Audio Beat Detection with Application to Robot Drumming

Michael James Engstrom
Portland State University

Audio Beat Detection with Application to Robot Drumming

by

Michael Engstrom

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Marek A. Perkowski, Chair
Fu Li
John M. Acken

Portland State University
2019

Abstract

This Drumming Robot thesis demonstrates the design of a robot which can play drums in rhythm to an external audio source. The audio source can be either a pre-recorded .wav file or a live sample .wav file from a microphone. The dominant beats-per-minute (BPM) of the audio would be extracted and the robot would drum in time to the BPM. A Fourier Analysis-based BPM detection algorithm, developed by Eric Scheirer (*Tempo and beat analysis of acoustical musical signals*)[i] was adopted and implemented. In contrast to other popular algorithms, the main advantage of Scheirer's algorithm is it has no prerequisite to decompose the audio information into notes beforehand and can therefore be automated. In contrast, the McKinney and Breebaart feature set detection and classification method has a result that typifies music genre into static features and is not suitable for real time control of a robot (*Features for Audio and Music Classification*)[ii]. A host computer inputs audio from the environment (via microphone) and extracts the BPM data with the Scheirer algorithm to be sent to a robot controller. A commercially available robot controller was used to control the Drumming Robot servo motors and to interface with the host.

The robot motion control task and the input audio BPM detection task are purposely separated in this implementation. One advantage is that each task could be developed independently. However, the main advantage of this approach is to create a generic interface between Input Logic and Robot Control functions, so each could

be used independently for application to other robots or control systems. Extracted BPM data is useful not for just the Drumming Robot but for any robotic system that interacts in real time with the sound environment, such as dancing robots. By the same token, the Drumming Robot can be controlled by any BPM information source, if the control signals are compatible.

The Robot Theater at Portland State University features animated robots with the goal of performing music and acting out scenes for the entertainment of the audience passing through the halls of the FAB building. The Robot Drummer idea was conceived following the construction of a Handshaking Robot class project involving the ‘DIM’ robot located in the PSU Robot Theater. By adding a second arm to the DIM torso and powering movement by servo motors and a robot controller, the motions of drumming could be performed for the Robot Theater. Audience members could play music, clap or otherwise make rhythmic sounds and a microphone would input the audio to be processed to control the motion of the Drumming Robot.

Table of Contents

List of Tables

List of Figures

List of Equations

Chapter 1 – Introduction

Music is composed of multiple acoustic elements which combine to be interpreted as tempo, melody, beat, etc. The human ear is very adept at psychoacoustic discernment of these elements in the music as a whole. Tempo includes counterpoint, grouping, and hierarchy which are subtly combined and interpreted by the human ear. In electronic decomposition of music or other repetitive audio, it is apparent that tempo is complex while the beat or pulse (BPM) is simple. ‘‘The experience of rhythm involves movement, regularity, grouping, and yet accentuation and differentiation’’ (Handel).[iii] Handel contends that beat in music is the “sense of equally spaced temporal units” and the repeating pattern is a candidate for frequency derived mathematical decomposition such as Fourier Transforms.

Fourier Transforms can detect frequency power information to determine the beat of an audio sample. This decomposed audio beat information can be used to control mechanical output, such as control of the arm movements of the Drumming Robot. Edward Large and John Kolen refer to beat as "one of a series of perceived pulses marking subjectively equal units in the temporal continuum" and go on to say that "beat is a subjective experience" (*Resonance and the Perception of Musical Meter*).[iv] In his paper Eric Scheirer describes a beat detection algorithm which is effective in determining the BPM of an audio sample (*Tempo and beat analysis of acoustical musical signals*).[v] The use of Fourier Transforms is effective for immediate analysis of

BPM information, allowing near-real time calculation. Fourier Transforms are elaborated in the Algorithm section.

Other methods exist for determining the beat (and sometimes tempo) of a musical signal, of varying complexities and effectiveness. Povel and Essens use the concept of an internal clock in the listener and accent distribution matching in the input signal to perceive temporal patterns (*Perception of Temporal Patterns*).[vi] Large and Kolen employ oscillatory resonance calculations to an input signal. Response to phase and period is tracked in a filtered form of phase-lock loop. Valtino et. al. use filter banks to detect beat in ECG signals (*ECG Beat Detection Using Filterbanks*),[vii] and while Scheirer uses filter banks in his algorithm, it is only to enhance the Fourier Transform + Comb Filter method employed. This previous work of Sheirer is elaborated on and continued with in this thesis.

For this Drumming Robot thesis, the Scheirer Beat Detection algorithm is implemented and explored for use in controlling the Drumming Robot. The program variables were parameterized using a range of inputs to evaluate the algorithm. A more detailed description and results of experiments will be presented in Chapter 2. By testing multiple parameter combinations, it was possible to optimize the accuracy and speed of the algorithm resulting in improved performance quality for the audience. By converting the input signal to the Frequency Domain using the Fourier Transform, complex convolution operations are reduced to simple multiplication operations. Input signals are multiplied in the Frequency Domain with known Comb

Filter frequencies to observe the result. This Scheirer Algorithm method is much simpler than the other beat detection methods mentioned above and is employed in this thesis.

Again, for this thesis the logic task of beat detection and the task of robot control were separated. A host Input Logic system which extracts and sends BPM data over a communication port does not need to know the configuration of the robot which implements the drumming motion. A drumming robot listening to a communication port for BPM audio control information does not need to know how the BPM information is obtained. It only cares about the data and is responsible for implementing the resulting BPM-controlled motion. The design and test of such a Logic-Control system is thereby simplified. The host Input Logic system is only required to accurately extract beat information and send the control data to the Robot Controller.

The innovation is in the application of the BPM information once detected by the host system. My contribution is in separating the BPM detection from robot control. With my method, any robot could use the host BPM information for a variety of unknown tasks beyond drumming; for instance light controllers, stage props, or other robots that can implement BPM data. The Robot Control system only needs to be able to input the BPM data and accurately implement the drumming or other desired actions. Any input, if it is in the correct defined serial format as described in the

Robot Control section, can be used with this drumming robot. This includes other BPM detection methods approaching real-time data input.

A drumming robot preferably exhibits human-like motion. As part of my work for this thesis it was observed that articulating lamp sections resemble jointed limbs. This humanoid resemblance to and the motion range of lamp arms led to my utilizing these items. Two jointed lamps were dismantled so that the remaining portion hinged like an elbow and swivel connectors were added to the top of the robot arm to simulate shoulder rotation and swing. The term “jointed robots” can be applied to the assembled robots. See the Robot Design - Arms section for images and details of construction and operation. This construction resulted in three degrees of freedom for each arm, or six degrees of freedom total. The overall cost was very low (a few hundred dollars, see the Bill of Materials section) when compared with commercial robots priced in the thousands of dollars.

The flow diagram shown in Figure 1 describes the steps used in the Scheirer Beat Detection Algorithm. Pseudocode of the operation is followed by a detailed explanation of the algorithm steps.

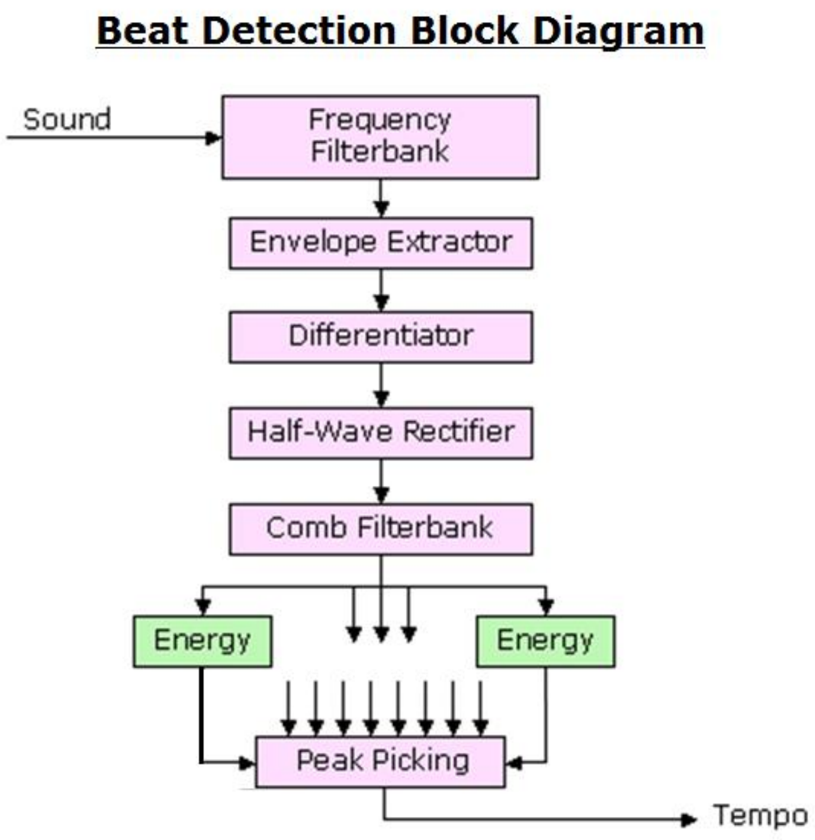


Figure 1 – <u>Beat Detection Diagram.</u> ***Showing the data processing flow of an input signal through the algorithm (Rice)***

Beat Detection Algorithm steps:

- Frequency Filter Bank
    - Split frequency range of sample into smaller segments
- Envelope Extraction (Fourier Transform)
    - Frequency power is extracted for later comb filter comparison
- Differentiation
    - Smoothing of extracted signal
    - This improves the accuracy of the results
- Rectification
    - Isolation of desired frequency information
- Multiple Comb Filters (Resonant Filterbank)
    - Match Step 2 Fourier frequency power to a series of known comb filter frequencies
    - Peak-Picking
    - Best Fit to Comb Filter comparison is our best candidate for a matched BPM output

Beat Detection Audio Input

The host-side processing of audio for BPM detection begins with a choice of inputs: microphone samples or stored *.wav* files. MATLAB offers the benefits of built-in sound device input functions which access the microphone and sound card on the host computer. With MATLAB there is also available built-in matrix manipulation for audio data, Fourier processing functions, and serial connection functions. All these features were implemented in the BPM algorithm. The host is entirely responsible for the algorithm which extracts the BPM from audio input. Then the extracted information is sent over a serial connection to the robot controller. In this thesis only MATLAB is used to perform the host-side audio input, BPM detection and serial output operations.

For live sound input, a signal from a microphone on the host computer is sampled and the corresponding digital data stored as a single channel 8000 Hz 8-bit array in MATLAB. Stored *.wav* files (for example, music or click tracks) are digitized using the center of the file. This is accomplished by dividing the number of samples in the file by 2 to find the center of the song or music file and sampling before and after this center point. This eliminates intro and outro portions of the sample file and focuses on the main section. Resulting data arrays both have stored frequency information that can be varied as a parameter from 2048 to 16384 samples in powers of 2, which is the input format required for Fourier Transforms. Varying this power of 2 parameter affects the accuracy and the processing time of the algorithm. While it is possible to analyze input data that is not a power of 2, there is little energy information beyond the last power of 2 and is therefore inefficient.

There is greater accuracy when using a higher power of 2 and a larger input sample for the Fourier Transform processing. However, the tradeoff for using a larger sample is a resulting increased processing time and output result lag. The implementation for this thesis utilizes 10 seconds of audio sampling data in the Beat Detection algorithm. Any added time due to processing high sample rate input will increase the lag from audio input to Robot Drum output. As discussed in the Algorithm Evaluation and Optimization section, the target processing time is less than 5 seconds with a BPM error of less than 5 percent. This places the overall algorithm response time to under 15 seconds.

Below is pseudo code for implementing the Beat Detection Algorithm in MATLAB:

```
1) Input audio from file or microphone
    a. .wav file or 10 seconds of microphone sample
    b. Sample is digitized and stored for BPM processing
2) Follow BPM Algorithm
    a. Scripts perform the BPM Algorithm Steps
        i. Result is BPM value
3) Send BPM control value using Serial connection to Robot Controller
```

a. Controller receives input

b. Change speed and tempo of robot drumming arms according to inputs

A full printout of MATLAB scripts and Atmel C code is included in Appendix B. The input file to the algorithm is the digital audio matrix.

BPM Algorithm Steps

Step 1: Frequency Filterbank

The input audio sample is split into several frequency ranges, and each range is passed through the BPM algorithm. This is targeted for audio samples such as music, which varies in frequency range according to the variety of instruments used. Different instruments use different frequency bands, and a frequency Filterbank allows for instruments in these varying frequencies to be detected in the BPM algorithm. Most rhythm instruments such as drums or bass use a lower frequency spectrum (0-200 Hz). Some audio samples exhibit only a small frequency range. An example of these audio samples is so-called ‘click track’ signal files. Click tracks are audio files created to have a specific BPM by repeating a pulse signal for the duration of the file.

For this algorithm the Filterbank split is:

0-200Hz, 200-400Hz, 400-800Hz, 800-1600Hz, 1600-3200Hz

Each passband filter is implemented using a sixth-order elliptical filter and cuts off frequencies below and above the desired range. Active filters such as elliptical frequency filters offer sharp defined ranges of filtering (Witte). By allowing only frequencies in a specific range to “pass” this type of filter has become known as a band-pass or passband filter. The passband filter implemented in this thesis results in

3dB of ripple in the passband and 40 dB of rejection in the stopband, with sharply defined ranges for each frequency band. This low-pass characteristic, when combined with a half-Hanning window (see the Windowing algorithm step), results in a -15 dB response with a 6-dB per octave roll off. The filter is implemented with MATLAB functions using a digitized input sample. Most BPM information for audio tracks is in the 0-200Hz band (correlating with rhythm instruments such as drums and bass). Melody, vocal and harmony elements in music tend to be in the higher band frequencies but are also less likely to follow the beat as closely (Scheirer). In Figure 2 and Figure 3 are plots of the bands, separated to show the drop-offs:

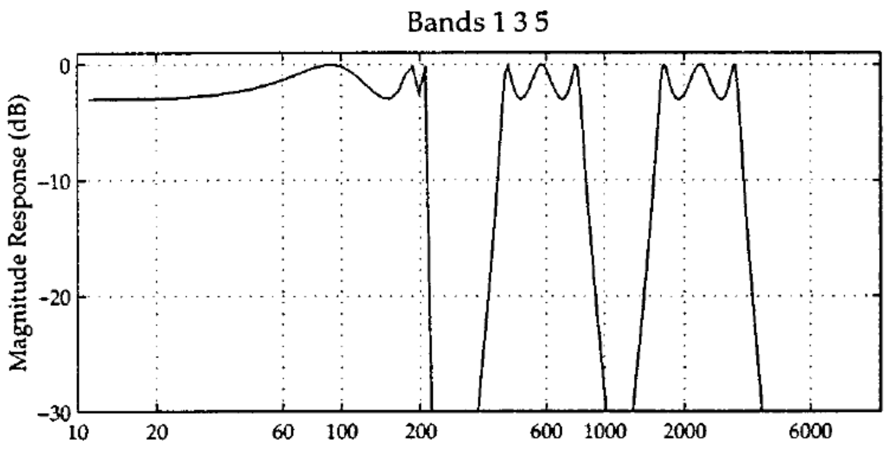


Figure 2 – Filterbank Bands 1, 3, 5 (Scheirer)

Figure 2 shows Bands 1, 3, 5 of the 6 band Filterbank. Bands 2, 4, 6 are graphed separately in Figure 3 allowing each band to be clearly distinguished for Magnitude Response (dB) characteristics.

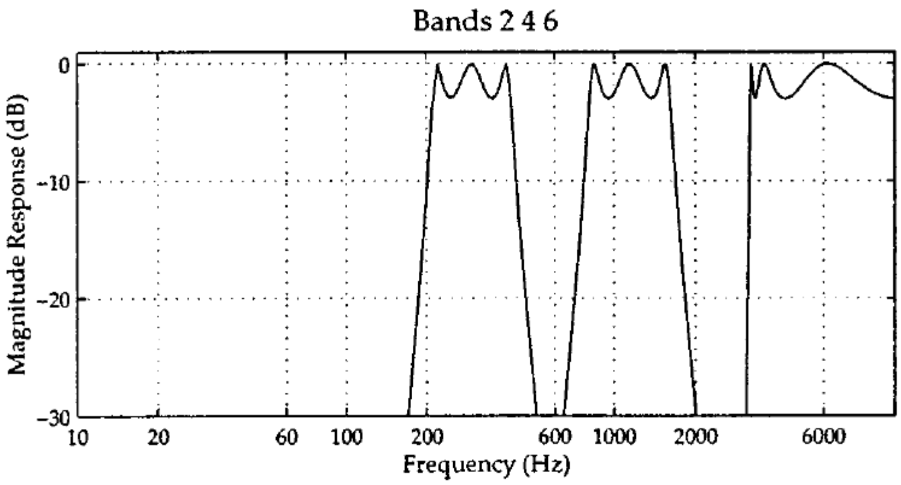


Figure 3 – Filterbank Bands 2, 4, 6 (Scheirer)

Figure 4 shows an input signal in the Time Domain in the top frame. The bottom frame shows the same signal after using the Fourier Transform to convert it to the Frequency Domain, showing the frequency and magnitude response.

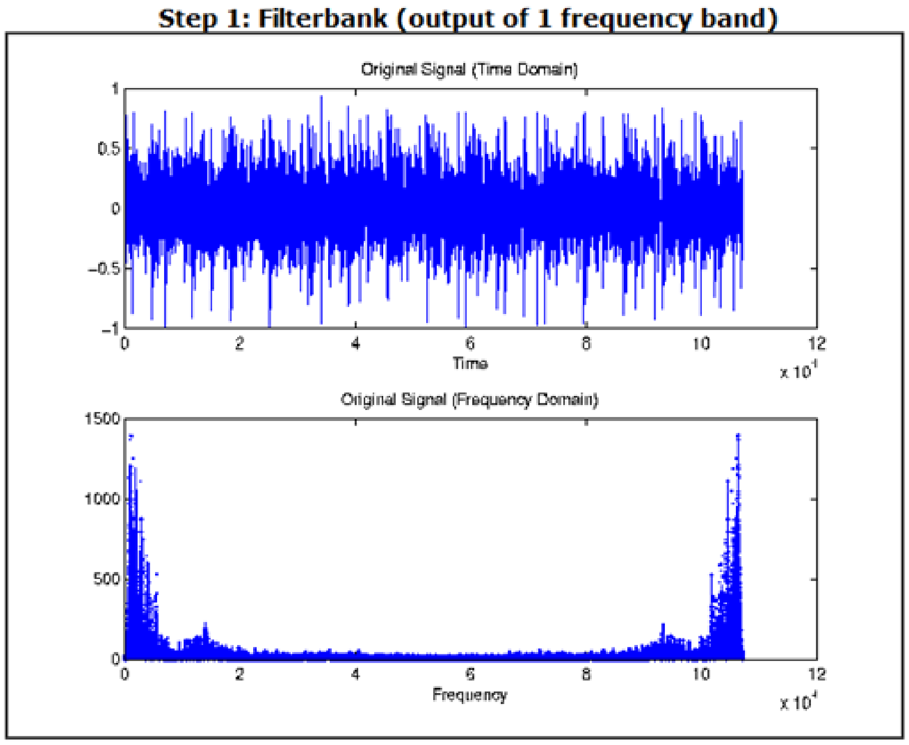


Figure 4 – <u>Step 1: Frequency Filterbank.</u> ***Algorithm step showing the output of a sample after passing through the filter (Rice)***

Step 2: Windowing and Envelope Extractor

After using the filterbank the signal is transformed using a Hanning Window to clean up the frequency range and improve signal clarity. Windowing the input signal in the Time Domain before processing in the Frequency Domain can improve the accuracy of the resulting signal as shown in Equation 1. Time record samples are weighted by

$$w(n) = 0.5\left(1 - \cos\left(2\pi \frac{n}{N}\right)\right), 0 \ \leq n \ \leq N$$

Equation 1 – Hanning Windowing Filter (Witte)

where n = bin number and N = number of bins. Bin refers to integer values corresponding to digitized frequency samples from the input source.

An input signal may have extraneous frequencies that are outside the desired periodic frame if a waveform does not fit precisely in a time period. Leakage in the frequency domain can occur, the transient noise of which can negatively impact the accuracy of Fourier analysis. By specifying a period ‘window’ much of the extra frequency information (acting as noise) can be trimmed off. In Figure 5 we see the Fourier result of a signal which has had windowing applied (Witte). The frequency and power representation is clearly represented with no spectral leakage of other frequencies outside the main frequency spike. Figure 6 shows a signal which was not windowed. The spectral leakage is apparent after applying the Fourier transform, in the form of power and frequency ‘noise’ around the signal.

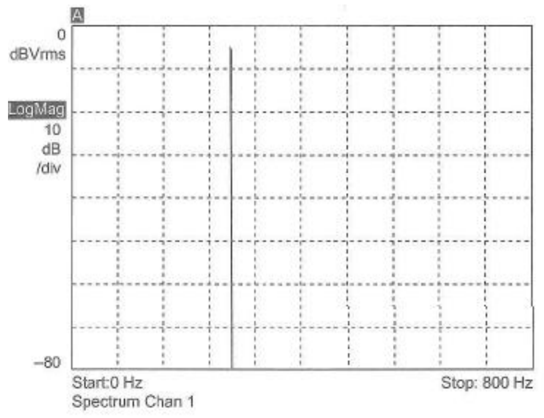


Figure 5 - **Signal with no Spectral Leakage**. ***Fourier analysis shows a discrete Frequency/Power spectrum (Witte). This signal is the desired result of a clean signal after windowing has been applied.***

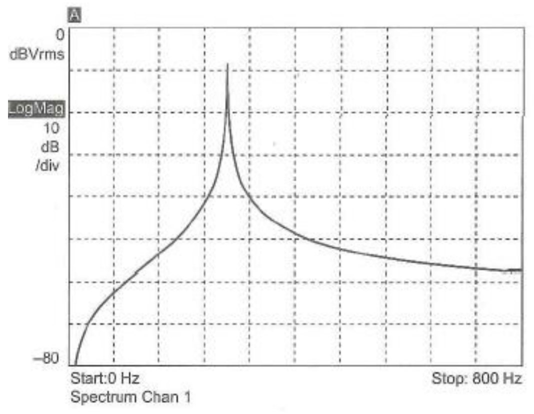


Figure 6 - **Signal with Spectral Leakage.** ***Note the frequency elements around the original signal which act as noise in this Fourier analysis of the original signal (Witte).***

Two of the most popular windowing functions are Hamming and Hanning (Hann) windows. The main difference between these methods is how sharply the resulting signal slope changes when the input signal is multiplied by the windowing function.

Hamming windowing offers a sharper center frequency; Hanning windowing reduces the side lobe amplitude away from the center frequency (see Figure 7).

For BPM detection as presented in this thesis, it is desired to lower the non-center frequency amplitude. The subsequent effect is to improve the result when later multiplying the signal with comb filter signals. Therefore, because it suppresses lower and higher frequencies, the Hanning window was chosen as a better implementation:

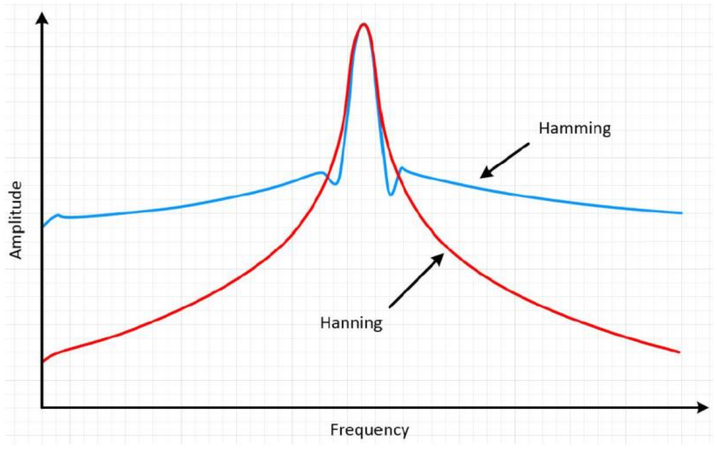
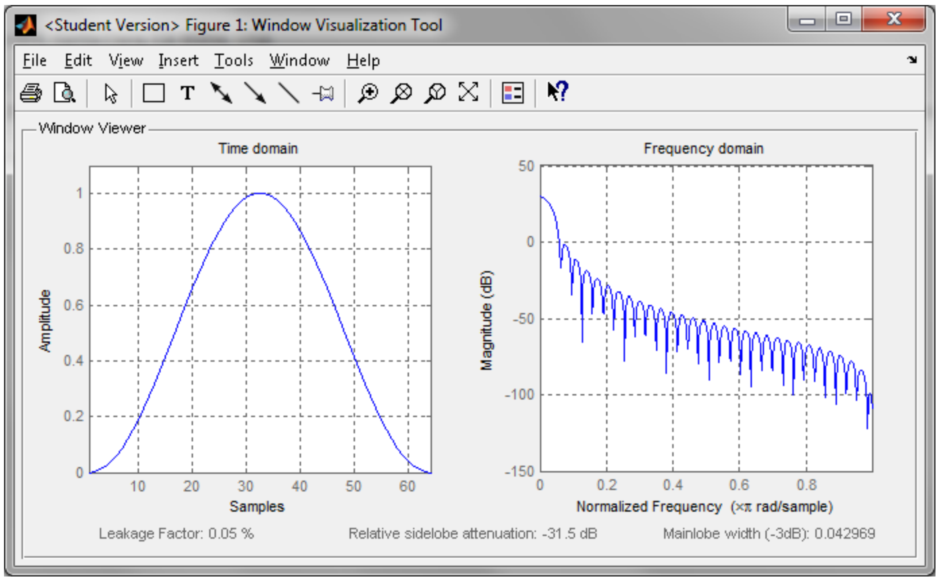


Figure 7 – <u>Hanning and Hamming Windowing Filters (National Instruments)</u>

Windowing limits the inclusion of partial-period waves which can skew the FFT. This is also known as ‘spectral leakage’. With windowing the signal is zero outside a chosen interval. This improves the result in the desired range of the FFT. Using the MATLAB Window Visualization Tool, the effects of windowing on a signal can be observed by amplifying the center frequency and suppressing the lower and higher frequency response. This is shown with an example in Figure 8 of a generated signal using N = 64. On the left is shown a windowed input pulse in the Time domain and its Frequency domain representation on the right.


<Student Version> Figure 1: Window Visualization Tool
File Edit View Insert Tools Window Help
Window Viewer
Time domain
Amplitude
Samples
Frequency domain
Magnitude (dB)
Normalized Frequency (×π rad/sample)

Leakage Factor: 0.05 %
Relative sidelobe attenuation: -31.5 dB
Mainlobe width (-3dB): 0.042969

Figure 8 – <u>Windowing Example in MATLAB</u>

In the next step an Envelope Extractor is used to filter each of the signal segments. The audio sample segments are converted from the Time Domain to the Frequency Domain using a Fast Fourier Transform (FFT) derived formula. Since the samples are already in digital form, a Discrete Fourier Transform (DFT) is performed. The Fourier Transform separates the frequency and magnitude components of the signal. In the Time Domain the signal would be convolved to extract the audio input data, but this is inefficient since the invention of the FFT. Convolving a signal in the Time Domain corresponds to multiplication in the Frequency Domain.

Converting the sample to the Frequency Domain and multiplying the signals is the same operation but much simpler to perform (and faster, which is always a consideration for real-time signal processing calculations). Without first converting the signal from the Time Domain to the Frequency Domain using the Fourier Transform, the signal must be convolved to extract the frequency information. Convolving a signal was once faster than converting the signals to the Frequency Domain using DFT, multiplying them, and converting back using an inverse DFT. However, with the advent of FFT in 1965 convolving was the slower method (Smith). Figure 9 shows an example of an input signal before Envelope Filtering (left) and after (right); notice that the data shape is retained, duplicated signal information is removed, and noise is reduced:

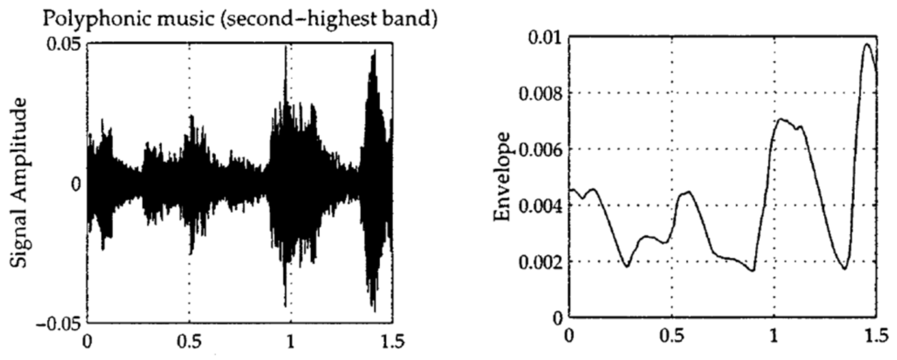


Figure 9 - <u>Signal Amplitude and Envelope (Scheirer)</u>

The Fourier Transform

The Fourier Transform greatly simplifies Signal processing by performing these complex Time Domain operations in the Frequency Domain. Fast Fourier Transform operations are faster than convolving in the Time Domain, even with the DFT conversion operations into and back out of the Frequency Domain. The transformed signal is then converted back to Time Domain using an inverse Fourier Transform. Converting a signal from the Time Domain to the Frequency Domain is performed mathematically with the Fourier Transform Pair where *X(f)* is the Frequency Domain signal and *x(t)* is the Time Domain signal as shown in Equation 2. Note that the algorithm used in this Thesis does not convert the signal back into the Time Domain; once the signal is multiplied with the comb filter and the result captured, the original signal is discarded:

$$X(f) = \int_{-\infty}^{\infty} x(t)\mathrm{e}^{-\mathrm{j}2\pi f t}\,\mathrm{d}t \longleftrightarrow \xcancel{x(t) = \int_{-\infty}^{\infty} X(f)\mathrm{e}^{\mathrm{j}2\pi f t}\,\mathrm{d}f}$$

Equation 2 – <u>Fourier Transform</u>

It is assumed that the input signal *x(t)* is periodic when considered from negative infinity to positive infinity. For digital audio sampling in this thesis our sound sample is not infinite but finite. The sample is already stored as discrete data points, so it is desired to use the Discrete Fourier Transform for digital signals as shown in Equation 3:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\left(\frac{2\pi}{N}\right)nk} \; (k = 0,1,\dots,N-1)$$

Equation 3 – <u>Discrete Fourier Transform</u>

After the original signal is converted to the Frequency Domain by using the Fourier Transform the data is represented in a power-frequency spectrum as a measure of power for the range of frequencies in the 60-120 BPM range. The BPM Algorithm assumes that the beat frequency of a music sample corresponds with FFT frequencies that have the most power. In a later stage of the algorithm, comb filters with known frequencies are used to determine the best BPM candidate.

In Figure 10 is an example of an FFT of an input signal, showing highest frequency power at 150 BPM and a slightly less power peak at 75 BPM.

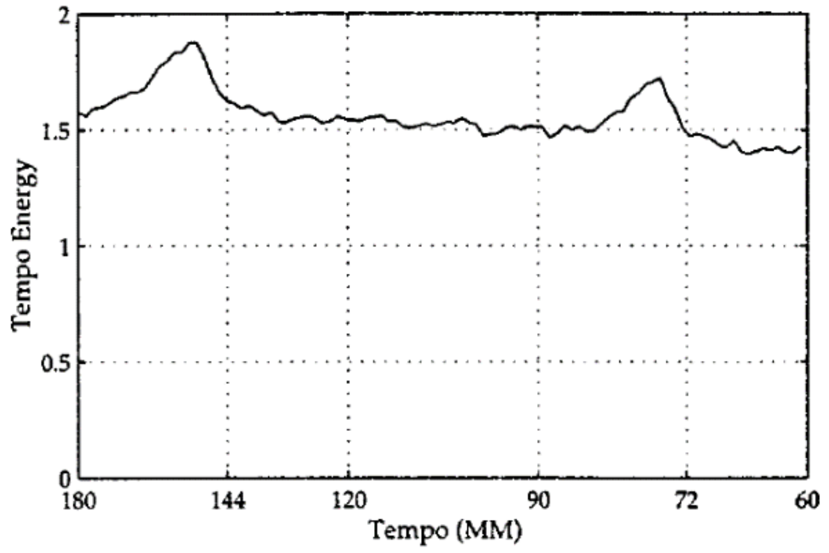


Figure 10 – **<u>Signal Tempo (Frequency) vs. Tempo Energy (Power) - (Scheirer)</u>**

This harmonic effect can be expected at multiples of BPM values for given audio input samples. In this thesis it was decided to limit the BPM range from 60 to 120 BPM because most music samples are in this range. The example in Figure 10 would be considered to be 75 BPM even though it has slightly less power than the harmonic at 150 BPM.

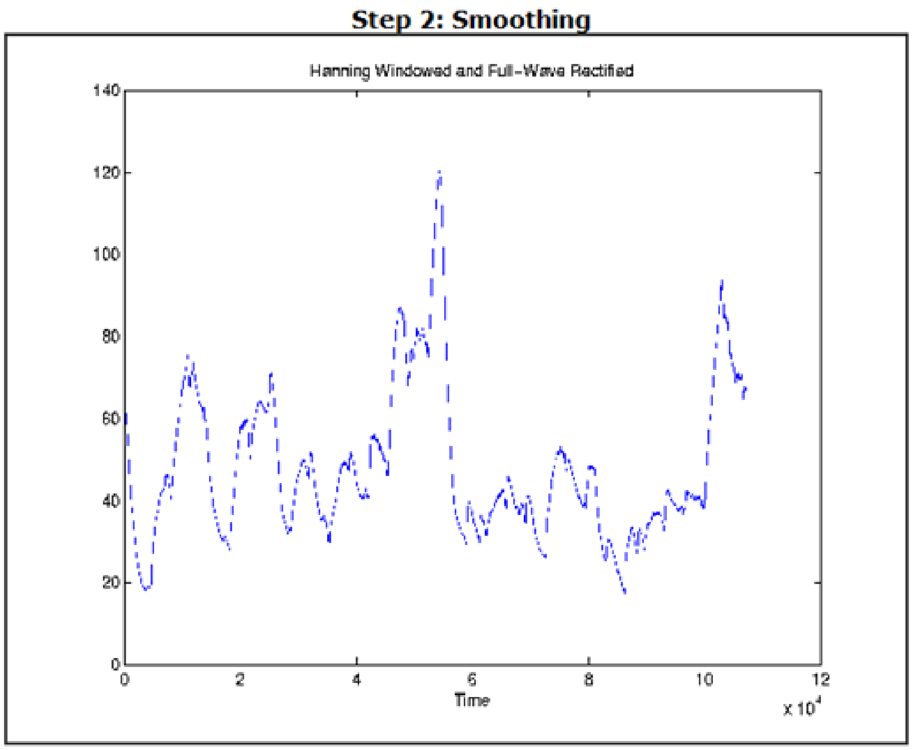


Figure 11 – Step 2: Smoothing (Rice)

Figure 11 shows the input signal after smoothing has been performed using a Hanning Window and Full-Wave Rectification (see Step 4 of algorithm).

Steps 3 and 4– Differentiate and Rectify Signal

We now implement differentiation and rectification to process the signal for improved accuracy of the final BPM determination. The differential of each digital sample to the sample next to it is calculated. The signal is retained only in the case of positive results, giving a half-wave rectified output signal (Figure 12). Differentiating a signal in MATLAB is accomplished in Equation 4 with the diff function, which is a first order finite impulse response (FIR) filter with a response of:

$$H(Z) = 1 - Z^{-1}$$

Equation 4 – FIR Filter Response

The input signal is processed with a half-wave rectify step. This helps accentuate the sound changes in the signal, which corresponds to beats. Rectifying a signal is trivial in MATLAB. For half-wave processing the positive wave portion is kept and the negative wave set to zero. In MATLAB the difference from one sample to the next of the input signal is derived. The result is retained only if the difference is positive, and the signal is now half-wave rectified. Figure 12 below illustrates the input sine wave (red) and the resulting half-wave output (blue):

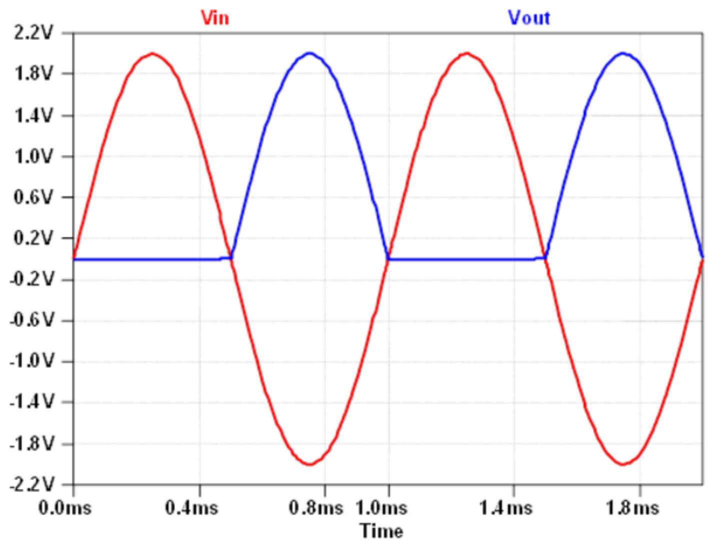


Figure 12 – Half-Wave Rectification (Analog Devices)

Next is a MATLAB example of the input signal which has been differentiated and then half-wave rectified.

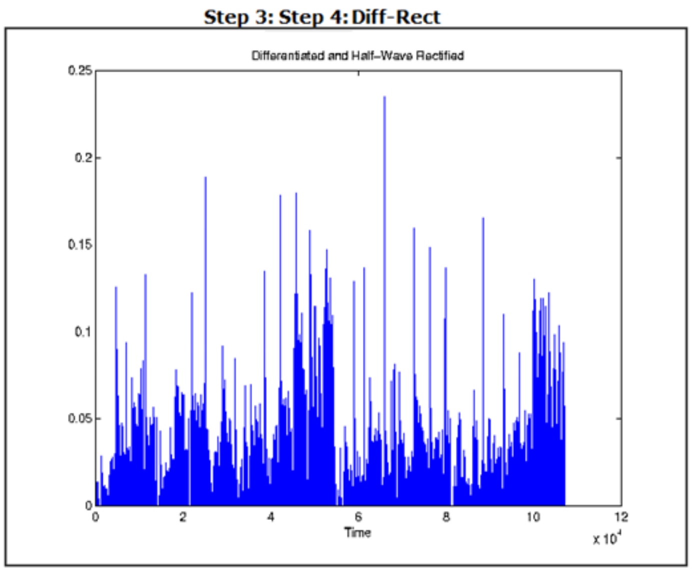


Figure 13 – Step 3: Differentiation and Step 4: Rectification (Rice)

In Figure 13 we can see that the higher power peaks are isolated, allowing for better accuracy when using comb filters in the next step. The comb filter step gives a determination of the best-fit BPM of the input signal (Figure 14).

Step 5 – Comb Filter

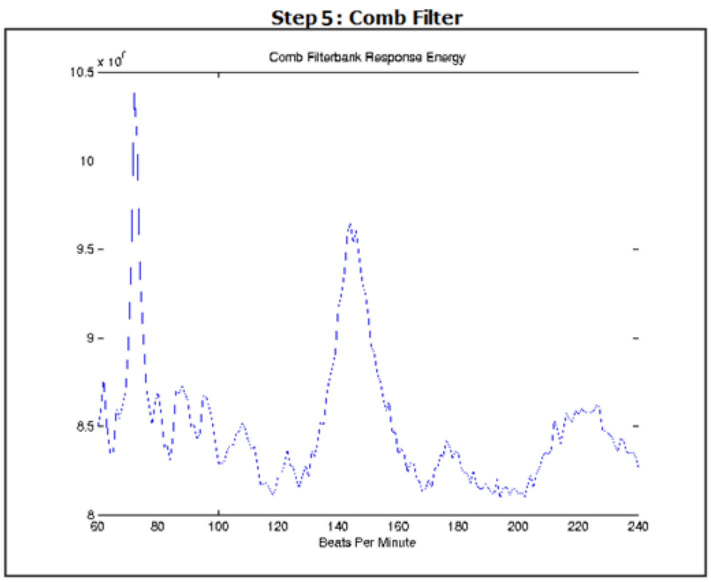


Figure 14 – <u>Step 5: Comb Filter (Rice)</u>

The final algorithm step determines the best estimate of BPM for an input signal. Convolution of the signal in the Time Domain with successive comb filters of increasing, known BPM values results in power products of the signal and comb filters. The best fit BPM is simply the product that has the highest power product. Derivation of convolution is complex in the Time Domain, which is why the signal is converted to the Frequency Domain using the FFT, changing the convolution operation to a simple multiplication operation.

In Step 2 the Beat Detection Algorithm the Fourier Transform of the signal was derived, resulting in a power spike at one or more frequencies, according to the frequency energy. This is multiplied by comb filters of increasing BPM. A Comb Filter (Equation 5) is used to find tempo maxima. For delay *T* and gain α the magnitude response is

$$|H(e^{j\omega})| = \left| \frac{1-\alpha}{1-\alpha e^{-j\omega T}} \right|,$$

Equation 5 – Magnitude Response

Local maxima are wherever $\alpha e^{-j\omega T}$ is near 1 at the *T*th roots of unity, expressed in Equation 6 as

$$e^{-j2\pi n/T}, \quad 0 \leq n < T.$$

Equation 6 – Local Maxima Unity

If we stimulate a comb filter with delay *T* and gain α with a right-sided pulse train of height *A* and period *k* we get reinforcement (resonance) if *T*=*k*. Let x_t and y_t be the input and output signals at time *t* and signal α then Equation 7 is written as

$$y_t = \alpha y_{t-T} + (1\text{-}\alpha)x_t$$

Equation 7 – <u>Output Signal</u>

For our purposes, if a comb filter energy response is higher than a previous ‘best fit’ comb filter (when compared to the input sample) we discard the previous result and keep the new comb filter as our ‘best fit’. This final value is our BPM determination and the Beat Detection Algorithm is complete. Next is a discussion of implementing the algorithm in software.

Chapter 2 – Host Side Software Design

The Scheier BPM Algorithm was implemented on the host using MATLAB scripts. A group from Rice University developed a related project to detect the BPM from input files, and the code for this thesis uses core functions to perform the BPM evaluation (*Beat This, Rice University*). Due to its solid support of matrix manipulation (useful for signal processing) MATLAB was chosen for developing the host-side processing of the Beat Detection Algorithm. MATLAB also offers built in functions for accessing audio input using computer microphones which was a core goal of this thesis for use in the PSU Robot Theater. MATLAB also has functions for establishing serial communication links. The host provides the BPM detection logic and uses MATLAB serial communication to send the robot controller BPM information for control of the drumming arms.

In the thesis planning stages the decision was made to develop the host BPM detection feature separately from the Robot Controller development. This decision was made in part because the host was developed using MATLAB scripts but the Orangutan Robot Controller is developed in C code with the Atmel Studio. The major benefit, however, of separating the host and controller by a serial connection is that each can be used in a modular 'black box' scenario. The Robot Controller is agnostic to the method used to extract the BPM information from an audio source and only listens to the coded control byte information provided by the serial input. Similarly, the host sends the BPM control information over the serial output to the Robot

Controller but the control bytes could be used by any end device which is connected. This allows for the Robot Theater to control the BPM of the Drumming Robot with any BPM detection method or desired control.

The Beat Detection Algorithm steps are implemented in several corresponding MATLAB files, with a main script calling the others. This is all wrapped in a user input script that establishes a serial connection and determines whether the audio source is from a file or the input will be from the system microphone. In the microphone input mode the microphone audio input is processed for BPM information, the control byte sent over the serial connection, and then loops back to repeat these two steps until the user exits the MATLAB script. In this way the Robot Controller is continually receiving the most current BPM information available to the microphone. The byte value of a-z which is sent to the Robot Controller over the serial connection corresponds to the output of the BPM algorithm.

The MATLAB code describes the user interface for calling the Scheirer BPM Algorithm functions and calls the BPM functions in MATLAB with the audio data stored in a matrix. This audio data is passed from function to function in the BPM algorithm until the output result is an integer value from 60-120. The wrapper code then sends a control byte of a-z over the serial connection, to be handled by the Robot Controller (see Chapter 4). Since the BPM range in this thesis is 60-120 inclusive (61 BPM values) and there are 25 control bytes (a-y, z is only used as a PAUSE command) the granularity of BPM accuracy is calculated in Equation 8 as

$$BPM\ Granularity = \frac{BPM\ Value\ Range}{Control\ Bytes} = \frac{61}{25} = 2.44$$

Equation 8 – BPM Granularity for [a, z] State Machine Implementation

Algorithm Evaluation and Optimization

Once the software was working it was important to optimize the BPM function. The performance of the beat detection algorithm varies with the given parameter set. Two goals were determined to be essential for this thesis: BPM accuracy, as determined by percent error deviation from a known BPM; and time, as determined from when an audio sample was entered and the resultant BPM value. This thesis utilizes MATLAB to input the audio, calculate the BPM value, and send the data over a serial connection to the Orangutan robot controller. A set of 'click tracks' were created using Audacity with known BPM values. See Appendices for tools and programs used. The range of 60-120 BPM was included, in 5 BPM granularity, and a few outlier BPMs were added to test robustness. The set is listed in Table 1:

Beat:	35 BPM	55 BPM	60 BPM	65 BPM	70 BPM	75 BPM	80 BPM	85 BPM
90 BPM	95 BPM	100 BPM	105 BPM	110 BPM	115 BPM	120 BPM	125 BPM	145 BPM

Table 1 – BPM Granularity for Parameter Testing. ***Click Tracks were created for each BPM value for use in testing.***

For each set, parameters were varied and the resultant time per BPM and averaged error from the known BPM were measured and graphed. Generic parameters used are listed in Table 2:

	Range
Band Limits	None to [0 200 400 800 1600 3200]
Sample Rate	[2048, 4096, 8192, 16384]
Scaling	[0.75, 1.0, 1.25, 1.5]

Table 2 – Algorithm Parameter Set. ***Variables were modified for combinations of values and the results were graphed for analysis.***

This experiment resulted in 20 different Time vs. Error data points. These were graphed for comparison in Figure 15. The goal for calculation time was to be under 10 seconds, and for error it was less than 10% as in Equation 9:

$$Error = \frac{abs(Expected - Measured)}{Expected} * 100\%$$

Equation 9 – Calculated Error

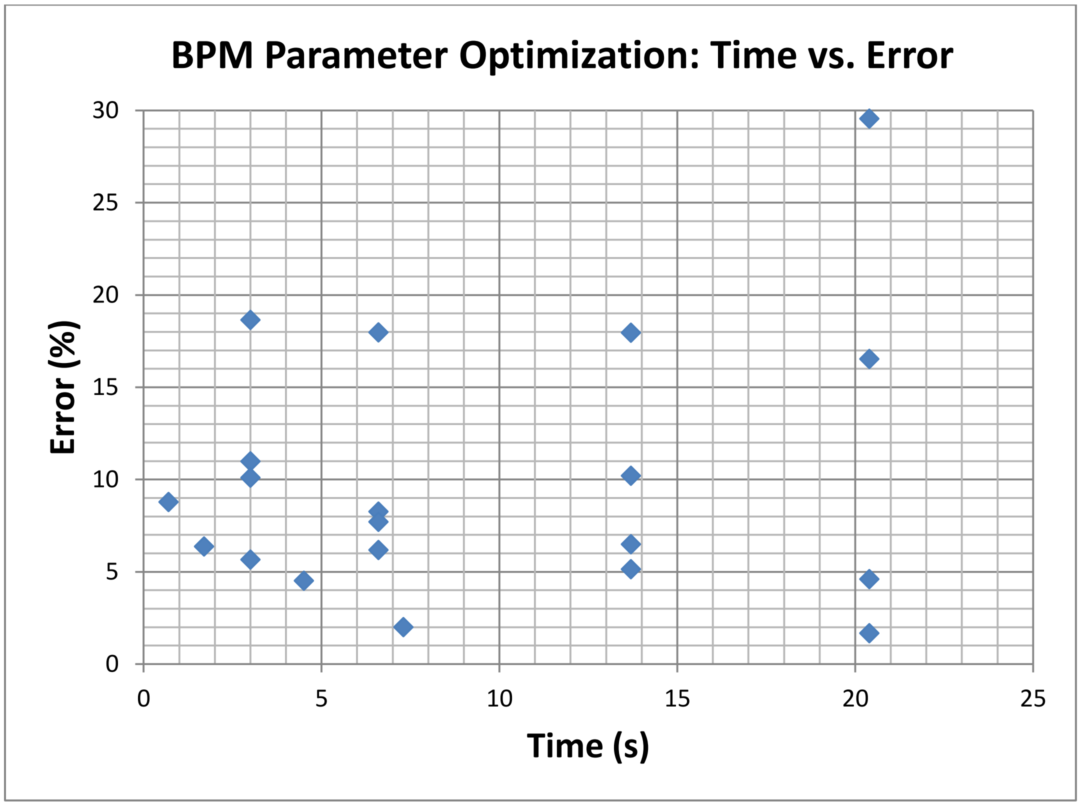


Figure 15 – Parameter Optimization Results. ***Allows for clear interpretation of results and best combination of speed and error.***

With the dual goals of less than 5 seconds processing and less than 5% error the results have been color coded in the tables. Looking at the graph in Figure 15, the data point with the best result and using a filterbank was with 6.6 seconds calculation time and 6.18% average error for the BPM as presented in Table 3. Red results show both goals have been exceeded and green indicates both goals have been met. The red color for both goals shows the result out of range with the default parameters.

(6.6s, 6.18%)	Range
Band Limits	None to [0 200 400 800 1600 3200]
Sample Rate	[2048, 4096, 8192, 16384]
Scaling	[0.75, 1.0, 1.25, 1.5]

Table 3 – <u>Parameter Set: No Scaling.</u> ***This run used filterbanks and a low sampling rate of 4096 but omitted the scaling parameter. Results of (6.6s, 6.18% error) are outside the speed and error targets of this thesis.***

Using the parameter values in Table 4 as the final parameter set, we can be confident that our input algorithm is both fast and accurate. This result highlights 4.5 seconds calculation time and 4.51% average error for the BPM were within our error and time goals. This data point is illustrated in Table 4. This parameter set surprisingly eliminates a major feature of the Handel algorithm, which is the splitting up of the band into smaller band limits. Rather, having a single Band Limit produced more accurate results. This parameter set meets our time and error goals, while using no band limits (per the Handel Algorithm), medium-high sample rate and 1.5 scaling.

(4.5s, 4.51%)	Range
Band Limits	None to [0 200 400 800 1600 3200]
Sample Rate	[2048, 4096, 8192, 16384]
Scaling	[0.75, 1.0, 1.25, 1.5]

Table 4 – Optimized Parameter Set. ***Surprising that omitting a main feature of the Scheirer algorithm, the filterbank, contributed to the best result of calculation time and error.***

The time and error results for the (6.6s, 6.18% error) parameter set in Table 3 are near to the goals of this thesis. However, for the operation of the robot speed and accuracy are desired, and our goal is less than 5 seconds and 5% error. Given this restriction, the parameter configuration used will be the (4.5s, 4.51% error) parameter result from Table 4. Using a wide range of parameter variations and combinations, along with graph decomposition, has enabled a comparator scale for choosing the best performing program tuning. Again, by performing this analysis we have the unexpected conclusion that a key part of Scheirer's algorithm, splitting the input signal into multiple frequency bands, was not present in the best performing parameter configuration. Next we will compare my results with those in Table 5 from the Rice group which did not use parameterization.

Song	**Human-Detected Tempo (BPM)**	**Machine-Detected Tempo (BPM)**	**Harmonics-Normalized Tempo (BPM)**	**Error (%) 100*abs(exp. – meas.)/meas.**
Beverly Hills Cop Theme	119	80.06	80.06	48.64
Lil Jon - Bia Bia	78	59.51	59.51	31.07
Venga Boys - Boom	139	140.16	140.16	0.83
Corelli	91	185.98	**92.99**	2.14
Copland - Fanfare for the Common Man	118	118.13	118.13	0.11
Green Acres Theme	119	62.68	**125.36**	5.07
Stan Getz - Girl from Ipanema	137	136.73	136.73	0.20

Will Smith - Getting Jiggy With It	110	109.5	109.5	0.46
Jurassic Park Theme	110	109.89	109.89	0.10
Green Day - Longview	152	77.67	**155.34**	2.15
Limp Bizkit - Rollin'	185	186.19	186.19	0.64

Table 5 - <u>Rice University 'Beat This' Project Algorithm Results</u>. Right column Error was added to allow comparison with Beat Detection Algorithm performance. Rice used subjects to determine BPM song values which were compared to machine-detected outputs. Many Rice results are comparable to those of this thesis but the first two have sizable detection errors.

A comparison can be made with the results from Table 3, which shows my Beat Detection optimized results for this thesis are within the 5% error and 5 second calculation goal. Parameterized input variables in multiple runs with graphed results clearly highlighted the best combinations (Figure 15). Results from the Rice 'Beat This' project are shown in Table 5. The Rice authors estimated the BPM of various popular songs and used them as inputs to their BPM algorithm. Their results vary from 0.10% to 48.64% error and no mention is made regarding calculation time.

Disregarding the large errors of 31.07% and 48.64% their results are within the range of the optimized results of this thesis from Table 3. This is perhaps not a surprise given that the same code base is derived from Rice for this thesis. Rice was not calculating BPM with the goal of real-time or near-real-time operation but rather for a static output file. The Rice results are therefore missing the constraint of optimizing for speed of calculation time. Also note that the samples used in this thesis were of known BPM from audio click tracks and were not required to be human-detected as the samples were by the authors of the Rice project.

Chapter 3 – Robot Design

Figure 16 – Atmel Studio

Robot Design – Software

The Orangutan Robot Controller is designed to be compatible with Atmel Studio Development Software, a free development program available for download via links from the www.pololu.com website. After installing the program and starting a new Atmel project the desired target device is chosen (Orangutan with the ATMega1284P processor in this case) and a C programming environment is opened. Many sample Atmel software projects are available for controlling the features of the Orangutan robot controller, as well as the rich API features available in the project libraries. For this thesis the Servo, LED, LCD and Serial sample Atmel projects were extremely useful as code references.

For the robot controller facet of this thesis, a looping program initialized the servos, serial interface, and LCD display, then set the arms to drum in 60 BPM. Button inputs allowed for increase or decrease of BPM. The Orangutan continually monitored the serial bus for byte inputs of [0, 9] corresponding to BPM granularity of 6 BPM within 60-120 BPM with 10 states. The final design used bytes [a, z] resulting in an improved 2-3 BPM granularity. This software state machine controlled the output of the BPM in the code running on the robot controller for drumming. Buttons can only increment or decrement states sequentially from 60 to 120 BPM by levels of granularity. A serial input immediately changes the Drumming Robot to the desired BPM mode. Figure 17 demonstrates a state machine diagram of the initial design showing the button and serial inputs, as well as the BPM delay and LCD output.

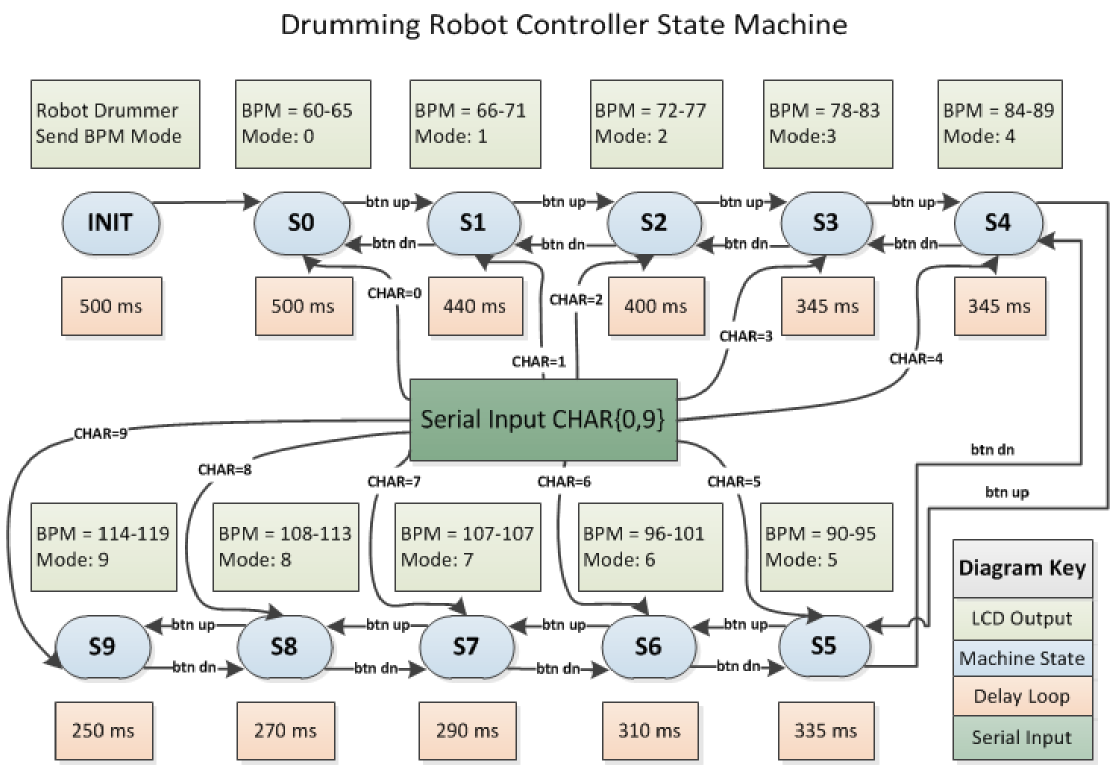


Figure 17 – Robot Controller State Machine. ***In this initial design, ten states allow for granularity of 6 BPM.***

Note that this state machine in the robot controller code has been updated to use a-z inputs (z is used for a 'pause' feature) instead of the 0-9 byte inputs. This allows for better granularity of BPM accuracy with 25 divisions between 60-120 BPM rather than the original 10 divisions. The previous implementation could result in BPM inaccuracy in implementing the serial byte input by as much as 6 BPM due to my designed granularity limitation. With the improved a-z implementation the maximum inaccuracy was reduced to 2-3 BPM due to the finer divisions between BPM state machine levels. An updated flow chart highlighting the design change is illustrated in Figure 18.

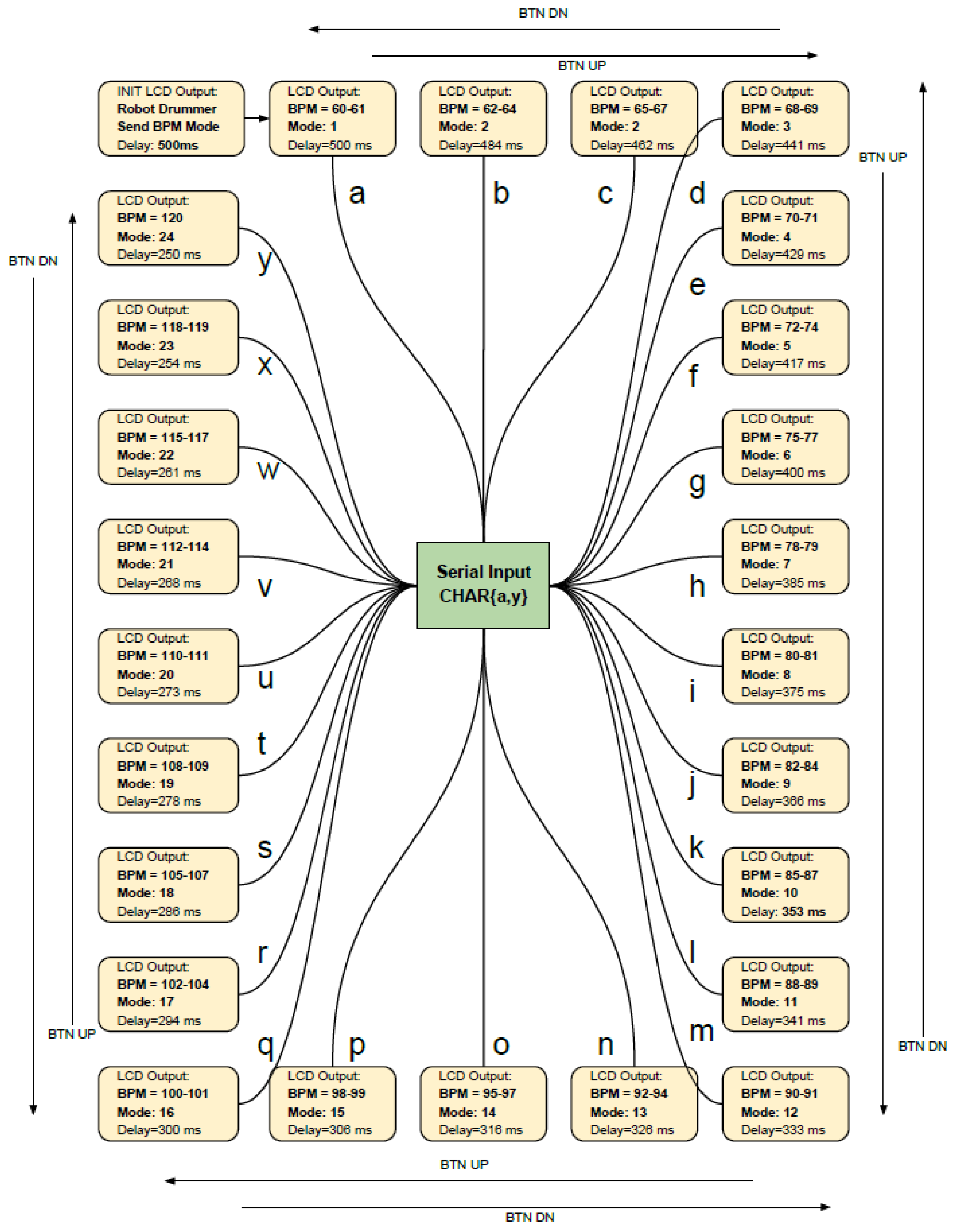

BTN DN
BTN UP
INIT LCD Output: Robot Drummer Send BPM Mode Delay: 500ms
LCD Output: BPM = 60-61 Mode: 1 Delay=500 ms
LCD Output: BPM = 62-64 Mode: 2 Delay=484 ms
LCD Output: BPM = 65-67 Mode: 2 Delay=462 ms
LCD Output: BPM = 68-69 Mode: 3 Delay=441 ms
LCD Output: BPM = 70-71 Mode: 4 Delay=429 ms
LCD Output: BPM = 72-74 Mode: 5 Delay=417 ms
LCD Output: BPM = 75-77 Mode: 6 Delay=400 ms
LCD Output: BPM = 78-79 Mode: 7 Delay=385 ms
LCD Output: BPM = 80-81 Mode: 8 Delay=375 ms
LCD Output: BPM = 82-84 Mode: 9 Delay=366 ms
LCD Output: BPM = 85-87 Mode: 10 Delay: 353 ms
LCD Output: BPM = 88-89 Mode: 11 Delay=341 ms
LCD Output: BPM = 90-91 Mode: 12 Delay=333 ms
LCD Output: BPM = 92-94 Mode: 13 Delay=326 ms
LCD Output: BPM = 95-97 Mode: 14 Delay=316 ms
LCD Output: BPM = 98-99 Mode: 15 Delay=306 ms
LCD Output: BPM = 100-101 Mode: 16 Delay=300 ms
LCD Output: BPM = 102-104 Mode: 17 Delay=294 ms
LCD Output: BPM = 105-107 Mode: 18 Delay=286 ms
LCD Output: BPM = 108-109 Mode: 19 Delay=278 ms
LCD Output: BPM = 110-111 Mode: 20 Delay=273 ms
LCD Output: BPM = 112-114 Mode: 21 Delay=268 ms
LCD Output: BPM = 115-117 Mode: 22 Delay=261 ms
LCD Output: BPM = 118-119 Mode: 23 Delay=254 ms
LCD Output: BPM = 120 Mode: 24 Delay=250 ms
Serial Input CHAR{a,y}
a b c d e f g h i j k l m n o p q r s t u v w x y
BTN UP
BTN DN
BTN UP
BTN DN
BTN UP
BTN DN

Figure 18 - **<u>Improved BPM granularity</u>**. ***The previous implementation used bytes [0, 9] for ten possible outputs of BPM. Using [a, z] allows for twenty-five possible BPM outputs with the 'z' byte (not shown) used for pause/resume.***

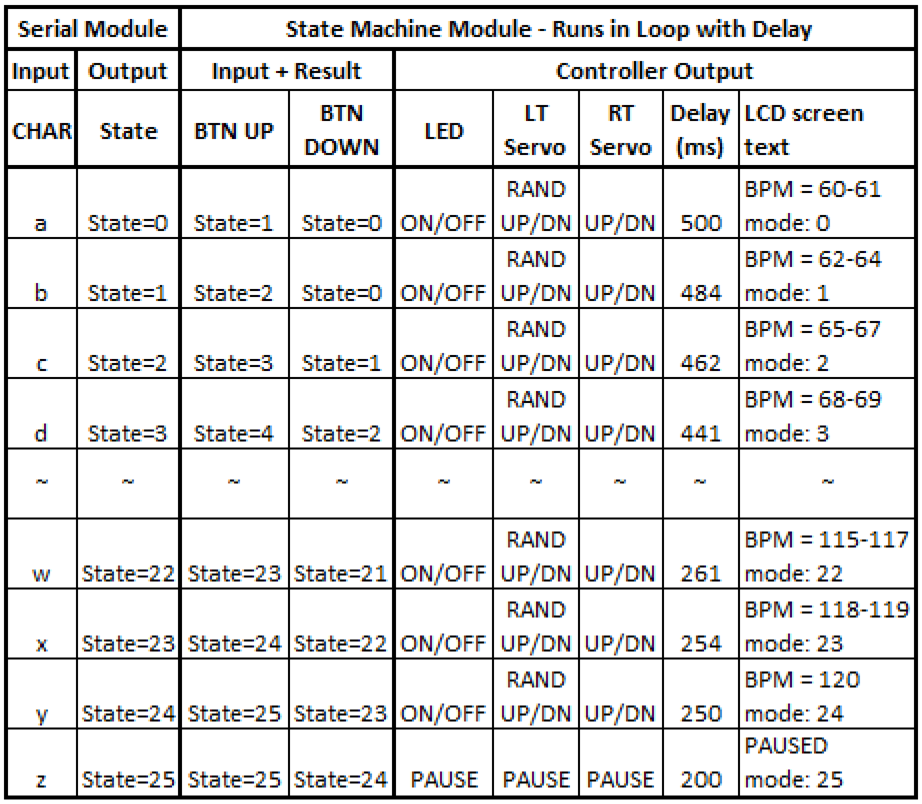
The robot controller program is designed to capture user BPM serial input for setting the state machine to the target BPM. This design is implemented in a framework similar to many other microcontrollers which target real-time operation. This runs in a loop as described in pseudocode below and will be fully explained later.

Orangutan Robot Controller Pseudocode

```
1) Check serial input
    a. If serial byte input of ‘a’-‘z’ detected
        i. Set state of state machine to BPM value according to serial
           input
        ii. Use LCD to notify user of serial character detected
2) Check button input
    a. If Button1, increment state
    b. Else if Button3, decrement state
        i. Set state
3) Perform delay for current BPM state
4) Output LCD and LED information regarding BPM and mode
```

The serial communication from the host-side BPM detection computer and the robot controller is bidirectional. Pressing the middle button on the Orangutan sends a byte string message of “Robots Rule” back to the host. A simple feedback operation of sending a copy of each received control byte allows the host to verify that the Orangutan has the correct byte. Noise on the serial line could lead to incorrectly received byte values and thereby incorrect BPM states. To prevent this occurrence a code is sent from the host before each control byte. This code is three colons sent sequentially then the control byte immediately after. The robot controller code identifies and counts the colons as received and only changes the BPM mode after successfully receiving the triple-colon code.

The robot controller code instantiates a state machine to save the BPM mode during each loop of the program. Byte characters are input over the serial connection and the BPM state is changed to the appropriate value. Also input to the BPM state machine is the input information from the physical buttons on the robot controller. While the serial bytes jump to the appropriate state based on the a-z values, physical buttons (UP, DOWN) move the BPM state incrementally up or down with a floor of 60 BPM and a ceiling of 120 BPM. Outputs of the BPM state machine are servo positions for the arms (up, down), LCD output to the display on the robot controller, and the delay value for the program loop to control the BPM cadence of the servo arms. See Figure 19 for details.



Serial Module		State Machine Module - Runs in Loop with Delay						
Input	Output	Input + Result		Controller Output				
CHAR	State	BTN UP	BTN DOWN	LED	LT Servo	RT Servo	Delay (ms)	LCD screen text
a	State=0	State=1	State=0	ON/OFF	RAND UP/DN	UP/DN	500	BPM = 60-61 mode: 0
b	State=1	State=2	State=0	ON/OFF	RAND UP/DN	UP/DN	484	BPM = 62-64 mode: 1
c	State=2	State=3	State=1	ON/OFF	RAND UP/DN	UP/DN	462	BPM = 65-67 mode: 2
d	State=3	State=4	State=2	ON/OFF	RAND UP/DN	UP/DN	441	BPM = 68-69 mode: 3
~	~	~	~	~	~	~	~	~
w	State=22	State=23	State=21	ON/OFF	RAND UP/DN	UP/DN	261	BPM = 115-117 mode: 22
x	State=23	State=24	State=22	ON/OFF	RAND UP/DN	UP/DN	254	BPM = 118-119 mode: 23
y	State=24	State=25	State=23	ON/OFF	RAND UP/DN	UP/DN	250	BPM = 120 mode: 24
z	State=25	State=25	State=24	PAUSE	PAUSE	PAUSE	200	PAUSED mode: 25

Figure 19 - BPM State Machine Inputs and Outputs. ***The left columns show the input byte and corresponding result for the controller state machine. The right columns show the manual button inputs as well as the outputs to the robot controller for each state.***

It is also possible to send other information such as servo position, servo speed, loop delay and other desired values using the serial connection. This is not currently implemented. The serial connection is used only for control bytes and to program the robot controller. Table 6 shows the input bytes, Robot Controller states and the necessary delay needed per loop for the desired BPM cadence.

Byte Input	**S.M. Mode**	**Effective BPM**	**Loop Delay(ms)**
a	1	60	500
b	2	62	484
c	3	65	462
d	4	68	441
e	5	70	429
f	6	72	417
g	7	75	400
h	8	78	385
i	9	80	375
j	10	82	366
k	11	85	353
l	12	88	341
m	13	90	333
n	14	92	326
o	15	95	316
p	16	98	306
q	17	100	300
r	18	102	294
s	19	105	286
t	20	108	278
u	21	110	273

v	22	112	268
w	23	115	261
x	24	118	254
y	25	120	250

Table 6 – Loop Delay Calculation. ***Each robot controller loop has a delay resulting in the desired BPM output.***

Robot Design – Servo Control

In examining jointed robots it was observed that many of these robots used servo motors (servos) directly as the joints. However, servos can be damaged by excessive torque and need to be programmed to limit motion which does not mimic human motion. One of the advantages of using lamp arms is the range of motion is very human-like, and the joint motion functions whether servos are working or not. In this thesis, servo motors were attached externally to the arms and linkages and springs were used to provide the powered range of motion. This mimics human arms with 'muscles' (servos) and 'tendons' or 'ligaments' (springs or brackets).

Servos are an inexpensive method of implementing motion for robots. For this reason, control boards were researched for features that would allow for effective servo control. Several types of control boards with Hardware Description Language (HDL) programming requirements were researched, including VHDL and Verilog. Both HDL languages are useful for simulating low-level circuits and interfacing with controller boards. It is also possible to instantiate 8-bit and 32-bit microcontrollers to perform advanced programming. Assembly language programming is available for 8-bit microcontrollers and higher level languages such as C can be used with the instantiated 32-bit microcontrollers.

Some advantages and disadvantages are present with HDL programming, however. Hardware control is more direct with HDL, and in fact it is required with most of the controller boards to set up a configuration file to assign all the pins, LEDs, memory bus lines, etc. before the controller boards can be operated. Most controller boards come with examples describing how to use the features of the board but do not usually have the exact fit for the desired project. Some experimentation is required, and there are usually many low-level system elements required to be modified. Some controller boards can be programmed directly in high-level languages such as C++, allowing the designer to take advantage of function libraries to quickly perform advanced projects. With this in mind, the Orangutan Robot Controller Board from Pololu was chosen for this thesis. Hardware features of the Orangutan can be programmed directly, or with the built-in libraries provided, or both.

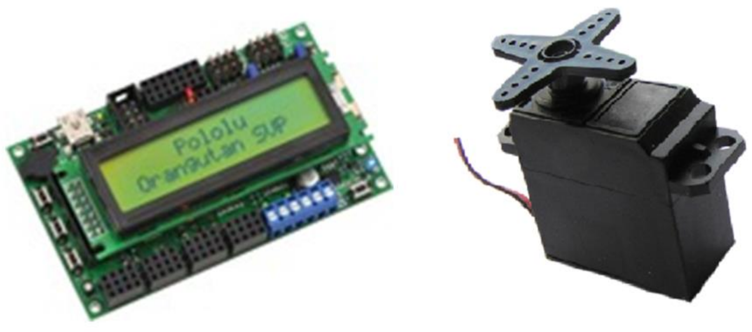
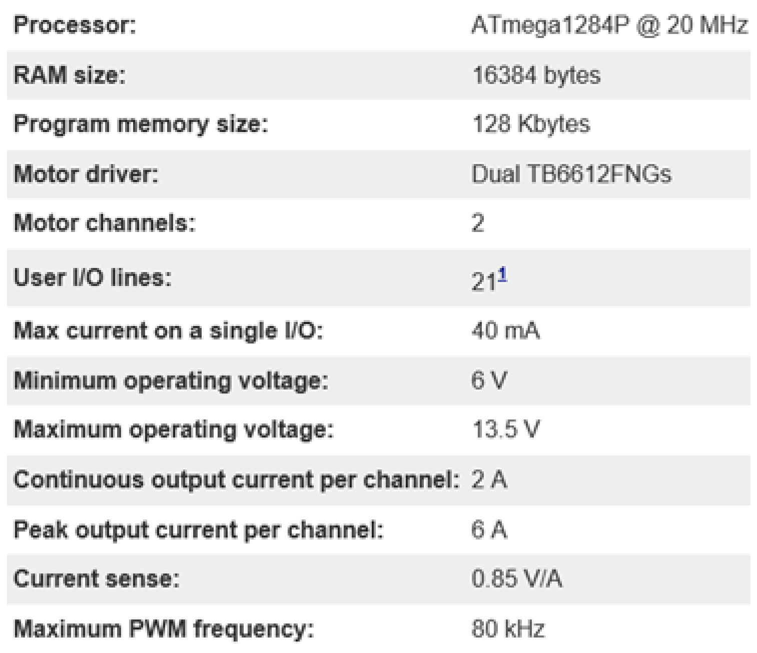


Figure 20 – **<u>Robot Controller and Servo Motor</u>**

Orangutan boards are cheap and can be purchased at www.pololu.com for about $100. The website also has downloads available with many examples for the controller boards. Pololu provides examples of implemented projects for features available on the Orangutan family of boards. These include the Buzzer (tone generation), Digital control via I/O pins, LCD display, LEDs, Motors, Pushbuttons, Serial input and output, SPI (Serial Peripheral Controller) communication, and Servo control. In addition to this, the Atmel programming interface includes libraries of functions that do not require direct control of the Orangutan hardware. Atmel library functions simplify the implementation of controlling the board features by providing the low-level signaling and allowing the use of variables. Someone with no previous knowledge of robot controllers (but with some C++ programming experience) can quickly implement, compile and flash example designs to the Orangutan board and experiment with modifying the behavior. See Table 7 for Orangutan controller board features.



Processor:	ATmega1284P @ 20 MHz
RAM size:	16384 bytes
Program memory size:	128 Kbytes
Motor driver:	Dual TB6612FNGs
Motor channels:	2
User I/O lines:	21[1]
Max current on a single I/O:	40 mA
Minimum operating voltage:	6 V
Maximum operating voltage:	13.5 V
Continuous output current per channel:	2 A
Peak output current per channel:	6 A
Current sense:	0.85 V/A
Maximum PWM frequency:	80 kHz

Table 7 – Orangutan Robot Controller Specifications

The Orangutan is relatively cheap yet it can control 8 servos using C++ API interface calls, as well as 8 more using general-purpose IO ports and lower level programming.

Since the Orangutan is designed with robots in mind it also includes powerful motors suitable for powering wheels, multiple programmable I/O pins for sensors and external control, a USB Serial communication port (doubling as a power source for low-drain usage), LCD display panel for onboard communication to the user, and buttons to interact with, as well as a tone-generator and indicator LEDs. For this thesis the Serial port, LCD display, buttons, LEDs and most especially the servo controller were essential for implementing the BPM beat information extracted from the Scheirer Algorithm on the host.

Servos are fairly simple to use, just give them 3.3V to 6V and a control signal and the arm moves to a position. Most have a range of movement of 180°, with the control signal square-wave pulse running at 50 Hz intervals and 1-2ms ‘high’ time (Figure 21). Changing the pulse signal changes the arm position. A pulse width of 1ms corresponds to one extreme end of the servo motion and a pulse width of 2ms to the other extreme. Pulse widths of 1.5ms put the arm about in the middle. Varying the interval between square pulses changes the speed at which the servo arm reaches the set position. Each servo should be calibrated before use to determine the positions. The APIs available in the Atmel-Programmed Orangutan controller easily control the position and speed of servo movement.

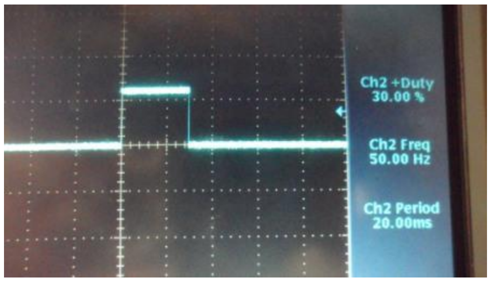


Figure 21 – <u>Servo Pulse Waveform</u>. ***Square wave width determines the servo position and the interval between pulses determines the speed at which the servo moves to the position. The shorter the interval, the faster the movement*.**

The Orangutan SVP 1284 board has eight onboard hardware servo controllers, two motor controllers, three serial interfaces (one USB and two UART) and 3 button inputs. Outputs include LEDs and an attached backlit 2x14 character LCD, as well as multiple programmable IOs. The Orangutan can be powered and programmed via USB, but for servo use a battery pack power supply was necessary. The use of servos caused current spikes which reflected back to the robot controller and interfered with the logic operation of the board. Orangutan robot controllers with battery packs can have issues as the batteries drain. The Orangutan battery pack was eventually replaced with an AC power supply which provided enough power to run the board as well as the attached servos. The AC power also allowed for a consistent current level for the robot controller.

An issue with servos is current reflection spikes in response to a control signal. When the signal is sent and the servo motor responds with movement, it also generates a reflected current to the control board. This can interfere with the operation of the control board in the form of power loss, restarts and even corruption of the programmed flash image. The Orangutan can run low-power operations such as the LCD display, LEDs and beeping noises with just the USB attached for power (although the cable can get alarmingly hot). However, servos require more power to operate and therefore have the current reflection issue as mentioned. Auxiliary power via battery pack (for mobile use) or power supply (stationary use) worked well for the six servos used in this thesis.

Robot Design – Arms

The Drumming Robot Arms needed an attachment point for operation, and the DIM robot (as has been seen in the Robot Theater window) was chosen since it had no arms and was in proportion to a human in stature. Part of the goal for the thesis was to simulate human movement and form wherever possible. Therefore, in addition to lamp arms for the drumming arms, they were attached to the DIM torso so as to mimic human shoulders using caster wheels (minus the actual wheels). Hobby plates and bolts were attached with nested servos to provide the torque for 1 DOF (Degree of Freedom) for the lower arms/elbows, and 2 DOF for the shoulder movement. The total DOF was 6 for both arms combined.

Another of the goals of the robot arm design was to mimic human drumming motion. The lamp arm was a good choice since it was already designed to be limited to a 180° range and resembled the range of the human elbow (Figure 22). There was a functional advantage in avoiding servos for arm joint attachment. If a servo failed, it could easily be replaced without disassembling the joint. Hobby straps were used to extend the swing of the servo motion and thus reduce the amount of torque applied directly to the arm servo. Even with high-torque metal gear servos (as used in this thesis) the load weight of the lower arm was high. Springs, reused from the original lamp arm, were used to counter this arm weight. Also, hinged brackets were used to move the point of contact for the elbow servo farther from the joint. This reduced the torque and force on the servo motor.

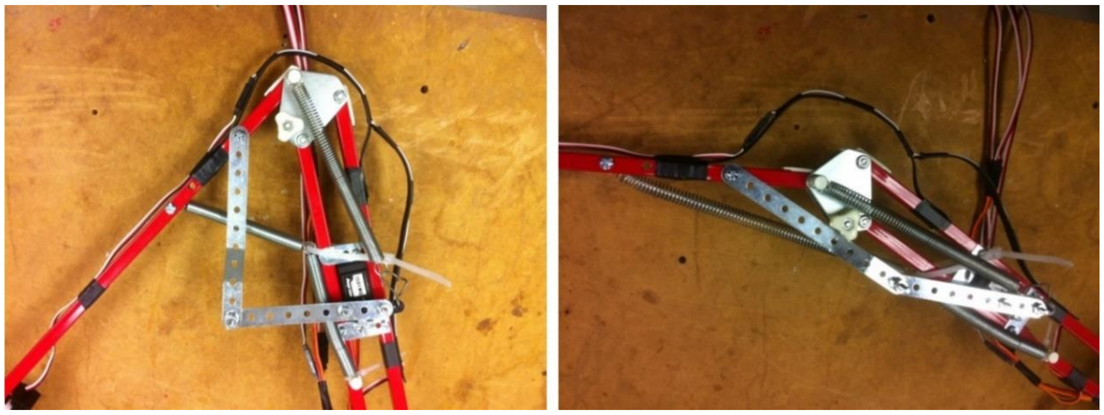


Figure 22 – <u>Robot Elbow Range of Motion</u>. ***A hinged bracket is attached to the elbow servo arm and reduces the torque and power needed to flex and extend the robot drumming arm.***

The design of the shoulders was more interesting (Figures 23 and 24). The initial design included a simple hinge to attach the arms with 1 DOF. While searching for parts to construct the drumming robot it was noted that a caster wheel is a 2 DOF object. Using a sufficiently large caster wheel frame it was possible to fit a pair of servo motors into the frame with the wheel removed (actual wheel not needed). The axle holes were drilled out to fit the arm post for left-right arm motion. As seen in the picture a combination of hobby brackets, bolts, and a servo accomplishes this motion. Using another hobby bracket, heavy wire, and a servo enables up-down shoulder rotation to lift the arm up and down. Figure 25 shows the completed arm and attachment to the DIM robot.

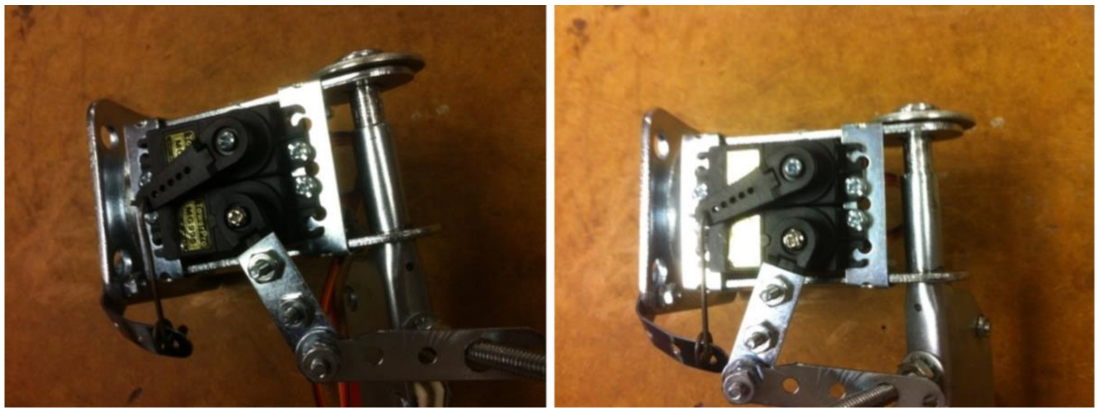


Figure 23 – <u>Robot Shoulder Left/Right Motion</u>. *The servo arm moves a jointed bracket forward and back over a bolt attached to the shoulder. This translates to left/right shoulder motion*.

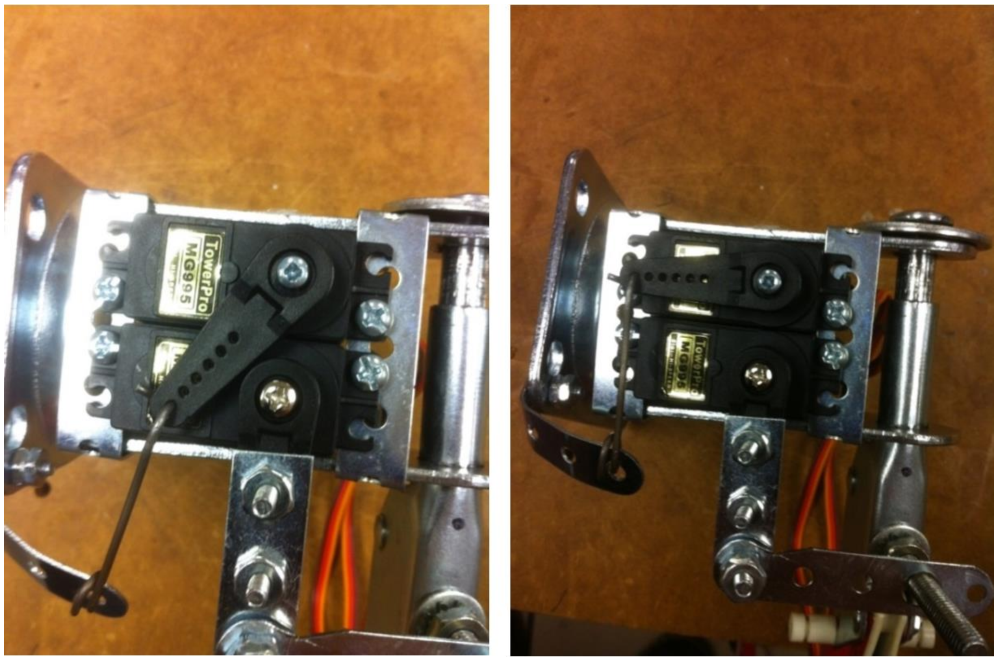


Figure 24 – <u>Robot Shoulder Up/Down Motion</u>. ***The servo arm moves a wire up and down. This wire is connected to the base of the shoulder where attached to the DIM body and moves the entire shoulder assembly up and down*.**

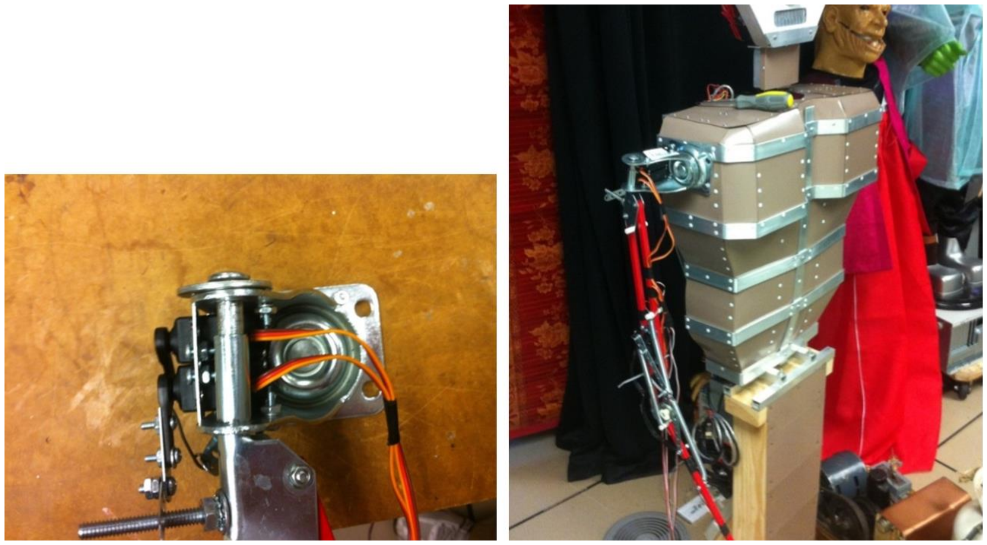


Figure 25 – <u>Robot Arm Mounted on Robot Torso</u>. ***A matched arm with reversed construction is later attached to the opposite side of the DIM robot torso for a two-armed drumming function*.**

Chapter 4 – Testing and Implementation

The robot arms were attached to the torso and the servos connected to the Orangutan robot controller for initial testing. This was initially performed using the controller-side software and buttons. The host and controller were designed separately and could be tested separately. The plan for testing the controller was first to use the on-board buttons to control the BPM states, second to send control bytes over the serial connection using a tty terminal (such as PuTTY), then third to send control bytes using the host BPM software.

The servos were required to be calibrated. The drumsticks attached to the arms were not striking the drum head in a precise position. This resulted in beat skipping when the BPM was in high range and servo motor strain when in low range. These servo position values were changed in the robot controller software until the up and down distance was correct. This corresponds to changing the interval between servo control pulses, as described in the Orangutan Servo Control section previously.

After increasing the BPM values it was also observed that at higher BPMs the arms were no longer striking the drum head. The drumsticks did not have enough time at higher BPMs to strike the head before the loop ended and the servos began to move to the up position. This was corrected by increasing the servo speed value so that at higher BPM loops the servos moved faster to their up or down position. As described in the 'Robot Design-Servo Control' section this is achieved by increasing the width of the servo control pulses to change the position of the servo. The robot arms were now accurately striking the drums with the drumsticks for the entire target 60-120 BPM range.

The calculated loop delay values were tested (counted over the space of a minute) for BPM accuracy and found to be correct. However, since each arm moved in the loop, the perceived BPM was twice the desired value. Also, the sound of the drumsticks quickly became monotonous after hours of testing. Both of these issues were addressed by making a single change to the robot control software: the left arm randomly chooses up or down servo arm positions for each loop, while the right arm continues to steadily alternate between up and down. This allows the right arm to always strike the correct BPM, while the left arm gives a random accompaniment to the performance of the Drumming Robot. The resulting rhythm is varied and changing, yet stays within the target BPM. This varied rhythm adds an enjoyable random beat experience for the audience.

Next, host control was added to the test scenario. A serial connection was established with the drumming robot and known control byte values were tested through the BPM state machine states. The response time was under 0.5 seconds from keyboard strike to state change. Next, the host BPM algorithm software was successfully used to input sound from a microphone and send the BPM control byte over the serial connection to the robot controller. Finally the host software was modified to run in a loop so that it is continually capturing audio, extracting the BPM using the Scheirer Algorithm, and sending the control byte to the robot controller. The Robot Drummer was complete! See Figure 26 for a proud Professor posing with the DIM robot.

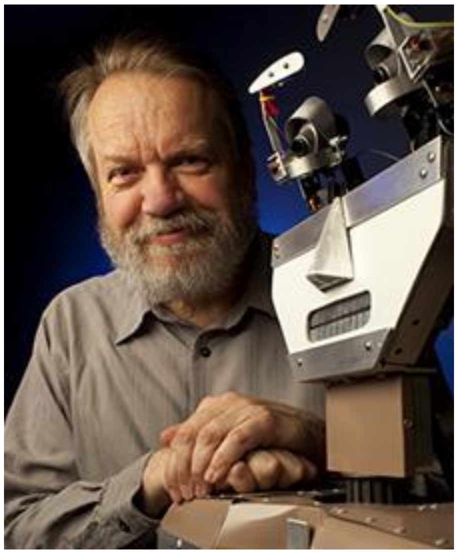


Figure 26 - <u>Professor Marek Perkowski poses with the DIM robot</u>

Chapter 5 - Conclusion and Future Work

A strong framework of matrix manipulation, Fourier function support and hardware interfacing made MATLAB a good medium for implementing the Scheirer BPM Algorithm. By parameterizing the inputs to the BPM functions it was possible to perform multiple variations of bandwidth and precision. After examining these results it was observed that one of the core aspects of the Scheirer Algorithm, filterbanks, aligned with the poorest performing parameter sets. Omitting filterbanks also greatly reduced the computation time. This in turn allowed the use of higher audio sample rates, improving the overall accuracy of the BPM results and a better user experience. The error percentage is less than 5% while the calculation time is less than 5 seconds.

The results obtained by the Rice group had high accuracy for many of theirs test but some of the samples had a high error gap. By parameterizing the inputs to the BPM functions it was possible to perform multiple variations of bandwidth and precision. This resulted in a number of data result sets that could be compared for speed and accuracy in BPM detection. After examining these results it was observed that one of the core aspects of the Scheirer Algorithm, filterbanks, aligned with the poorest performing parameter sets. Omitting filterbanks also greatly reduced the computation time. This in turn allowed the use of higher audio sample rates, improving the overall accuracy of the BPM results and a better user experience. The error percentage is less than 5% while the speed is less than 5 seconds.

My contribution consisted of separating the beat detection and robot control tasks of this Thesis. My approach used the Scheirer Beat Detection algorithm and parameterized the inputs to find the best results for speed and accuracy. This result was used to control the movements of a drumming robot in time with input audio. Other groups, such as Rice, used the Scheirer algorithm with its Fourier Transform method to detect BPM. My innovation was in using that calculated BPM result for the immediate control of robot drumming. The BPM information could be used to control any physical robot or system, and the robot could be controlled by any provided BPM information via byte information over the serial connection.

The Pololu Orangutan robot controller is a good choice for implementing the movement and logic of the Drumming Robot. Robot controllers differ from other development boards such as Raspberry Pi. The Raspberry Pi 3 processor is a quad-core 64-bit Cortex A53 at 1.2 GHz and 1 GB of memory is onboard, as well as wired + wireless LAN and video output. While these features are useful for some applications, the Raspberry Pi is missing key components for robot development. Orangutan boards in comparison include onboard motor power controllers, motor channel connectors and servo controllers. External boards may be added to the Raspberry Pi for these functions but may not have built in API support such as the Orangutan provides with the Atmel programming integration. Orangutan boards are designed to be a complete solution for logic, control and movement of robots in a single package.

The Orangutan robot controller uses an Atmel Studio C language development environment with a rich library of API functions to control the servo motors, buttons, LEDs and other features available with the robot controller. It is useful as a standalone manually controlled device, but also allows remote BPM input and control from the host over the serial connection. An audience can control the Drumming Robot using the buttons for a specific BPM, or the system can run in a continual loop where microphone audio BPM data is extracted and is controlling the Drumming Robot BPM.

Also, audio extraction would be a problem on the Orangutan since it has no onboard microphone. Maybe a future solution would be to use a different controller with the inputs and processing capabilities to input audio and perform Fourier operations in a reasonable amount of time. The development and testing process of this thesis leads to these conclusions: 1) MATLAB host processing is a viable method of beat detection; 2) the Pololu Orangutan robot controller is satisfactory for receiving serial BPM data and implementing beat output on a drumming robot.

Currently the Drumming Robot has six degrees of freedom between the shoulder and elbow control for the two drumming arms. In future work, the robot could be improved by adding more limbs (legs or more arms) and varying the percussion instruments. A bass drum, cowbell, floor tom or cymbal would give the audience a better experience. Also, the Scheirer BPM Algorithm could be implemented on the robot controller by developing functions to perform the Fourier Transform. However, there is no guarantee that this would be an improvement in speed. It is possible and even likely that the greater power of a host-based processor and memory outweigh the performance of the Orangutan. There is also the aforementioned problem of lack of microphone on the robot controller and the need to somehow interface an audio input system to the device and digitize the input signal. This is trivial on the host-side using MATLAB scripts.

This has been a satisfying Masters Thesis topic. The goal of a functional Drumming Robot system has been accomplished. On the host side the laptop microphone inputs external audio, and accurate BPM information is extracted using MATLAB and the Scheirer Algorithm. This information is sent to the controller, which performs a percussive drumming pattern using servo-powered robot arms and a drum head. By separating the development of detection and execution the thesis results are useful for various timekeeping robots (not just drumming) as well as any project requiring BPM information in real time. Both the Beat Detection and Robot Controller portions of this thesis would be useful for Perkowski’s Robot Theater at Portland State University.

REFERENCES

Scheirer, Eric D. (1997). *Tempo and beat analysis of acoustic musical signals.* MIT Media Laboratory.

Cheng, Kileen; Nazer, Bobak; Uppuluri, Jyoti; Verret, Ryan (2001). *Beat This – A Beat Synchronization Project.* Rice University.

Handel, Stephen (1989). *Listening.* MIT Press.

Large, Edward W. and Kolen, John F. (1994). *Resonance and the Perception of Musical Meter.* Connection Science, Vol. 6, Nos. 2 & 3, p. 177.

Afonso, Valtino X.; Thompkins, Willis J.; Nguyen, Truong Q.; Luo, Shen (1999). *ECG Beat Detection Using Filter Banks.* IEEE Transactions on Biomedical Engineering, Vol. 46, No. 2.

Povel, Dirk-Jan and Essens, Peter (1985). *Perception of Temporal Patterns.* Music Perception: An Interdisciplinary Journal, Vol. 2, No. 4.

Jackson, Preston A.; Chan, Cy P.; Scalera, Jonathan E.; Rader, Charles M.; Vai, Michael (2004). *A Systolic FFT Architecture for Real Time FPGA Systems.* MIT Lincoln Laboratory.

Buso, Vincent (2012). *VLSI Report.* Illinois Institute of Technology.

Gerez, Sabih H. (2013). *Hardware Implementations of the Fast Fourier Transform (FFT).* University of Twente, The Netherlands.

Meyer-Baese, U. (2007). *Digital Signal Processing with Field Programmable Gate Arrays.* Springer.

Witte, Robert A. (2014). *Spectrum and Network Measurements.* SciTech.

Smith, Steven W. (2011). *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing.

McKinney, Martin F. and Breebaart, Jeroen (2003). *Features for Audio and Music Classification.* Johns Hopkins University.

Lu, Li; Jiang, Hao; Zhang, HongJiang (2001). *A Robust Audio Classification and Segmentation Method.* ACM Multimedia.

O'Haver, Tom (2016). *A Pragmatic Introduction to Signal Processing.* University of Maryland at College Park.

Davies, Matthew E. P. and Plumbley, Mark D. (2006). *A Spectral Difference Approach to Downbeat Extraction in Musical Audio.* University of London Center for Digital Music.

Farid, Hany and Simoncelli, Eero P. (2004). *Differentiation of Discrete Multi-Dimensional Signals.* IEEE Transactions on Image Processing, Vol. 20, No. 20.

Appendices

A. Bill of Materials

Item Image	Description	Cost	Quantity	Subtotal
	Metal Straps - Pack of 20	$ 12.98	1	$ 12.98
	5" Rubber Swivel Caster	$ 22.96	2	$ 45.92
	Machine Screws + Nuts Kit	$ 3.97	1	$ 3.97
	Tower Pro MG995 High Torque Metal Gear Servo	$ 9.99	6	$ 59.94
	Orangutan SVP-1284 Robot Controller from Pololu	$ 99.95	1	$ 99.95
	Swing Arm Lamp	$ 10.00	2	$ 20.00
			Total	$ **242.76**

Table 8– Bill of Materials

B. List of Tools, Programs and Methods

- MATLAB 2007b
    - Audio input via microphone or .wav file
    - Host-side computation of BPM
    - Serial BPM output communication with robot controller
- Atmel Studio 7.0
    - C code software integrated development environment (IDE)
    - Creation of files for Orangutan robot controller
    - Orangutan robot controller .hex file flashing
- Pololu Orangutan SVP-1284
    - Runs looping BPM code from Atmel output
- Audacity
    - Creation of BPM Click Tracks for use in development and testing of BPM algorithm implementation on host-side
- Microsoft Movie Maker
    - Editing video files of drumming robot operation

C. Robot Controller Code

```
/* DrummingRobot - an application for the Pololu
Orangutan SVP
 *
 * This application uses the Pololu AVR C/C++ Library.
For help, see:
 * -User's guide: http://www.pololu.com/docs/0J20
 * -Command reference: http://www.pololu.com/docs/0J18
 *
 *  Author: mjengstx
 *
 * Updates: improved granularity of BPM output by
changing the control
 * character set to CHAR[a-r] (25 chars) over the 60-120
BPM range. Granularity is
 * now 61/25 = 2.44
 * Previously used CHAR[0-9] (10 chars) with a
granularity accuracy of 61/10 = 6.1
```

```
 * Assuming that the control character coming in from the
serial input is accurate,
 * the maximum Robot Drum output offset gap is improved
by 60%.
 */
#include <pololu/orangutan.h>
#include <string.h>


      /*
      * To use the SERVOs, you must connect the correct
AVR I/O pins to their
      * corresponding servo demultiplexed output-
selection pins.
      *   - Connect PB3 to SA.
      *   - Connect PB4 to SB.
      */


     // This array specifies the correspondence between
I/O pins and DEMUX
     // output-selection pins.  This demo uses three
pins, which allows you
```

```
     // to control up to 8 servos.  You can also use two,
one, or zero pins

     // to control fewer servos.

     //const unsigned char demuxPins[] = {IO_B3, IO_B4,
IO_C0}; // eight servos

     const unsigned char demuxPins[] = {IO_B3, IO_B4}; //
four servos

     //const unsigned char demuxPins[] = {IO_B3};
// two servos

     //const unsigned char demuxPins[] = {};
// one servo



     static unsigned char init_speed = 150;

     static unsigned char servo_speed = 150;

     static unsigned int neutral_servo_pos = 1300;

     //static unsigned int rt_shoulder_up = 300;

     //static unsigned int rt_shoulder_dn = 1300;

     //static unsigned int rt_shoulder = 1800;

     static unsigned int rt_shoulder_rot_lt = 2000;

     static unsigned int rt_shoulder_rot_rt = 1600;

     static unsigned int rt_shoulder_rot = 1600;
```

```
	static unsigned int rt_elbow_up = 1950;
	//ltdn

	static unsigned int rt_elbow_dn = 1775;
	//ltup

	static unsigned int rt_elbow = 1800;

	//static unsigned int lt_shoulder_up = 300;

	//static unsigned int lt_shoulder_dn = 1300;

	//static unsigned int lt_shoulder = 1800;

	static unsigned int lt_shoulder_rot_lt = 1200;

	static unsigned int lt_shoulder_rot_rt = 850;

	static unsigned int lt_shoulder_rot = 1200;

	static unsigned int lt_elbow_up = 1900;

	static unsigned int lt_elbow_dn = 2150;

	static unsigned int lt_elbow = 2200;


// receive_buffer: A ring buffer that we will use to
receive bytes on USB_COMM.

// The OrangutanSerial library will put received bytes in
to

// the buffer starting at the beginning
(receiveBuffer[0]).
```

```
// After the buffer has been filled, the library will
automatically

// start over at the beginning.

char receive_buffer[32];


// receive_buffer_position: This variable will keep track
of which bytes in the

// receive buffer we have already processed. It is the
offset(0-31) of the

// next byte in the buffer to process.


unsigned char receive_buffer_position = 0;


// send_buffer: A buffer for sending bytes on USB_COMM.

char send_buffer[32];


// sensor_buffer: A buffer for holding sensor bytes
received on USB_COMM.

//char sensor_buffer[5];

char mode[2];       // Changed to single char 3/22/13 -ME
char result[20];

int test = 0;

unsigned int pb_delay = 500;  //60 BPM Default starting
value

int flipper2 = 0;




int byte_counter = 0;

//string aNiceString = "";



// wait_for_sending_to_finish:  Waits for the bytes in
the send buffer to

// finish transmitting on USB_COMM.  We must call this
before modifying

// send_buffer or trying to send more bytes, because
otherwise we could

// corrupt an existing transmission.

void wait_for_sending_to_finish()

{

     while(!serial_send_buffer_empty(USB_COMM))
```

```
          serial_check();          // USB_COMM port is
always in SERIAL_CHECK mode

}


// process_received_byte: Responds to a byte that has
been received on

// USB_COMM.  If you are writing your own serial program,
you can

// replace all the code in this function with your own
custom behaviors.

void process_received_byte(char byte)

{

     clear();       // clear LCD

     print("Byte Received");

     lcd_goto_xy(0, 1);  // go to start of second LCD row

     print("RX: ");

     delay_ms(750);

/*

byte = '3';*/

     switch(byte)
     {
          // State Machine-style setup for incoming
Serial values; expecting ':::'
          // then single byte over Serial connection.
Increment 'byte_counter'
          // for each ':' until we have three, then next
Serial byte is valid.
          // Single byte is BPM with granularity of 6
from range 60-120.
          case ':':
               byte_counter += 1;
               print_character(byte);
               break;

          case 'a':
               test = 0;
               print_long(test);
               delay_ms(100);
               byte_counter += 1;
               break;
```

```
case 'b':

     test = 1;

     print_long(test);

     delay_ms(100);

     byte_counter += 1;

     break;


case 'c':

     test = 2;

     print_long(test);

     delay_ms(100);

     byte_counter += 1;

     break;


case 'd':

     test = 3;

     print_long(test);

     delay_ms(100);
```

```
          byte_counter += 1;

          break;


     case 'e':

          test = 4;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;


     case 'f':

          test = 5;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;


     case 'g':

          test = 6;
```

```
        print_long(test);

        delay_ms(100);

        byte_counter += 1;

        break;


    case 'h':

        test = 7;

        print_long(test);

        delay_ms(100);

        byte_counter += 1;

        break;


    case 'i':

        test = 8;

        print_long(test);

        delay_ms(100);

        byte_counter += 1;

        break;
```

```
case 'j':

     test = 9;

     print_long(test);

     delay_ms(100);

     byte_counter += 1;

     break;


case 'k':

     test = 10;

     print_long(test);

     delay_ms(100);

     byte_counter += 1;

     break;


case 'l':

     test = 11;

     print_long(test);

     delay_ms(100);

     byte_counter += 1;
```

```
          break;

     case 'm':
          test = 12;
          print_long(test);
          delay_ms(100);
          byte_counter += 1;
          break;

     case 'n':
          test = 13;
          print_long(test);
          delay_ms(100);
          byte_counter += 1;
          break;

     case 'o':
          test = 14;
          print_long(test);
          delay_ms(100);

          byte_counter += 1;

          break;


     case 'p':

          test = 15;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;


     case 'q':

          test = 16;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;


     case 'r':
          test = 17;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;


     case 's':

          test = 18;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;


     case 't':

          test = 19;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;
```

```
case 'u':

     test = 20;

     print_long(test);

     delay_ms(100);

     byte_counter += 1;

     break;


case 'v':

     test = 21;

     print_long(test);

     delay_ms(100);

     byte_counter += 1;

     break;


case 'w':

     test = 22;

     print_long(test);

     delay_ms(100);
```

```
          byte_counter += 1;

          break;


     case 'x':

          test = 23;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;


     case 'y':

          test = 24;

          print_long(test);

          delay_ms(100);

          byte_counter += 1;

          break;


     case 'z':

          test = 25;
```

```
                    print_long(test);

                    delay_ms(100);

                    byte_counter += 1;

                    break;

/*          case '0':

                    test = 0;

                    print_long(test);

                    delay_ms(400);

                    byte_counter += 1;

                    break;

                case '1':

                    test = 1;

                    print_long(test);

                    delay_ms(400);

                    byte_counter += 1;

                    break;

                case '2':

                    test = 2;

                    print_long(test);
```

```
        delay_ms(400);

        byte_counter += 1;

        break;

    case '3':

        test = 3;

        print_long(test);

        delay_ms(400);

        byte_counter += 1;

        break;

    case '4':

        test = 4;

        print_long(test);

        delay_ms(400);

        byte_counter += 1;

        break;

    case '5':

        test = 5;

        print_long(test);

        delay_ms(400);
        byte_counter += 1;

        break;

    case '6':

        test = 6;

        print_long(test);

        delay_ms(400);

        byte_counter += 1;

        break;

    case '7':

        test = 7;

        print_long(test);

        delay_ms(400);

        byte_counter += 1;

        break;

    case '8':

        test = 8;

        print_long(test);

        delay_ms(400);

        byte_counter += 1;
```

```
                        break;

                case '9':

                        test = 9;

                        print_long(test);

                        delay_ms(400);

                        byte_counter += 1;

                        break;

        */

                // Default is to place byte in 'send_buffer'

                default:

                        wait_for_sending_to_finish();

                        send_buffer[0] = byte;// ^ 0x20;


                        //green_led(TOGGLE);

                        //print(byte_counter);

                        //delay_ms(400);


                        break;

            }
```

```
}


void check_for_new_bytes_received()

{

     while(serial_get_received_bytes(USB_COMM) !=
receive_buffer_position)

     {

          // Process the new byte that has just been
received.


     process_received_byte(receive_buffer[receive_buffer_
position]);



          // Increment receive_buffer_position, but wrap
around when it gets to

          // the end of the buffer.

          if (receive_buffer_position ==
sizeof(receive_buffer)-1)

          {

               receive_buffer_position = 0;

          }
          else

          {

               receive_buffer_position++;

          }

     }

}


int main()

{




     servos_start(demuxPins, sizeof(demuxPins));



     // Set the servo speed to 150.  This means that the
pulse width

     // will change by at most 15 microseconds every 20
ms.  So it will

     // take 1.33 seconds to go from a pulse width of
1000 us to 2000 us.

     set_servo_speed(0, init_speed);
     set_servo_speed(1, init_speed);

     set_servo_speed(2, init_speed);

     set_servo_speed(3, init_speed);


     // Make all the servos go to a neutral position.

     set_servo_target(0, rt_shoulder_rot);   //right
shoulder rotation

     set_servo_target(1, rt_elbow);
     //right elbow

     set_servo_target(2, lt_shoulder_rot);   //left
shoulder rotation

     set_servo_target(3, lt_elbow);               //left
elbow


     clear();  // clear the LCD

     print("Robot Drummer");

     lcd_goto_xy(0, 1);  // go to start of second LCD row

     //print("or press Btn");

     print("Send BPM Mode");
     delay_ms(2000);



     // Set the baud rate to 9600 bits per second.  Each
byte takes ten bit

     // times, so you can get at most 960 bytes per
second at this speed.

     serial_set_baud_rate(USB_COMM, 9600);



     // Start receiving bytes in the ring buffer.

     serial_receive_ring(USB_COMM, receive_buffer,
sizeof(receive_buffer));



    while(1)

    {

          // USB_COMM is always in SERIAL_CHECK mode, so
we need to call this

          // function often to make sure serial
receptions and transmissions

          // occur.

          serial_check();
          // Deal with any new bytes received unless we
have a complete sample
          // of three ':' bytes, then 4th byte is desired
BPM byte
          if (byte_counter < 4)
          {
               check_for_new_bytes_received();
          }

           //NEW Mode value key:
           // a = 60 BPM
           // b = 62 BPM
           // c = 65 BPM
           // d = 68 BPM
           // e = 70 BPM
           // f = 72 BPM
           // g = 75 BPM
           // h = 78 BPM
           // i = 80 BPM
           // j = 82 BPM
```

```
 // k = 85 BPM

 // l = 88 BPM

 // m = 90 BPM

 // n = 92 BPM

 // o = 95 BPM

 // p = 98 BPM

 // q = 100 BPM

 // r = 102 BPM

 // s = 105 BPM

 // t = 108 BPM

 // u = 110 BPM

 // v = 112 BPM

 // w = 115 BPM

 // x = 118 BPM

 // y = 120 BPM


//OLD Mode value key:

// 0 = 60-65 BPM

// 1 = 66-71 BPM
```

```
          // 2 = 72-77 BPM

          // 3 = 78-83 BPM

          // 4 = 84-89 BPM

          // 5 = 90-95 BPM

          // 6 = 96-101 BPM

          // 7 = 102-107 BPM

          // 8 = 108-113 BPM

          // 9 = 114-120 BPM


          // The 'flipper2' variable in this section and
the next makes sure that

          // the drumming arms alternate beats. Only one
of the two drumming arms

          // strikes the drum per beat, and the other is
up in the air ready to

          // strike on the next beat.

          if ( flipper2 % 2 != 0 )

          {

               //set_servo_speed(0, servo_speed);

               set_servo_speed(1, servo_speed);
               //set_servo_speed(2, servo_speed);

               set_servo_speed(3, servo_speed);



               // Make all the servos go to a neutral
position.

               //set_servo_target(0, rt_shoulder_rot_lt);
     //right shoulder rotation

               set_servo_target(1, rt_elbow_dn);
          //right elbow

               //set_servo_target(2, lt_shoulder_rot_rt);
     //left shoulder rotation

               set_servo_target(3, lt_elbow_up);
     //left elbow

               //set_servo_target(3, lt_elbow_up);
          //make left elbow random for up

               if ( (rand()) % 2 != 0 )

               {

                    set_servo_target(3, lt_elbow_up);
          //left elbow

               }

               else
```

```
                {

                     set_servo_target(3, lt_elbow_dn);
           //left elbow

                }

           }

           else

           {

                //set_servo_speed(0, servo_speed);

                set_servo_speed(1, servo_speed);

                //set_servo_speed(2, servo_speed);

                set_servo_speed(3, servo_speed);


                // Make all the servos go to a neutral
position.

                //set_servo_target(0, rt_shoulder_rot_lt);
      //right shoulder rotation

                set_servo_target(1, rt_elbow_up);
           //right elbow

                //set_servo_target(2, lt_shoulder_rot_rt);
      //left shoulder rotation
```

```
                //set_servo_target(3, lt_elbow_dn);
          //make left elbow random for down

                if ( (rand()) % 2 != 0 )

                {

                     set_servo_target(3, lt_elbow_dn);
          //left elbow

                }

                else

                {

                     set_servo_target(3, lt_elbow_up);
          //left elbow

                }

          }



          flipper2 += 1;               // increment
flipper2 toggle value



          if (test == 0)               // 0 = serial
input 'a' = 60 BPM

          {

                clear();               // clear the LCD
                print("BPM = 60-61");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 500;

                //delay_ms(500);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter



           }

           else if (test == 1)           // 1 = serial
input 'b' = 62 BPM

           {

                clear();                 // clear the LCD

                print("BPM = 62-64");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row
```

```
               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 484;

               //delay_ms(440);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter



          }

          else if (test == 2)           // 2 = serial
input 'c' =65 BPM

          {



               clear();                 // clear the LCD

               print("BPM = 65-67");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);
               green_led(TOGGLE);

               pb_delay = 462;

               //delay_ms(400);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter



          }

          else if (test == 3)           // 3 = serial
input 'd' = 68 BPM

          {

               clear();                 // clear the LCD

               print("BPM = 68-69");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 441;

               //delay_ms(360);
               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter




          }

          else if (test == 4)           // 4 = serial
input 'e' = 70 BPM

          {

               clear();                 // clear the LCD

               print("BPM = 70-71");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 429;

               //delay_ms(345);

               servo_speed = 200;       // faster BPM
needs faster servo speed
```

```
               byte_counter = 0;          //reset counter


          }

          else if (test == 5)             // 5 = serial
input 'f' = 72 BPM

          {

               clear();                   // clear the LCD

               print("BPM = 72-74");

               lcd_goto_xy(0, 1);         // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 417;

               //delay_ms(335);

               servo_speed = 200;         // faster BPM
needs faster servo speed

               byte_counter = 0;          //reset counter


          }
          else if (test == 6)           // 6 = serial
input 'g' = 75 BPM

          {

               clear();                 // clear the LCD

               print("BPM = 75-77");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 400;

               //delay_ms(310);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter



          }

          else if (test == 7)           // 7 = serial
input 'h' = 78 BPM

          {
               clear();                 // clear the LCD

               print("BPM = 78-79");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 385;

               //delay_ms(290);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter



          }

          else if (test == 8)           // 8 = serial
input 'i' = 80 BPM

          {

               clear();                 // clear the LCD

               print("BPM = 80-81");
```

```
                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 375;

                //delay_ms(270);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter



          }

          else if (test == 9)            // 9 = serial
input 'j' = 82 BPM

          {

                clear();                 // clear the LCD

                print("BPM = 82-84");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");
```

```
               print_long(test);

               green_led(TOGGLE);

               pb_delay = 366;

               //delay_ms(250);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter


          }

          else if (test == 10)          // 10 = serial
input 'k' = 85 BPM

          {

               clear();                 // clear the LCD

               print("BPM = 85-87");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 353;
               //delay_ms(440);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter


          }

          else if (test == 11)          // 11 = serial
input 'l' = 88 BPM

          {


               clear();                 // clear the LCD

               print("BPM = 88-89");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 341;

               //delay_ms(400);
```

```
                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter



          }

          else if (test == 12)           // 12 = serial
input 'm' = 90 BPM

          {

                clear();                 // clear the LCD

                print("BPM = 90-91");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 333;

                //delay_ms(360);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter
          }

          else if (test == 13)          // 13 = serial
input 'n' = 92 BPM

          {

                clear();                 // clear the LCD

                print("BPM = 92-94");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 326;

                //delay_ms(345);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter


          }
          else if (test == 14)          // 14 = serial
input 'o' = 95 BPM

          {

                clear();                 // clear the LCD

                print("BPM = 95-97");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 316;

                //delay_ms(335);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter



          }

          else if (test == 15)          // 15 = serial
input 'p' = 98 BPM

          {
                clear();                 // clear the LCD

                print("BPM = 98-99");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 306;

                //delay_ms(310);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter



          }

          else if (test == 16)          // 16 = serial
input 'q' = 100 BPM

          {

                clear();                 // clear the LCD

                print("BPM = 100-101");
                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 300;

                //delay_ms(290);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter


          }

          else if (test == 17)          // 17 = serial
input 'r' = 102 BPM

          {

                clear();                 // clear the LCD

                print("BPM = 102-104");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");
                print_long(test);

                green_led(TOGGLE);

                pb_delay = 294;

                //delay_ms(270);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter



          }

          else if (test == 18)          // 18 = serial
input 's' = 105 BPM

          {

                clear();                 // clear the LCD

                print("BPM = 105-107");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 286;
```

```
                //delay_ms(250);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;
     //reset counter



           }

           else if (test == 19)          // 19 = serial
input 't' = 108 BPM

           {

                clear();                 // clear the LCD

                print("BPM = 108-109");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 278;

                //delay_ms(440);

                servo_speed = 200;       // faster BPM
needs faster servo speed
```

```
                byte_counter = 0;        //reset counter


            }

            else if (test == 20)         // 20 = serial
input 'u' = 110 BPM

            {


                clear();                 // clear the LCD

                print("BPM = 110-111");

                lcd_goto_xy(0, 1);       // go to start of
second LCD row

                print("mode: ");

                print_long(test);

                green_led(TOGGLE);

                pb_delay = 273;

                //delay_ms(400);

                servo_speed = 200;       // faster BPM
needs faster servo speed

                byte_counter = 0;        //reset counter
```

```
          }

          else if (test == 21)          // 21 = serial
input 'v' = 112 BPM

          {

               clear();                 // clear the LCD

               print("BPM = 112-114");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 268;

               //delay_ms(360);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter




          }
          else if (test == 22)          // 22 = serial
input 'w' = 115 BPM

          {

               clear();                 // clear the LCD

               print("BPM = 115-117");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 261;

               //delay_ms(345);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter


          }

          else if (test == 23)          // 23 = serial
input 'x' = 118 BPM

          {
               clear();                 // clear the LCD

               print("BPM = 118-119");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 254;

               //delay_ms(335);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter



          }

          else if (test == 24)          // 24 = serial
input 'y' = 120 BPM

          {

               clear();                 // clear the LCD

               print("BPM = 120");
               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");

               print_long(test);

               green_led(TOGGLE);

               pb_delay = 250;

               //delay_ms(310);

               servo_speed = 200;       // faster BPM
needs faster servo speed

               byte_counter = 0;        //reset counter



          }

          else if (test == 25)          // 25 = serial
input 'z' = PAUSED

          {

               clear();                 // clear the LCD

               print("PAUSED");

               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("mode: ");
               print_long(test);

               delay_ms(200);

               byte_counter = 0;        //reset counter

               flipper2 = 1;            // set flipper2
toggle value to 1 so

                                             // that arms
stop drumming in this mode



          }

          //Default mode of 60 BPM

          else

          {

               green_led(TOGGLE);

               clear();  // clear the LCD

               print("Robot Drummer");

               lcd_goto_xy(0, 1);  // go to start of
second LCD row

               print("Default mode");

               pb_delay = 500;

               //delay_ms(pb_delay);
```

```
                servo_speed = 200;      // faster BPM
needs faster servo speed


          }


          delay_ms(pb_delay);      //moved delay out of
'else if' tests to here


          // If the user presses the middle button, send
"Robots Rule!"

          // and wait until the user releases the button.

          if (button_is_pressed(MIDDLE_BUTTON))

          {

               wait_for_sending_to_finish();

               memcpy_P(send_buffer, PSTR("Robots
Rule!\r\n"), 12);

               serial_send(USB_COMM, send_buffer, 12);

               send_buffer[12] = 0;     // terminate the
string

               clear();                 // clear the LCD
               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("Delay (ms): ");

               print_long(pb_delay);



               delay_ms(1000);

               byte_counter = 0;        // reset detect
cycle by pressing button



               // Wait for the user to release the
button.  While the processor is

               // waiting, the OrangutanSerial library
will not be able to receive

               // bytes from the USB_COMM port since this
requires calls to the

               // serial_check() function, which could
cause serial bytes to be

               // lost.  It will also not be able to send
any bytes, so the bytes

               // bytes we just queued for transmission
will not be sent until
```

```
               // after the following blocking function
exits once the button is
               // released.
               wait_for_button_release(MIDDLE_BUTTON);
          }
          // If the user presses the TOP button,
increment BPM Mode by 1
          if (button_is_pressed(TOP_BUTTON))
          {
               wait_for_sending_to_finish();
               clear();                 // clear the LCD
               print("BPM Mode Up");

               if (test <= 25)               // BPM Mode
'10' is wait state
               {
                    test = test + 1;
               }
               lcd_goto_xy(0, 1);       // go to start of
second LCD row

               print("To Mode ");

               print_long(test);

               delay_ms(1000);

               byte_counter = 0;        // reset detect
cycle by pressing button

               wait_for_button_release(TOP_BUTTON);

          }

          // If the user presses the BOTTOM button,
decrement delay by 10 ms

          if (button_is_pressed(BOTTOM_BUTTON))

          {

               wait_for_sending_to_finish();

               clear();                 // clear the LCD

               print("BPM Mode Down");


               if (test >= 1)      //fastest speed,

               {

                    test = test - 1;
```

```
            }


            lcd_goto_xy(0, 1);       // go to start of
second LCD row

            print("To Mode ");

            print_long(test);

            delay_ms(1000);

            byte_counter = 0;        // reset detect
cycle by pressing button

            wait_for_button_release(BOTTOM_BUTTON);

        }

    }

}
```

D. Host MATLAB Code

MatlabSerialCode.m

```
%Interface for launching Beat Detection script

%This is the script to run for starting Beat Detection
using MATLAB

%#########################################################
##################

%#######################--MATLAB CODE--
###################################

%#########################################################
##################

%What is the COM port name?

%       input COM port name

%       SerBEAT = serial([Com port name]);

%Do you want to use a .wav file or the microphone?

% if file then

%       input file name

%       test = strcat('xx', num2str(control([file
name]))

% else

% ...
```

```
%

%NOTE:

%If MATLAB gives a serial error, it will most likely say
'unable to open

%serial port' next time you run the program; restart
MATLAB to recover.

%

loop_val = 1;

repeat_val = 0;

Mode = 'a';



prompt = 'Enter your COM port: ';

com_str = input(prompt,'s');

% make sure com port is CAPITALS

str = upper(com_str);

prompt2 = strcat(str, ': Is this correct? Y/N [Y]: ');

str2 = input(prompt2,'s');

if (strcmp(str2, 'Y') || strcmp(str2, 'y'))
```

```
    SerBEAT = serial(str); %<--SET UP SERIAL CONNECTION
IN MATLAB

    set(SerBEAT,'BaudRate', 9600, 'DataBits', 8,
'Parity', 'none','StopBits', 1, 'FlowControl', 'none');

    fopen(SerBEAT); %--open the serial port to the PIC



    while loop_val == 1

    BPM = 1;

         if (repeat_val ~= 1)

             prompt3 = ('Enter .wav file name, ENTER for
microphone, or z to pause: ');

            str3 = input(prompt3,'s');

            if isempty(str3)    %if ENTER has been
pressed

                BPM = control_accurate();

                %BPM = control_optimizer();

            elseif (strcmp(str3, 'Z') || strcmp(str3,
'z'))

                BPM = 1;     %set to zero so Mode check
drops out to PAUSE

            else
```

```
                BPM = control_accurate(str3);
                %BPM = control_optimizer(str3);
            end
         else
            BPM = control_accurate();
            pause(1);
         end


        %%%%%%%%%%%Logic for beat mode to send to Robot
Controller%%%%%%%%%
%         while ((BPM > 120) || (BPM < 60))
%             if BPM > 120
%                 BPM = (BPM / 2);
%             end
%             if BPM < 60
%                 BPM = (BPM * 2);
%             end
%         end
        BPM
%        Mode = (ceil((BPM - 59)/6)-1);
```

```
if ((BPM > 59) && (BPM < 62))          %60-61 BPM
    Mode = 'a'; %60 BPM
elseif ((BPM > 61) && (BPM < 65))     %62-64 BPM
    Mode = 'b'; %62 BPM
elseif ((BPM > 64) && (BPM < 68))     %65-67 BPM
    Mode = 'c'; %65 BPM
elseif ((BPM > 67) && (BPM < 70))     %68-69 BPM
    Mode = 'd'; %68 BPM
elseif ((BPM > 69) && (BPM < 72))     %70-71 BPM
    Mode = 'e'; %70 BPM
elseif ((BPM > 71) && (BPM < 75))     %72-74 BPM
    Mode = 'f'; %72 BPM
elseif ((BPM > 74) && (BPM < 78))     %75-77 BPM
    Mode = 'g'; %75 BPM
elseif ((BPM > 77) && (BPM < 80))    %78-79 BPM
    Mode = 'h'; %78 BPM
elseif ((BPM > 79) && (BPM < 82))    %80-81 BPM
    Mode = 'i'; %80 BPM
elseif ((BPM > 81) && (BPM < 85))   %82-84 BPM
```

```
        Mode = 'j'; %82 BPM
    elseif ((BPM > 84) && (BPM < 88))   %85-87 BPM
        Mode = 'k'; %85 BPM
    elseif ((BPM > 87) && (BPM < 90))   %88-89 BPM
        Mode = 'l'; %88 BPM
    elseif ((BPM > 89) && (BPM < 92))   %90-91 BPM
        Mode = 'm'; %90 BPM
    elseif ((BPM > 91) && (BPM < 95))   %92-94 BPM
        Mode = 'n'; %92 BPM
    elseif ((BPM > 94) && (BPM < 98))   %95-97 BPM
        Mode = 'o'; %95 BPM
    elseif ((BPM > 97) && (BPM < 100))   %98-99 BPM
        Mode = 'p'; %98 BPM
    elseif ((BPM > 99) && (BPM < 102))   %100-101 BPM
        Mode = 'q'; %100 BPM
    elseif ((BPM > 101) && (BPM < 105))   %102-104
BPM
        Mode = 'r'; %102 BPM
    elseif ((BPM > 104) && (BPM < 108))   %105-107
BPM
```

```
            Mode = 's'; %105 BPM
        elseif ((BPM > 107) && (BPM < 110))   %108-109
BPM
            Mode = 't'; %108 BPM
        elseif ((BPM > 109) && (BPM < 112))   %110-111
BPM
            Mode = 'u'; %110 BPM
        elseif ((BPM > 111) && (BPM < 115))   %112-114
BPM
            Mode = 'v'; %112 BPM
        elseif ((BPM > 114) && (BPM < 118))   %115-117
BPM
            Mode = 'w'; %115 BPM
        elseif ((BPM > 117) && (BPM < 120))   %118-119
BPM
            Mode = 'x'; %118 BPM
        elseif ((BPM > 119) && (BPM < 121))   %120 BPM
            Mode = 'y'; %120 BPM
        elseif (BPM == 0)    %user has input a z or Z
            Mode = 'z';         %Drum is PAUSED
        end
        Mode

        pause(1);

        %%%%%%%%%%%%End Beat Mode Logic%%%%%%%%%%

        %test = num2str(Mode);

        test = strcat(':::', num2str(Mode));

    %    for s = 1: 1: 100

            fprintf(SerBEAT, '%s', test); %--send BPM
mode to Orangutan Robot Controller

    %        pause(0.1);

    %    end


        if (repeat_val ~= 1)

            prompt4 = 'Press: R=Repeat, B=BPM Detect Loop
(CTRL+C to exit), or Q=finish: ';

            str4 = input(prompt4,'s');


            if (strcmp(str4, 'Q') || strcmp(str4, 'q'))

                loop_val = 0;

                fclose(SerBEAT) %--close the serial port
when done
```

```
                delete(SerBEAT)

                clear SerBEAT


            elseif (strcmp(str4, 'B') || strcmp(str4,
'b'))

                repeat_val = 1;

            elseif (strcmp(str4, 'R') || strcmp(str4,
'r'))

                repeat_val = 0;

            end

        end

        pause(1);

            %continue to end of script

    end

else

    %exit

end


%NOTE 1:
```

```
%if MATLAB ever gives a serial error, it will most likely
say 'unable to

%open serial port' next time you

%run the program, so you have to restart MATLAB

%

%http://www.instructables.com/id/MATLAB-to-PIC-serial-
interface/
```

control.m

```
function output=control_accurate(song1, bandlimits,
maxfreq)

% CONTROL takes in the names of two .wav files, and
outputs their

% combination, beat-matched, and phase aligned.

%

%     SIGNAL = CONTROL(SONG1, SONG2, BANDLIMITS, MAXFREQ)
takes in

%     the names of two .wav files, as strings, and
outputs their

%     sum. BANDLIMITS and MAXFREQ are used to divide the
signal for

%     beat-matching

%

%     Defaults are:

%        BANDLIMITS = [0 200 400 800 1600 3200]

%        MAXFREQ = 4096


  if nargin < 1, song1 = 'None'; end
  if nargin < 2, bandlimits = [0]; end

  if nargin < 3, maxfreq = 16384; end


  % Length (in power-2 samples) of the song

  sample_size = floor(16*maxfreq);

  scaling = 0.73;   % Experimentally derived


  % Takes in the two wave files

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%  RECORDING LOGIC  %%%%%%%%%%%%

if (strcmp(song1, 'None'))

     recObj = audiorecorder;

     disp('Start of Recording')

     recordblocking(recObj, 10);

     disp('End of Recording');

    x1 = getaudiodata(recObj);

    short_sample = x1;

else

    x1 = wavread(song1);
    short_song = x1;

    short_length = length(x1);

    start = floor(short_length/2 - sample_size/2);

    stop = floor(short_length/2 + sample_size/2);


  % Finds a 5 second representative sample of each song

  short_sample = short_song(start:stop);

end


  % Implements beat detection algorithm for each song

  a = filterbank(short_sample, bandlimits, maxfreq);

  b = hwindow(a, 0.1, bandlimits, maxfreq);

  c = diffrect(b, length(bandlimits));


  % Recursively calls timecomb to decrease computational
time

  d = timecomb(c, 5, 60, 240, bandlimits, maxfreq);

  e = timecomb(c, .5, d-2, d+2, bandlimits, maxfreq);

  f = timecomb(c, .1, e-.5, e+.5, bandlimits, maxfreq);

  g = timecomb(c, .01, f-.1, f+.1, bandlimits, maxfreq);
  h = floor(scaling*g);


  % We want 60-120 BPM, so scale harmonics into range.
Assume 240 Max
  % and 15 Min BPM in audio input sample.
  if ((h > 120) || (h < 60)) % Only scale if out of range
      if (h < 30 )
          h = 3*h;      %double if less than 60, assume
never below 30BPM
      elseif ((h > 30) && (h < 60))
          h = 2*h;      %double if less than 60, assume
never below 30BPM
      elseif (h > 121)
          h = 0.5*h;    %halve if more than 120 but less
than 180
      %assume never over 300
      end
  end
  short_song_bpm = floor(h);
  output = short_song_bpm;
```

filterbank.m

```
function output = filterbank(sig, bandlimits, maxfreq)

% FILTERBANK divides a time domain signal into individual
frequency

% bands.

%

%     FREQBANDS = FILTERBANK(SIG, BANDLIMITS, MAXFREQ)
takes in a

%     time domain signal stored in a column vector, and
outputs a

%     vector of the signal in the frequency domain, with
each

%     column representing a different band. BANDLIMITS is
a vector

%     of one row in which each element represents the
frequency

%     bounds of a band. The final band is bounded by the
last

%     element of BANDLIMITS and  MAXFREQ.
```

```
%
%     Defaults are:
%        BANDLIMITS = [0 200 400 800 1600 3200]
%        MAXFREQ = 4096
%
%     This is the first step of the beat detection
sequence.
%
%     See also HWINDOW, DIFFRECT, and TIMECOMB


  if nargin < 2, bandlimits=[0 200 400 800 1600 3200];
end
  if nargin < 3, maxfreq=4096; end


  dft = fft(sig);
  n = length(dft);
  nbands = length(bandlimits);


  % Bring band scale from Hz to the points in our vectors
  for i = 1:nbands-1
```

```
    bl(i) = floor(bandlimits(i)/maxfreq*n/2)+1;

    br(i) = floor(bandlimits(i+1)/maxfreq*n/2);

  end



  bl(nbands) = floor(bandlimits(nbands)/maxfreq*n/2)+1;

  br(nbands) = floor(n/2);

  output = zeros(n,nbands);



  % Create the frequency bands and put them in the vector
output.

  for i = 1:nbands

    output(bl(i):br(i),i) = dft(bl(i):br(i));

    output(n+1-br(i):n+1-bl(i),i) = dft(n+1-br(i):n+1-
bl(i));

  end



  %output(1,1)=0;
```

hwindow.m

```
function output = hwindow(sig, winlength, bandlimits,
maxfreq)

% HWINDOW rectifies a signal, then convolves it with a
half Hanning

% window.

%

%     WINDOWED = HWINDOW(SIG, WINLENGTH, BANDLIMITS,
MAXFREQ) takes

%     in a frequecy domain signal as a vector with each
column

%     containing a different frequency band. It
transforms these

%     into the time domain for rectification, and then
back to the

%     frequency domain for multiplication of the FFT of
the half

%     Hanning window (Convolution in time domain). The
output is a

%     vector with each column holding the time domain
signal of a
```

```
%     frequency band. BANDLIMITS is a vector of one row
in which
%     each element represents the frequency bounds of a
band. The
%     final band is bounded by the last element of
BANDLIMITS and
%     MAXFREQ. WINLENGTH contains the length of the
Hanning window,
%     in time.
%
%     Defaults are:
%        WINLENGTH = .4 seconds
%        BANDLIMITS = [0 200 400 800 1600 3200]
%        MAXFREQ = 4096
%
%     This is the second step of the beat detection
sequence.
%
%     See also FILTERBANK, DIFFRECT, and TIMECOMB


  if nargin < 2, winlength = .4; end
  if nargin < 3, bandlimits = [0 200 400 800 1600 3200];
end

  if nargin < 4, maxfreq = 4096; end


  n = length(sig);

  nbands = length(bandlimits);

  hannlen = winlength*2*maxfreq;

  hann = [zeros(n,1)];


  % Create half-Hanning window.

  for a = 1:hannlen

    hann(a) = (cos(a*pi/hannlen/2)).^2;

  end


  % Take IFFT to transfrom to time domain.

  for i = 1:nbands

    wave(:,i) = real(ifft(sig(:,i)));

  end


  % Full-wave rectification in the time domain.
  % And back to frequency with FFT.

  for i = 1:nbands

    for j = 1:n

      if wave(j,i) < 0

     wave(j,i) = -wave(j,i);

      end

    end

    freq(:,i) = fft(wave(:,i));

  end


  % Convolving with half-Hanning same as multiplying in

  % frequency. Multiply half-Hanning FFT by signal FFT.
Inverse

  % transform to get output in the time domain.

  for i = 1:nbands

    filtered(:,i) = freq(:,i).*fft(hann);

    output(:,i) = real(ifft(filtered(:,i)));

  end
```

diffrect.m

```
function output=diffrect(sig,nbands)

% DIFFRECT differentiates signal, then half-wave
rectifies the result.

%

%     DIFF = DIFFRECT(SIG, NBANDS) takes in a time domain
signal

%     stored in a vector with each column representing a
different

%     frequency band. The number of frequency bands is
passed in

%     through NBANDS.

%

%     Defaults are:

%        NBANDS = 6

%

%     This is the third step of the beat detection
sequence

%     See also FILTERBANK, HWINDOW, and TIMECOMB

if nargin <2, nbands=6; end
```

```
n = length(sig);

output=zeros(n,nbands);

for i = 1:nbands

   for j = 5:n

     % Find the difference from one sample to the next

     d = sig(j,i) - sig(j-1,i);

     if d > 0

       % Retain only if difference is positive (Half-Wave
rectify)

       output(j,i)=d;

     end

   end

end
```

timecomb.m

```
function output = timecomb(sig, acc, minbpm, maxbpm,
bandlimits, maxfreq)
```

```
% TIMECOMB finds the tempo of a musical signal, divided
into
% frequency bands.
%
%     BPM = TIMECOMB(SIG, ACC, MINBPM, MAXBPM,
BANDLIMITS, MAXFREQ)
%     takes in a vector containing a signal, with each
band stored
%     in a different column. BANDLIMITS is a vector of
one row in
%     which each element represents the frequency bounds
of a
%     band. The final band is bounded by the last element
of
%     BANDLIMITS and MAXFREQ. The beat resolution is
defined in
%     ACC, and the range of beats to test is  defined by
MINBPM and
%     MAXBPM.
%
%     Defaults are:
%        ACC = 1
%         MINBPM = 60

%         MAXBPM = 240

%         BANDLIMITS = [0 200 400 800 1600 3200]

%         MAXFREQ = 4096

%

%      Note that timecomb can be recursively called with
greater

%      accuracy and a smaller range to speed up
computation.

%

%      This is the last step of the beat detection
sequence.

%

%      See also FILTERBANK, HWINDOW, and DIFFRECT



  if nargin < 2, acc = 1; end

  if nargin < 3, minbpm = 60; end

  if nargin < 4, maxbpm = 240; end

  if nargin < 5, bandlimits = [0 200 400 800 1600 3200];
end
```

```
if nargin < 6, maxfreq = 4096; end


n=length(sig);

bpms = [0,0,0,0,0,0,0,0,0,0];

bpms_cnt = 1;

nbands=length(bandlimits);


% Set the number of pulses in the comb filter

npulses = 3;


% Get signal in frequency domain

for i = 1:nbands

  dft(:,i)=fft(sig(:,i));

end


% Initialize max energy to zero

maxe = 0;


for bpm = minbpm:acc:maxbpm
```

```
    % Initialize energy and filter to zero(s)

    e = 0;

    fil=zeros(n,1);

    % Calculate the difference between peaks in the
filter for a

    % certain tempo

    nstep = floor(120/bpm*maxfreq);


    % Set every nstep samples of the filter to one

    for a = 0:npulses-1

      fil(a*nstep+1) = 1;

    end


    % Get the filter in the frequency domain

    dftfil = fft(fil);


    % Calculate the energy after convolution

    for i = 1:nbands

      x = (abs(dftfil.*dft(:,i))).^2;

      e = e + sum(x);
    end


    % If greater than all previous energies, set current
bpm to the
    % bpm of the signal
    if e > maxe
      sbpm = bpm;
      bpms(bpms_cnt) = sbpm;
      bpms_cnt = bpms_cnt + 1;
      maxe = e;
    end
  end


  output = sbpm;
```

[i] Eric D. Scheirer, “Tempo and beat analysis of acoustic musical signals,” *J. Acoust Soc. Am Vol* 103, no. 1 (Jan 1998): 588-601

[ii] McKinney, Martin F. and Breebaart, Jeroen, “Features for Audio and Music Classification,” Johns Hopkins University (2003)

[iii] Handel, Stephen, “Listening,” MIT Press (July 1989)

[iv] Large, Edward W. and Kolen, John F., “Resonance and the Perception of Musical Meter”, Connection Science, Vol. 6, Nos. 2 & 3 (1994)

[v] Eric D. Scheirer, “Tempo and beat analysis of acoustic musical signals,” *J. Acoust Soc. Am Vol* 103, no. 1 (Jan 1998): 588-601

[vi] Povel, Dirk-Jan and Essens, Peter, “Perception of Temporal Patterns,” Music Perception: An Interdisciplinary Journal, Vol. 2, No. 4 (1985)

[vii] Valtino X. Afonso, Willis J. Tompkins, Truong Q. Nguyen, and Shen Luo, “ECG Beat Detection Using Filter Banks,” IEEE TRANSACTIONS ON BIOMEDICAL ENGINEERING, VOL. 46, NO. 2 (Feb 1999)