum_test)
    class = drum_names(j);
    testThing = drum_test{j};
    lengthH = round(length(testThing));
    envelope = zeros(1, lengthH);

    numPeaks = [];
    peakRatio = [];
    peakFreq = [];
    secFreq = [];
    numfPeaks = [];
    correct = 0;
total = 0;

    i = 1;
    while (i < (lengthH))
        envelope(i+1) = (tiny*testThing(i)+ (1.0-tiny)*envelope(i));
%         envelope(i+1) = (tiny*abs(testThing(i))+ (1.0-tiny)*envelope(i));
        i = i+1;
    end

    i = 1;

    while (i <= (lengthH))
        if (envelope(i) > threshold && counter <= 0)
            start = i - 15;
            last = i + 65;
%             sample = envelope(start:last);
            sample = testThing(start:last);
            [peaks, loc] = findpeaks(sample, 'MinPeakDistance', 10);
```

```
            numPeaks = [numPeaks, length(peaks)];
            [sortedPeaks, sortedPeakI] = sort(peaks, 'descend');
            peek = (sample(end)/sortedPeaks(1));
            peakRatio = [peakRatio, peek];
            L = length(sample);
            NFFT = 256;%2^nextpow2(L); % Next power of 2 from length of myRecording
            Y = fft(sample,NFFT)/L;
            f = Fs/2*linspace(0,1,NFFT/2+1);
            max_fft = max(abs(Y(1:round(NFFT/2+1))));
            [fpeaks, floc] = findpeaks(abs(Y(1:round(NFFT/2+1))));
            [sortedfPeaks, sortedfPeakI] = sort(fpeaks, 'descend');
            peakFreq = [peakFreq, f(floc(sortedfPeakI(1)))];
            secFreq = [secFreq, f(floc(sortedfPeakI(2)))];
            [nfpeaks, nfloc] = findpeaks(abs(Y(1:round(NFFT/2+1))), 'MinPeakHeight',
max_fft/2);
            numfPeaks = [numfPeaks, length(nfpeaks)];

%            if (i > (22600/(1/(Fs/1000))) && i < (23400/(1/(Fs/1000))))
%             figure();
%
%             subplot(2,1,1)
%             plot ((1:length(sample))/Fs, sample);
%             subplot (2, 1, 2);
%             plot(f,2*abs(Y(1:NFFT/2+1)))
%             title(["thing " num2str(i)]);
%              end

            sampleData = [peek, f(floc(sortedfPeakI(1))), f(floc(sortedfPeakI(2)))];
            drumClass = predict(mdl,sampleData);
            if (strcmp(drumClass{1},drum_names{j}))
                correct = correct +1;
            end
            total = total + 1;
            counter = 100;
            envelope(i) = 0;
        end
        if (counter > 0)
            counter = counter - 1;
        end
        i = i +1;

    end
    disp(['For ' drum_names{j} ' got ' num2str(correct) '/' num2str(total) '='
num2str(correct/total)])

end
```