Beat Extraction with Application to Robot Drumming

~

Michael Engstrom

Abstract

The goal of this thesis is to design a robot which can play drums in rhythm to an external audio source. The audio source can be either a pre-recorded track or a live sample 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 fast BPM extraction algorithm developed by Eric Scheirer 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.

The tasks of control of the robot motion and the BPM extraction of the input audio are separated. The main advantage of this approach is that by creating a generic interface between Input Logic and Robot Control each could be used independently for application to multiple robots or control systems (explain what systems). A host computer inputs audio from the environment (via microphone) and extracts the BPM data with the Scheirer algorithm to send 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 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 noise and a microphone would input the audio to be processed to control the motion of the Drumming Robot.

Table of Contents

List of Figures

List of Equations

List of Tables

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. **add quotes for Handel reference**[Handel, 1989] 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. **add Handel mp3 classification background**

Fourier Transforms can detect frequency power trends to determine the beat of an audio sample. This decomposed audio beat information can be used to control mechanical output, such as the Drumming Robot. In his paper 'Tempo and beat analysis of acoustical musical signals' Eric Scheirer describes a beat extraction algorithm which is effective in determining the BPM of an audio sample [Scheirer, 1996]. The use of Fourier Transforms is effective for immediate analysis of BPM information.

[Describe other beat-extraction approaches, and why mine is superior]

For this Drumming Robot thesis, the Beat Extraction 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 section [SECTION] of this thesis. By testing multiple parameter combinations, it was possible to optimize the accuracy and speed of the algorithm which resulted in improved performance quality for the audience.

A host Input Logic system which 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 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.

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. Any input, if it is in the correct format, can be used with this drumming robot. This includes other BPM extraction methods approaching real-time input.

A drumming robot preferably exhibits human-like motion. It has been observed that articulating lamp sections resemble jointed limbs; this humanoid resemblance of lamp arms has been utilized for this thesis. Two 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. This construction resulted in three degrees of freedom for each arm, or six degrees of freedom total.

This flow diagram is followed by an explanation of the algorithm steps:

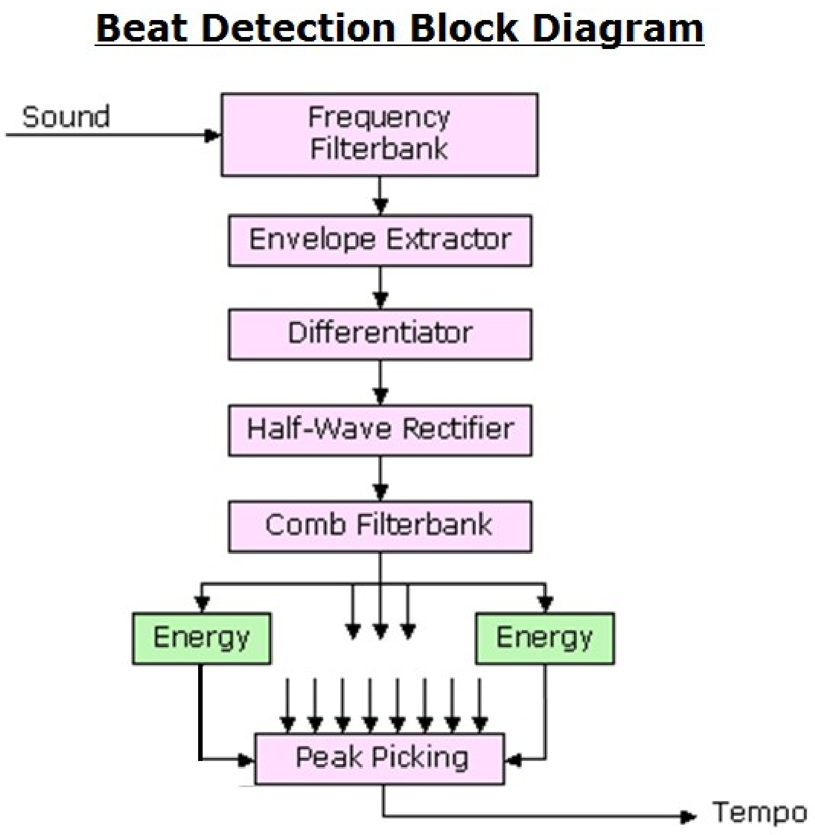


Figure 1 – Beat Detection Diagram

- Beat Detection Algorithm steps:
  1. Frequency Filter Bank
     - Split frequency range of sample into smaller segments
  2. Envelope Extraction (Fourier Transform)
     - Frequency power is extracted
  3. Differentiation
     - Smoothing of extracted signal
  4. Rectification

    - Isolation of desired frequency information

5. Multiple Comb Filters (Resonant Filterbank)
    - Match frequency sample to known BPM
    - Peak-Picking
    - Best Fit is our desired 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 [EXPOUND on MATLAB input features], matrix manipulation for audio data, Fourier processing functions, and serial connection functions. 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 extraction and serial output operations.

For live sound input a signal from a microphone 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. Resulting data arrays both have stored frequency information that can be varied as a parameter from 2048 to 16384 samples, in powers of 2[CLARIFY]. Varying this parameter affects the accuracy and the processing time of the algorithm.

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 filter is implemented using a sixth-order elliptical filter. This results in 3dB of ripple in the passband and 40 dB of rejection in the stopband, with sharply defined ranges for each frequency band. 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. [reference?] In Figure 2 and Figure 3 are plots of the bands, separated to show the drop-offs:[ADD full explanations of figures in captions]

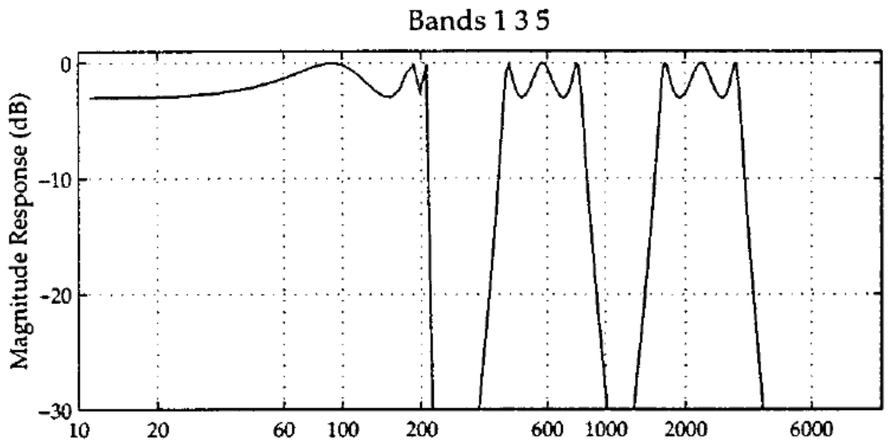


Figure 2 – Filterbank Bands 1, 3, 5

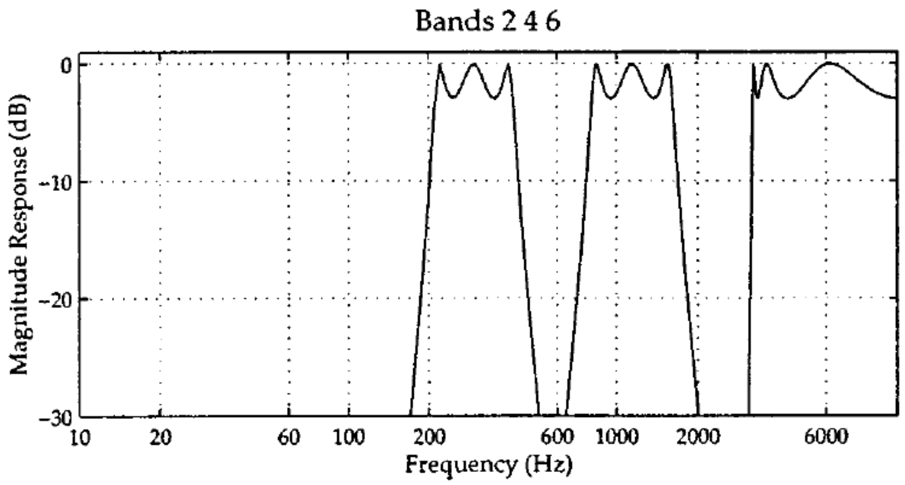


Figure 3 – Filterbank Bands 2, 4, 6

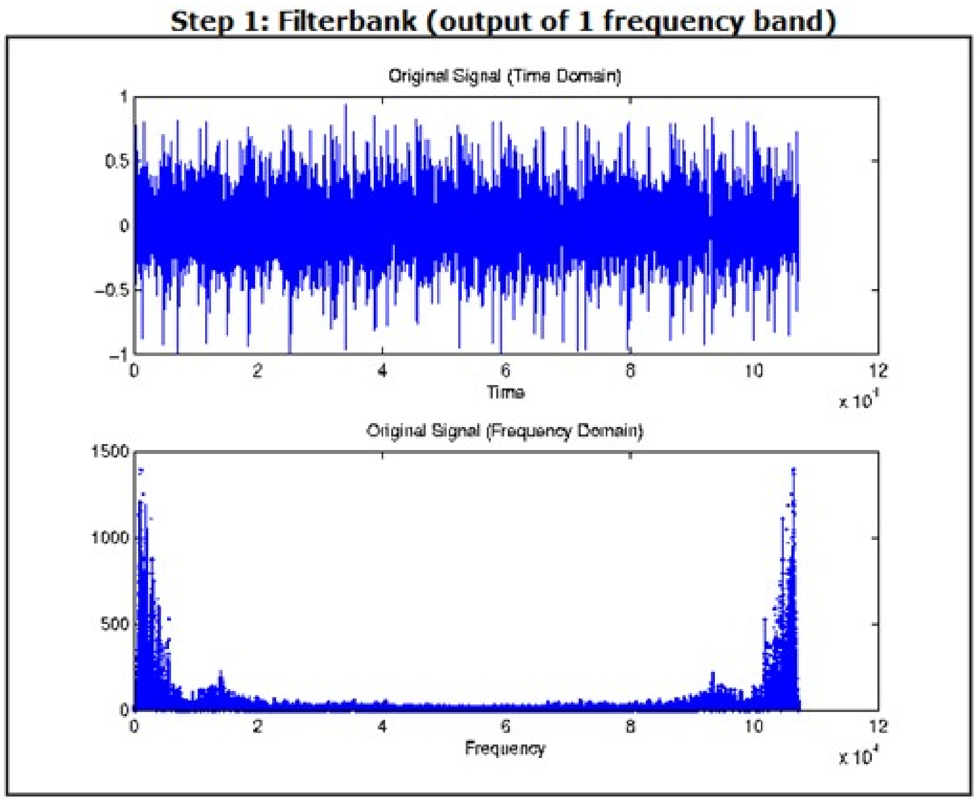


Figure 4 – Step 1: Frequency Filterbank

Step 2: Envelope Extractor

Next 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 data, but this is inefficient. 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).

[add section to explain DSP and FT, with the benefits – needs to be exhaustive review for readers not familiar with FT]

The signal is then transformed using a Hanning Window to clean up the frequency range and improve signal clarity [National Instruments][explain why windowing is useful]:

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

Equation 1 – Hanning Windowing Filter

Windowing the input signal can improve the accuracy of the resulting signal[EXPLAIN]. 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 5). For BPM extraction as presented in this thesis, it is desired to lower the non-center frequency amplitude to improve the result of later multiplying the signal with the comb filters. Therefore, because it suppresses lower and higher frequencies, the Hanning window was chosen as a better implementation [source = NI]:

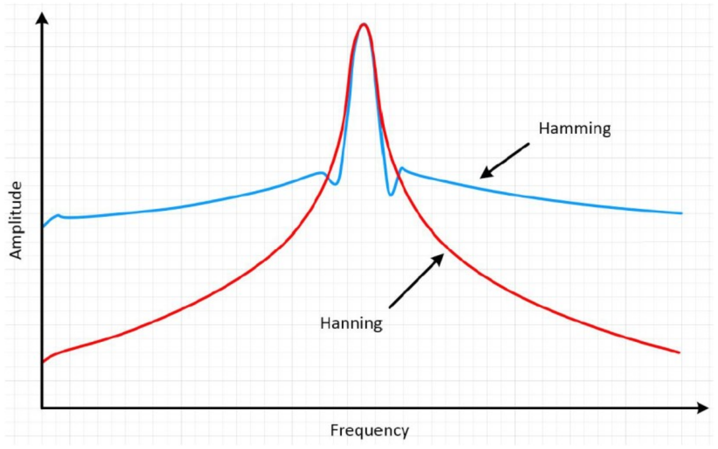
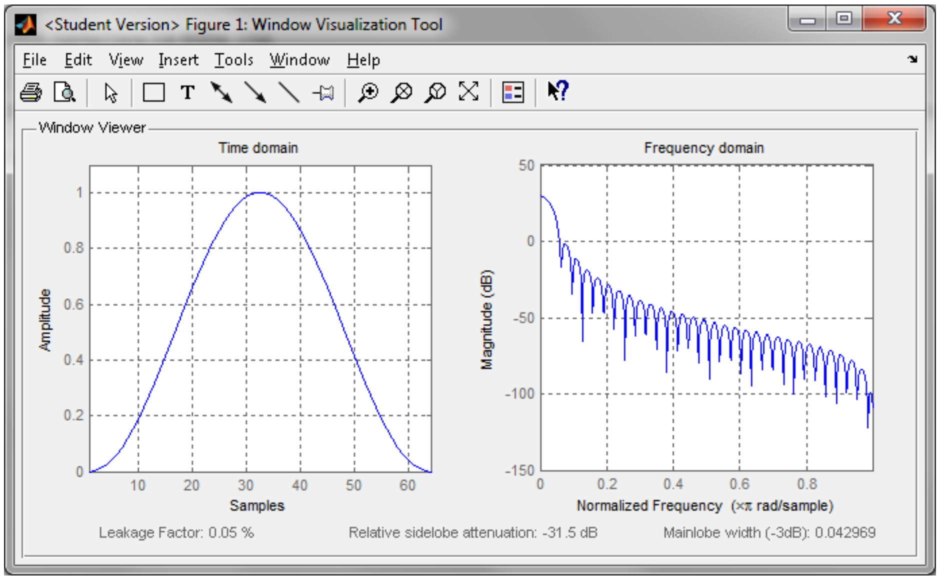


Figure 5 – Hanning and Hamming Windowing Filters

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 which improves the result in the desired range. 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 6 of a generated signal using N = 64.


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

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

Figure 6 – Windowing Example in MATLAB

Figure 7 shows an example of an input signal before windowing (left) and after (right); notice that the data shape is retained, duplicated signal information is removed, and noise is reduced:

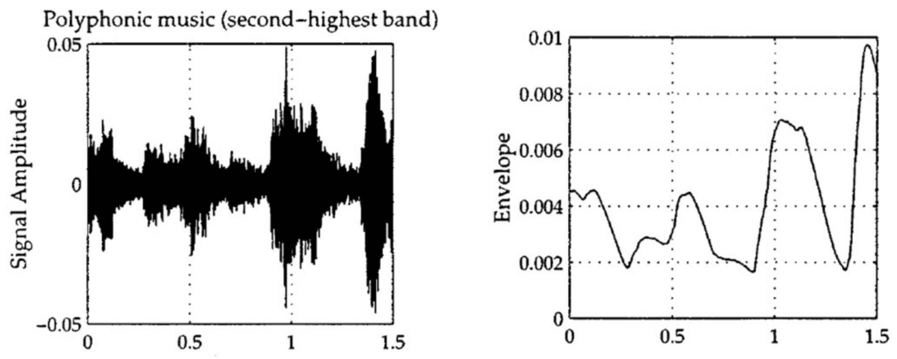


Figure 7 - Signal Amplitude and Envelope

Signal processing is greatly simplified by performing these complex Time Domain operations in the Frequency Domain using the Fourier Transform. [EXPLAIN AGAIN] 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 [CHANGE to show only one transform, not the pair- I do not convert back to Time Domain in my algorithm]where *X(f)* is the Frequency Domain signal and *x(t)* is the Time Domain signal as shown in Equation 2:

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

Equation 2 – Fourier Transform

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 – Discrete Fourier Transform

After the original signal is converted to the Frequency Domain by using the Fourier Transform the data is represented in a [POWER-FREQ?] spectrum as a measure of power. 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 8 is an example of an FFT of an input signal, showing highest frequency power at about 150 BPM and a slightly less power peak at 75 BPM.

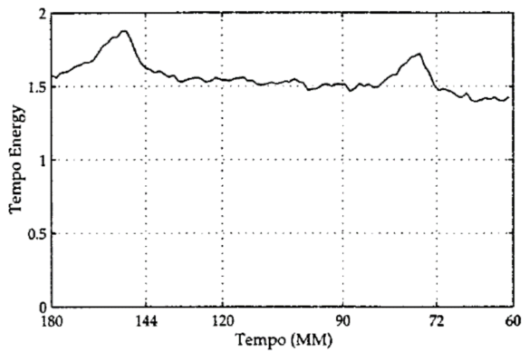


Figure 8 – Signal Tempo (Frequency) vs. Tempo Energy (Power)

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 8 would be considered to be 75 BPM even though it has slightly less power than the harmonic at 150 BPM.

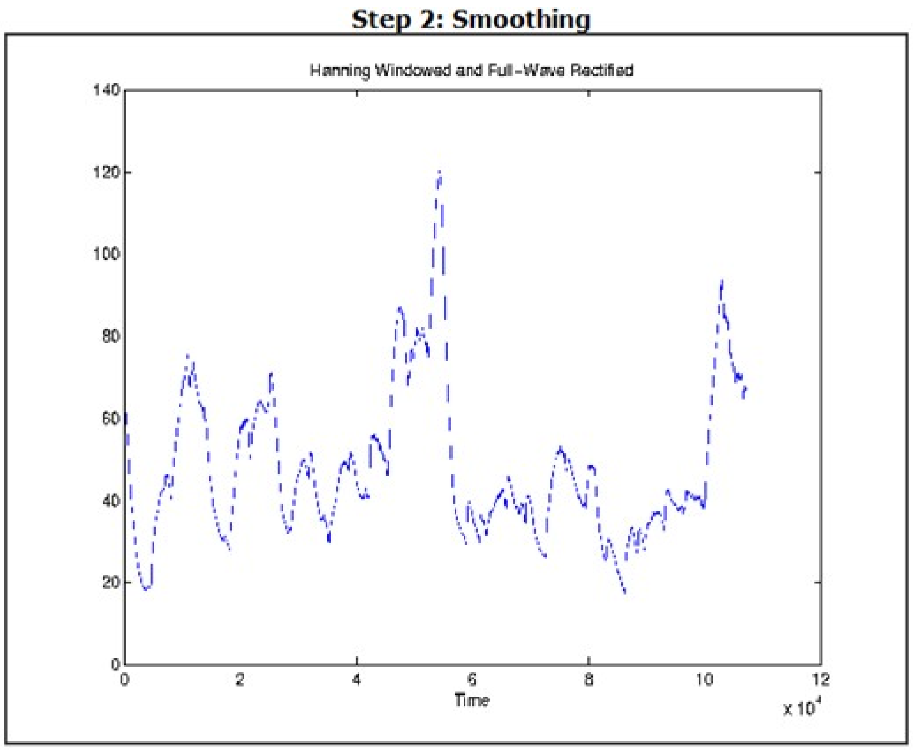


Figure 9 – Step 2: Smoothing

In algorithm steps 3 and 4 the signal is further processed to improve the accuracy of the final BPM determination.[[END of FIRST REVIEW SESSION]]

Steps 3 and 4– Differentiate and Rectify Signal

The differential of each digital sample to the next is calculated and retained only in case of positive results, giving a half-wave rectified output signal. Differentiating a signal in MATLAB is accomplished 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. Below is the input sine wave (red) and half-wave output (blue):

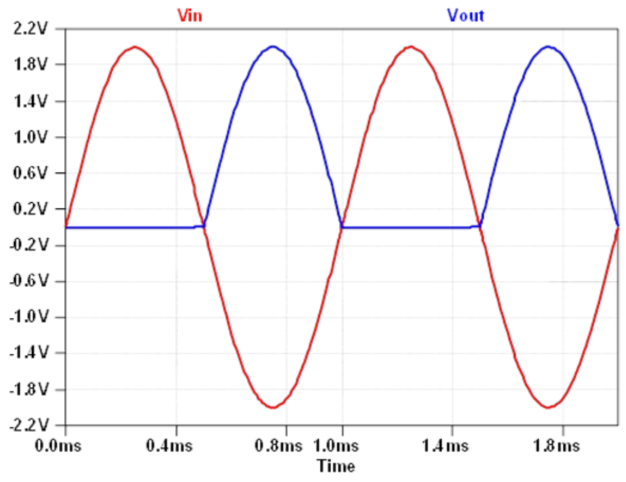


Figure 10 – Half-Wave Rectification

(Source: Analog Devices WiKi page)

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

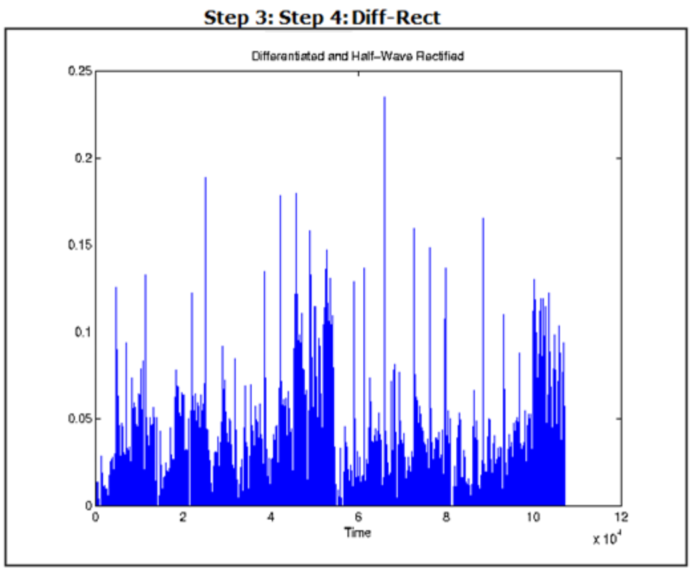


Figure 11 – Step 3: Differentiation and Step 4: Rectification

The higher power peaks are isolated and allow 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.

Step 5 – Comb Filter

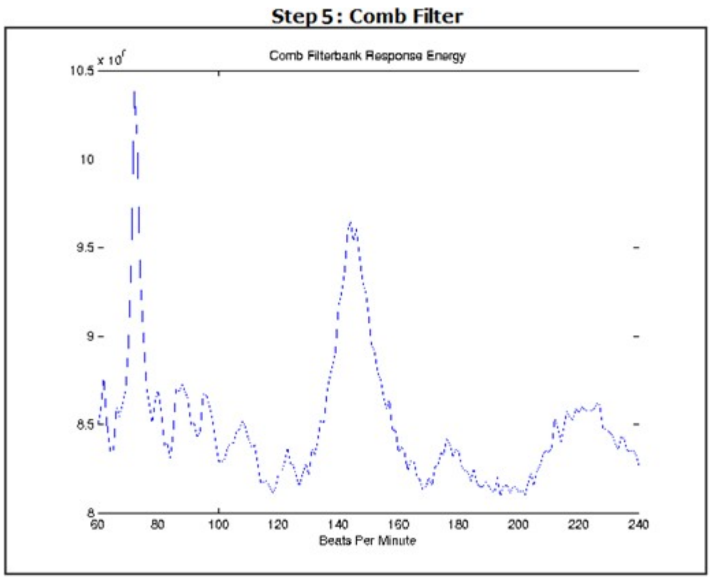


Figure 12 – Step 5: Comb Filter

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 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 as

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

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*, then

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

Equation 7 – Output Signal

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 MIT 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. [reference] MATLAB was chosen for the built in functions for accessing audio input using computer microphones, a core goal of this thesis for use in the PSU Robot Theater. MATLAB also has functions for establishing serial communication links.[expand explanation]

In the thesis planning stages the decision was made to develop the host BPM extraction feature separately from the Robot Controller development. This decision was made in part because the host and controller use different technologies (MATLAB and C code Atmel/Orangutan Robot Controller, respectively).[explain each separately to avoid confusion] 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 extraction method or desired control.

The Beat Extraction 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 as

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

Equation 8 – BPM Granularity

Below is pseudo code for implementing the Beat Detection Algorithm in MATLAB: (Possibly move to beginning of thesis)

1. Input audio from file or microphone
    a. .wav file or 10 seconds of microphone sample
2. Follow BPM Algorithm
    a. EXPLAIN
        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 the code is included in Appendix B (link?). Since the MATLAB BPM algorithm is spread over multiple files, FIGURE X flowchart helps explain the data flow and algorithm steps. The input file to the algorithm is the digital audio matrix:

[INSERT FLOWCHART]

Algorithm Evaluation and Optimization

Once the software was working it was important to optimize the BPM function. The performance of the beat extraction 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 (reference here or previously or a section that describes all tools, programs, methods, public domain, where to obtain) with known BPM values. The range of 60-120 BPM was included, in 5 BPM granularity, and a few outlier BPMs were added to test robustness. The set includes:

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

For each set, parameters were varied and the resultant time per BPM and averaged error from the known BPM were measured and graphed. Parameters used:

	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

This experiment resulted in 20 different Time vs. Error data points. These were graphed for comparison in Figure 13. 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

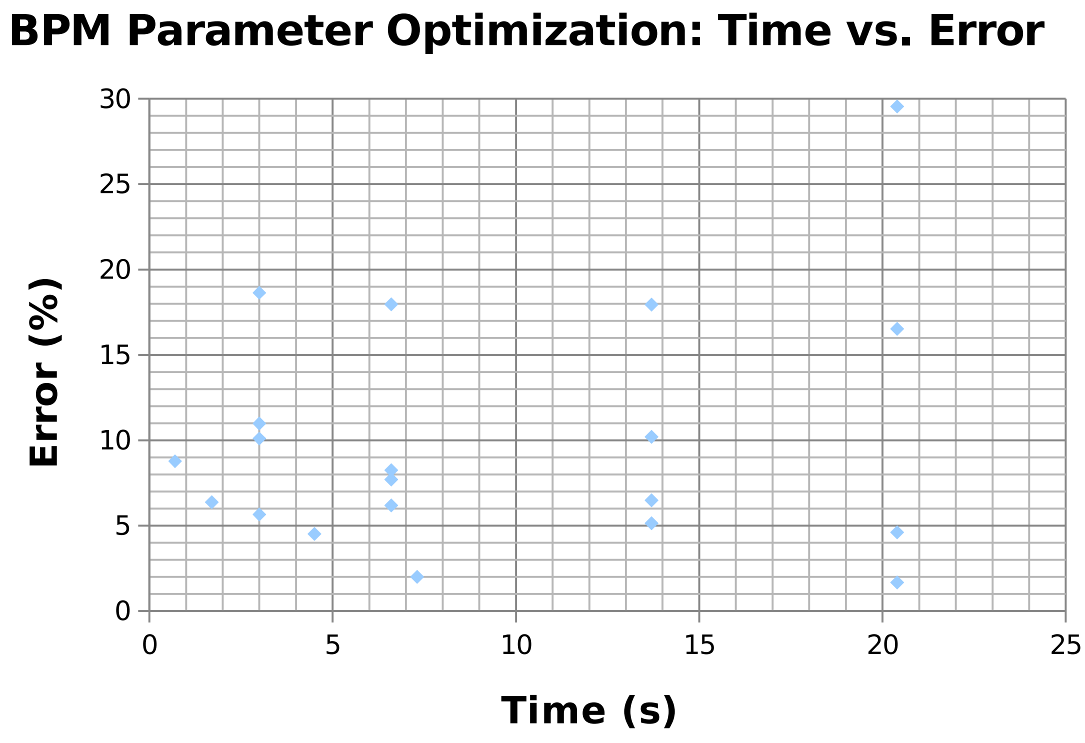


Figure 13 – Parameter Optimization Results

Looking at the graph in Figure 13, the data point with 4.5 seconds calculation time and 4.51% average error for the BPM exceeded our error and time goals.

This data point is illustrated in Table 3. (explain why this is the best result and what is the meaning of colors green, yellow, etc.)

(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 3 – Optimized Parameter Set

Using the parameter values in Table 3 as the final parameter set, we can be confident that our input algorithm is both fast and accurate. This parameter set also 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. The parameter set that meets our time and error goals, while using the band limits (per the Handel Algorithm) and no scaling, is the data point with 6.6 seconds calculation time and 8.25% average error for the BPM as presented in Table 4. (explain yellow color, and maybe 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 4 – Parameter Set: No Scaling

The time and error results for the (6.6s, 6.18%) parameter set in Table 4 are acceptable. However, for the operation of the robot speed and accuracy are desired so the parameter configuration used will be the (4.5s, 4.51%)

parameter set from Table 3. (explain better why Table 3 was preferred to Table 4 parameters)

Chapter 3 – Robot Design

Figure 14 – 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 initializes the servos, serial interface, and LCD display, then sets the arms to drum in 60 BPM. Button inputs allow for increase or decrease of BPM. In addition, the Orangutan continually monitors the serial bus for byte inputs of [a,z] (corresponding to BPM granularity of 2 BPM increments of 60-120). [explain state machine, why it was used, how it was implemented] This software state machine controls the implementation of the BPM in the code running on the robot controller. Buttons can only increment sequentially from 60 to 120 BPM by values of 5. A serial input will immediately change the Drumming Robot to the desired BPM mode. Figure 15 demonstrates a state machine diagram showing the button and serial inputs, as well as the BPM delay and LCD output.

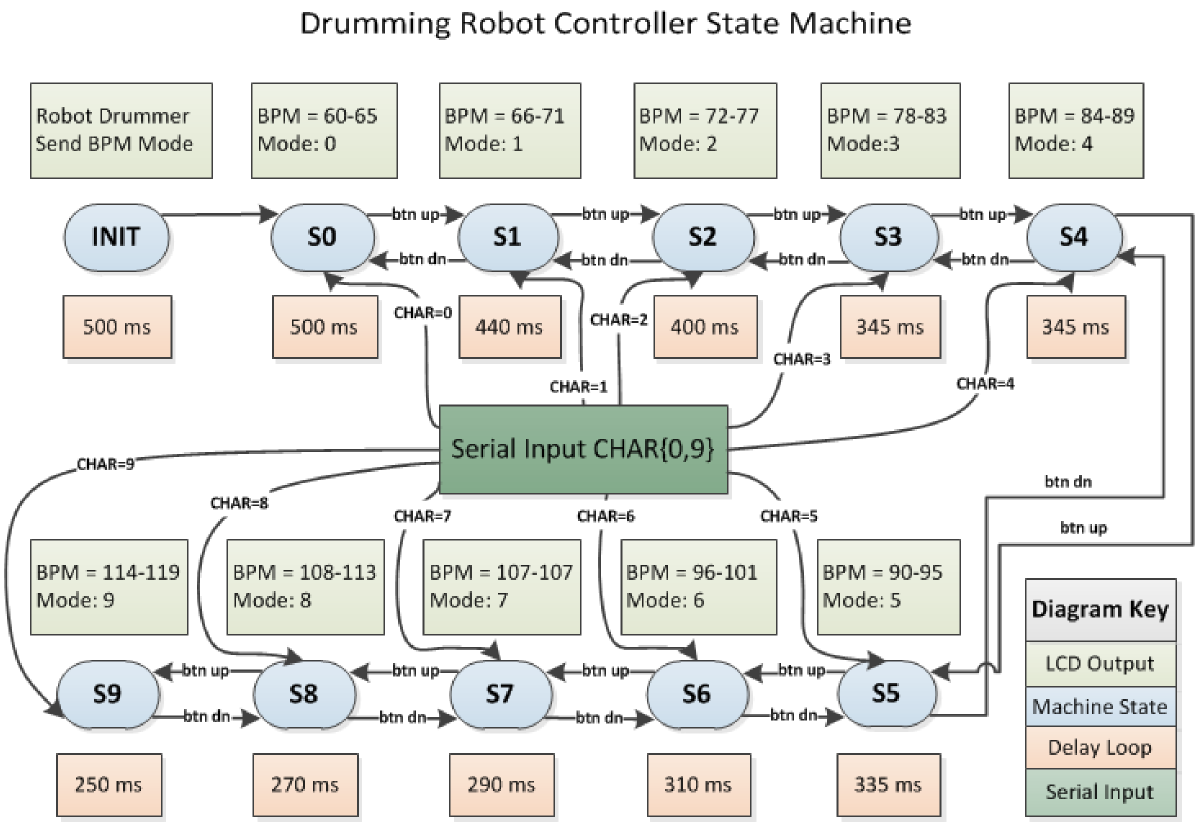


Figure 15 – Robot Controller State Machine

[Explain arrows, squares, circles, program flow with regard to Figure 15]

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.

```
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 is bidirectional. Pressing the middle button on the Orangutan sends a 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. [add section for troubleshooting byte input on serial bus] It is also possible to send other information such as servo position, servo speed, loop delay and other desired values. This is not currently implemented. Table 5 shows the input bytes, Robot Controller states and the necessary delay needed per loop for the desired BPM cadence.

Input	**Mode**	**BPM**	**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 5 – Loop Delay Calculation

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[BE MORE Specific as to which languages]. [Describe controller boards, the drawbacks of HDL, and the advantages of boards which use high-level languages] The Orangutan Robot Controller Board from Pololu was chosen for this thesis. 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. [compare to raspberry pi-type boards, advantages, drawbacks, purpose]

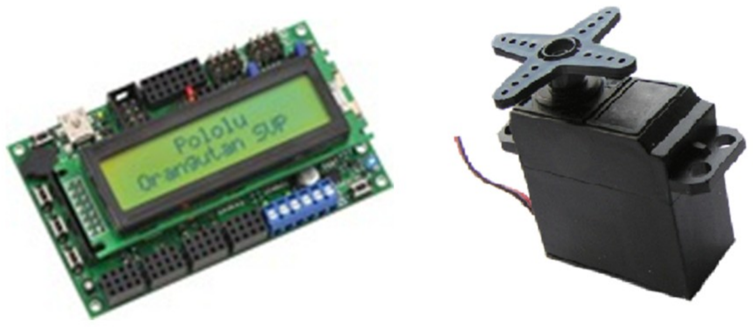
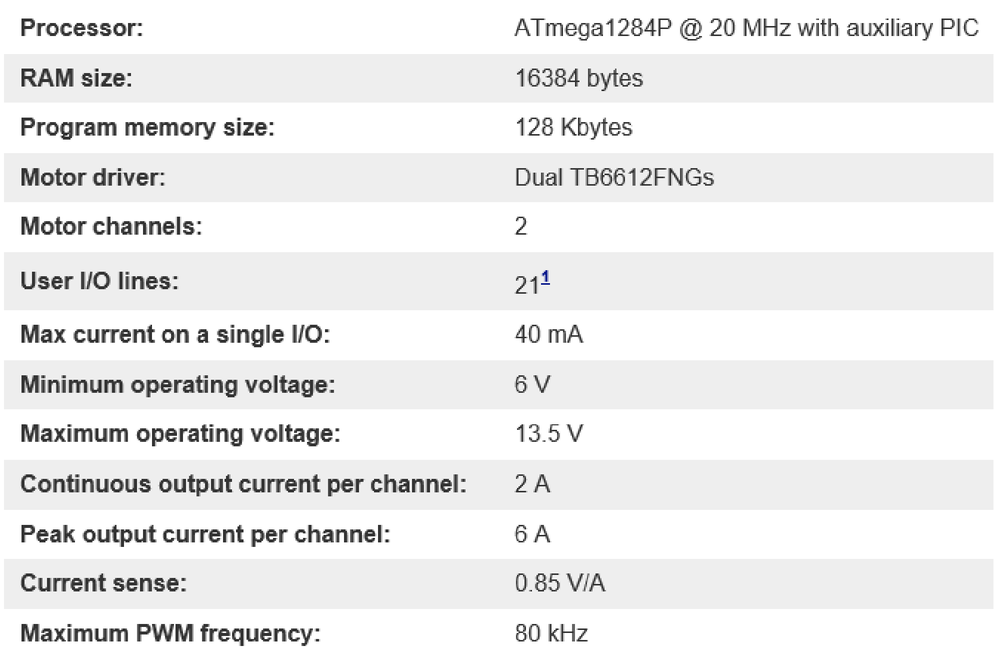


Figure 16 – Robot Controller and Servo Motor

The Orangutan boards are cheap and can be purchased at www.pololu.com for about $100, and downloads are available with many examples for the controller boards. [include verbose examples of projects] 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 6 for features.



Processor:	ATmega1284P @ 20 MHz with auxiliary PIC
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 6 – Orangutan Robot Controller Specifications

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 and 1-2 ms ‘high’ time. Changing the pulse signal changes the arm position. 1ms is one extreme, 2ms is the other extreme, and 1.5 ms pulses put the arm about in the middle. 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.

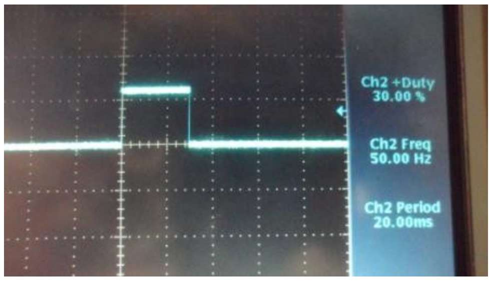


Figure 17 – Servo Pulse Waveform

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.[explain why] Later this battery pack was replaced with an AC power supply, which provided enough power to run the board and the attached servos.

An issue with servos is the current reflection 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 (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.

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. 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. [figure out how to add torque calculations with physics equations]

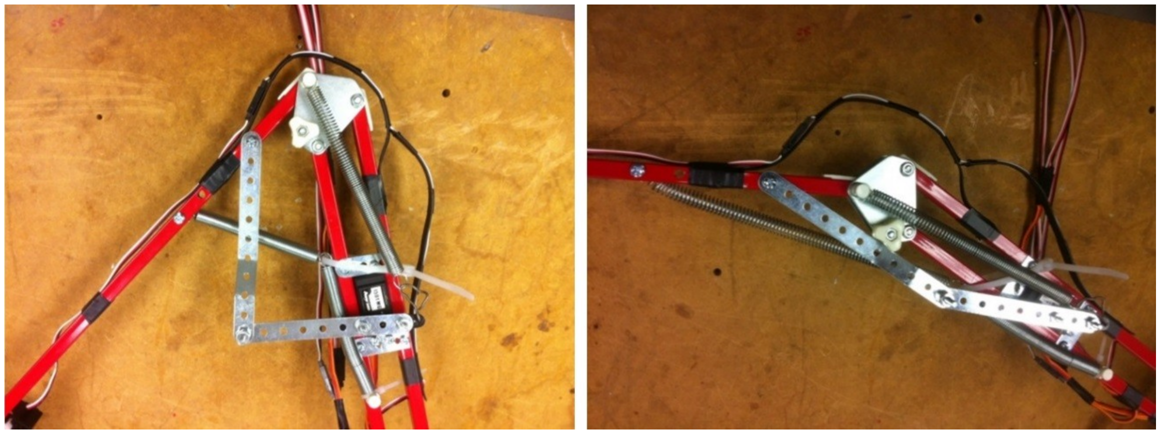


Figure 18 – Robot Elbow Range of Motion

The design of the shoulders was more interesting. 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.

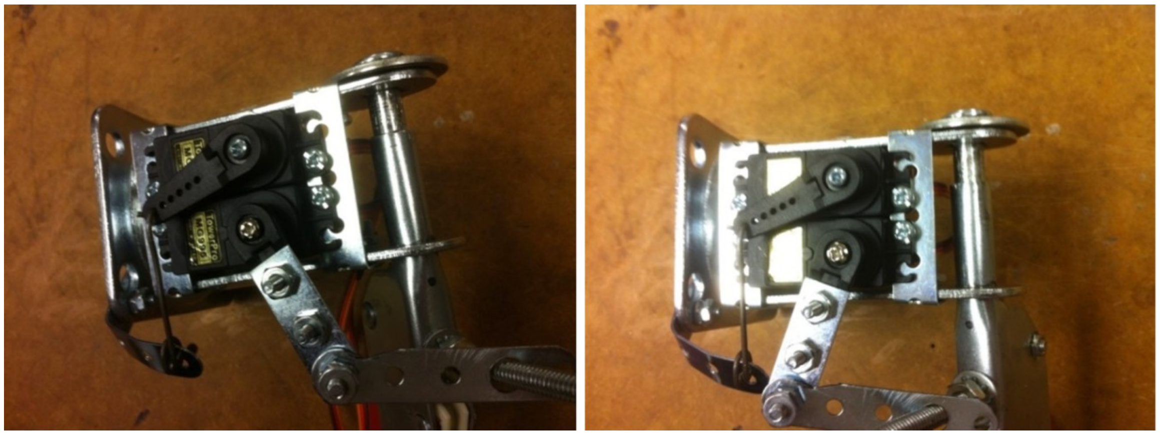


Figure 19 – Robot Shoulder Left/Right Motion

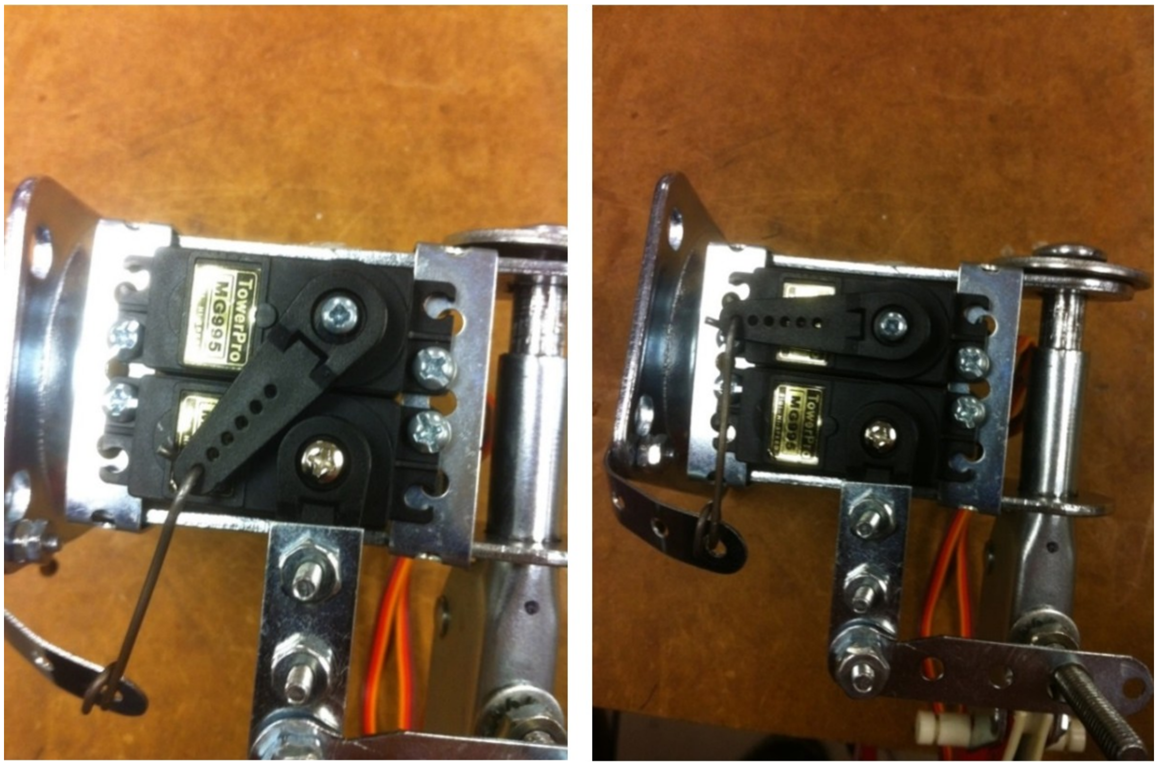


Figure 20 – Robot Shoulder Up/Down Motion

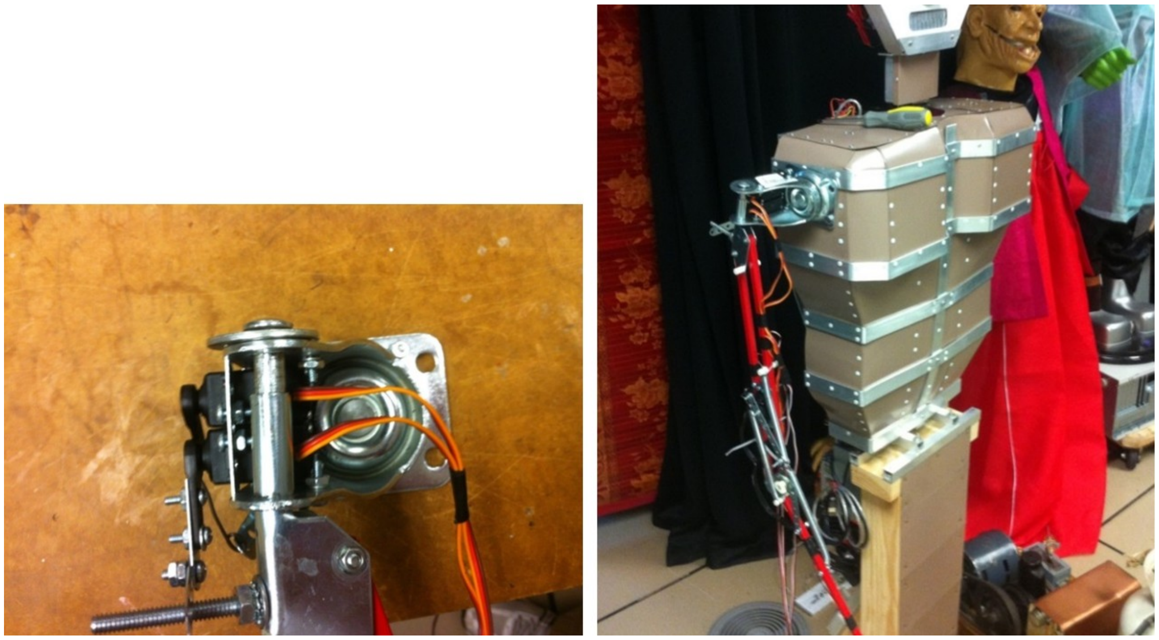


Figure 21 – Robot Arm Mounted on Robot Torso

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 arms' attached drumsticks were not striking the drum head in a precise position, causing skips in beats when too high and stressing the servo motors when too low. These 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 'Orangutans and Servo Control' 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 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 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 with the target BPM.

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!

Chapter 5 - Conclusion and Future Work

1. Contributions of this Thesis
2. Conclusions
3. Future Work (integrated in Robot T, etc.)

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 extraction 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.[find other drumming robots, how they work, how they compare to my drumming robot]

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 speed is less than 5 seconds.

The Orangutan robot controller was a good choice for implementing the movement and logic of the Drumming Robot. [recap comparison to the raspberry pi] The Orangutan used 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 controlling the Drumming Robot BPM.

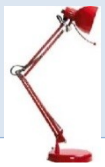
Currently the Drumming Robot has six degrees of freedom between the shoulder and elbow control. In future work, the robot could be improved by adding more limbs (legs or more arms) and varying the percussion instruments. A bass drum, cowbell, floor tom or cymbal would give the audience a better experience. Also, the Scheirer BPM Algorithm could be implemented on the robot controller. However, there is no guarantee that this would be an improvement in speed. It is possible and even likely that a host-based processor and memory greatly outweigh the performance of the Orangutan. Also, the audio extraction is 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.

REFERENCES

- *Tempo and beat analysis of acoustic musical signals*
  - Scheirer, Eric D.
  - MIT Media Laboratory, 1997
- Beat Sync Project
  - http://www.clear.rice.edu/elec301/Projects01/beat_sync/
  - Rice University, 2001
- *Listening*
  - Handel, S.
  - MIT, Cambridge, MA, 1989
- A Systolic FFT Architecture for Real Time FPGA Systems
  - Preston A. Jackson, Cy P. Chan, Jonathan E. Scalera, Charles M. Rader, and M. Michael Vai
  - MIT Lincoln Laboratory, 2004
- VLSI Report
  - Vincent Buso
  - Illinois Institute of Technology, 2012
- Hardware Implementations of the Fast Fourier Transform (FFT)
  - Sabih H. Gerez
  - University of Twente, The Netherlands, 2013
- Digital Signal Processing with Field Programmable Gate Arrays
  - U. Meyer-Baese
  - 3rd edition, Springer, 2007

Appendices

A. Bill of Materials

	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 7 – Bill of Materials

B. Robot Controller Code

Change Font, add current a-y code, explain why greater granularity was used

```
/* 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);
            }
     }
   }
```

C. Host MATLAB Code

control.m

make sure there is a diagram for how the functions talk to each other, and flow

also add wrapper MATLAB code for setting up serial connection

```
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;
```