

# Rhythm Detection in Recorded Music

Joseph E. Flannick<sup>1</sup>  
Department of Mathematics and Computer Science  
Saint Joseph's University.

July 8, 2003

<sup>1</sup>Faculty Advisors: Dr. Rachel Hall and Dr. Adlai Waksman

## **Abstract**

My research concerns rhythm detection in recorded music. Using the matrix mathematics system MATLAB, we analyzed the rhythmic structure of several musical selections and compared the results to our musical analysis by ear. Rhythm is a low frequency component of music, in contrast to pitch, which is a high frequency component. Our first task was to filter out the high frequencies in order to study rhythm. Our data reduction algorithm separates recorded music into frequency bands, of roughly half an octave each, and extracts the energy in each band. Once this was done, the Discrete Fourier Transform can identify the primary rhythmic content of each band. We found that the time signatures of the pieces we analyzed were characterized by the rhythmic relationship between low and high frequency bands. We have written MATLAB algorithms that implement these ideas.

# 1 Introduction

Music is very complex. Popular music contains many high and low sounds, many different instruments, voice etc. Despite all of this activity, our ears can generally always detect the rhythm in a musical work. Why is it that the rhythm stands out so easily? The beauty of rhythm is that it is *periodic*, that is, it is a pattern that reoccurs at regular intervals. When a repeating pattern is constantly hitting our ears, it stands out more. There are more advantages to rhythm being periodic. Once something is periodic, we can categorize the waveform in terms of *frequency* or *pitch*. *Frequency* is the number of times that a periodic function repeats a sequence of values within a unit of time. *Hertz (Hz)* is a unit for frequency (1 Hz = 1 cycle per second). *Pitch* is the relative high or lowness of audible sound depending on the frequency of the wave produced by the sound. When the frequency of a sound wave is increased, the pitch is increased as well. Similarly, when the frequency of a sound wave is decreased the pitch is decreased. The limits of audible sound are 20 to 20,000 Hz.

Rhythm is a strong pattern of bursts of energy, which typically falls between 0 and 6 Hz. Thus, rhythm is a low frequency component of music. In popular music, we associate the rhythm of a song with the bass drum or the rapping of the snare drum. Imagine a drummer performing a steady drum roll of roughly 2 Hz. Let's assume that our drummer can drum as fast as he or she wants. As the drummer speeds up to roughly 6 Hz, our rhythm begins to sound like vibration. Finally, at about 20 Hz our vibration begins to sound like a solid pitch. This phenomena can be seen on track 1 of the CD. Consider a guitar string that has just been plucked. The string is vibrating well above 20 Hz and thus does not appear as rhythm but rather a solid pitch. Thus, pitch is a high frequency component of music.

When we hear music played in a studio the waveform is *continuous*. That is, the function has no breaks in it and its graph can be traced with a pencil. Even over a finite time interval, the domain of a continuous function contains an infinite number of points. An infinite amount of data cannot fit on a CD or any other digital media. Thus, when the waveform is recorded it must be *sampled*. That is, a discrete set of values is taken from the original waveform at regular time intervals. Thus, sampling converts a continuous function into a *discrete function*, meaning a function whose domain is a discrete set of points. An example of sampling can be seen in Figure 1.

Here we can see that our continuous waveform is sampled at 4 samples per second. When a CD is created, the original waveform is sampled at 44.1 kHz (44,100 Hz). This means that 44,100 sample points are taken per second! The reason for this high number of samples is the Nyquist Theorem, which states that a continuous waveform

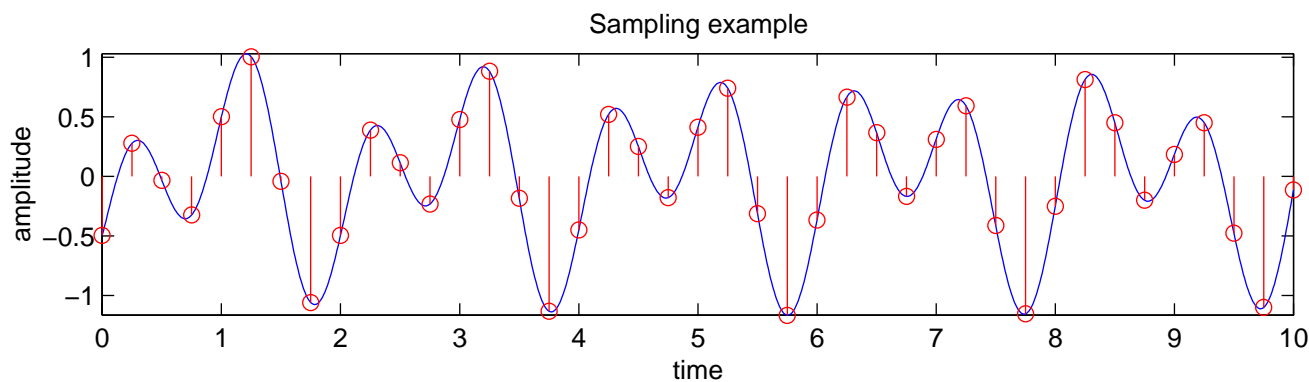


Figure 1: Sampling Example

can be reconstructed from discrete samples as long as its frequency is more than half the sampling rate. Since the limit of audible sound is 20 kHz, we must sample at more than 40 kHz. Once a function is discrete, we can use MATLAB to analyze it—think of the sampled music as a very long vector.

At this sampling rate, a one minute clip contains  $2.646 \times 10^6$  samples! Analyzing this amount of data is a problem for several reasons. First, performing any analysis on this many samples will require expensive computations. That is, the running time of most computations will be high. Secondly, we have established that rhythm is a low frequency component of music. Therefore, much of the high frequency samples need to be filtered out. In order to focus on the rhythm, data reduction is necessary. We began with a data reduction algorithm suggested by Sheirer [3], and improved upon its implementation in MATLAB by Sethares and Staley [2]. Our algorithm separates recorded music into 21 frequency bands, of roughly half an octave each, and extracts the energy in each band. The details of these bands, as well as the algorithm itself, will be discussed later.

Once the data is reduced, we can apply a Fourier Transform. Fourier Analysis simply decomposes the signal into a sum of sinusoids. In the case of our data reduction, the Discrete Fourier transform can identify the primary rhythmic content of each band. Graphing the magnitudes of the Fourier transform coefficients gives us information about particular integer frequencies in the signal. A spike at a particular frequency would imply that either the frequency occurred very periodically or the magnitude of that frequency was high throughout the signal. The Fourier Transform of the unreduced data would also show spikes at periodic high frequencies. This is another reason why data reduction is necessary: rhythm is a periodic, low frequency component and not a periodic, high frequency component. Finally, we compare the DFTs

of each frequency band to determine the time signature of the piece.

## 2 Data Reduction

### 2.1 Overview

Recorded music is very busy! We cannot determine the rhythm directly, other than simply listening to it. There are many unwanted elements in it such as aperiodic high frequencies as well as periodic high frequencies. If we plotted a Fourier transform of the original waveform, the rhythm would become “masked” by other periodic sequences that are not rhythm. Thus, the rhythm cannot be determined by simply looking at the graph of the Fourier transform coefficients. In addition, the amount of data in recorded music is massive. A particular signal is composed of samples or discrete points. We can calculate the number of samples using the following idea:

$$\text{number of samples} = \text{length of song (seconds)} \times \text{samples per second} \quad (1)$$

A 4 minute song sampled at 44.1 kHz contains  $4 \times 60 \times 44,100 = 1.0584 \times 10^7$  samples! Analysis of any sort on this much data requires expensive computations, so data reduction is necessary. We need to filter out this unwanted data so that we can apply Fourier analysis, as well as other techniques, and determine the rhythm. Our data reduction algorithm is based on the methods proposed by Eric D. Scheirer [3] and implemented by William A. Sethares. This algorithm reduces the amount of data found in a particular recorded signal.

Consider an arbitrary signal,  $s$ , sampled at 44.1 kHz residing in wave format on the PC’s hard disk. Since  $s$  is in stereo (2 channels), MATLAB reads it in as a  $n \times 2$  matrix, where  $n$  is the number of samples in the signal. Each column is simply duplicated data for each channel. Therefore, to begin,  $s$  is stripped to a mono signal (1 channel,  $n \times 1$  matrix or a vector of length  $n$ ). The algorithm moves along  $s$  from beginning to end taking *windows* (that is, vectors consisting of some fixed number of points from  $s$ ) of the song. The windows are overlapped slightly so as not to lose data. Let  $nfft$  be the length of the windows and *overlap* be the number of samples that the windows  $w_i$  and  $w_{i+1}$  have in common.<sup>1</sup>

---

<sup>1</sup>Note: In Appendix A.2, the variable “overlap” refers to the number of samples that  $w_i$  and  $w_{i+1}$  have in common (Figure 2). In Appendix A.1, the variable “overlap” refers to the number of samples that  $w_i$  and  $w_{i+1}$  do NOT have in common ( $nfft - \text{overlap}$  samples).

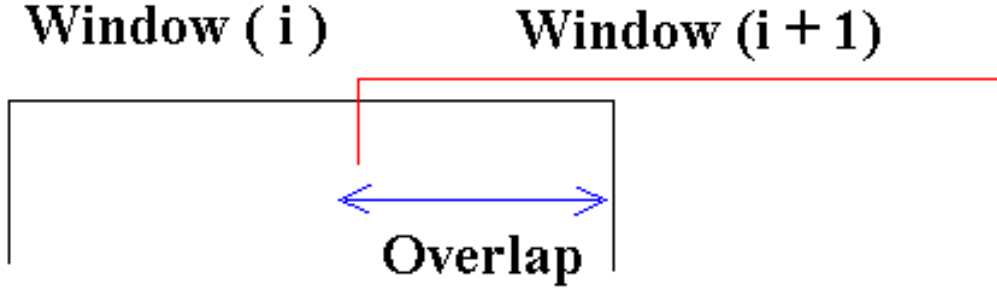


Figure 2: Overlap used when determining the samples in the  $i^{\text{th}}$  window

We can determine the number of windows, given a particular signal length, by the following:

$$\text{number of windows} = \text{floor} \left( \frac{ns - nfft}{nfft - \text{overlap}} \right) \quad (2)$$

where  $ns \in \mathbf{Z}$  is the original signal length.

Figure 3 demonstrates this iterative behavior. Each window contains  $2^{pow}$  samples, where  $pow$  is 9, 10, 11, or 14. For future discussion, let  $2^{pow} = nfft$  and let  $w_i$  denote the  $i^{\text{th}}$  window containing the  $nfft$  samples from  $s$ . In general terms, we can determine the samples from  $s$  contained in the  $i^{\text{th}}$  window by the following:

$$w_i = s[i(nfft - \text{overlap})], \dots, s[i(nfft - \text{overlap}) + nfft - 1] \quad (3)$$

where  $i$  runs from 0 to  $\text{floor}((ns - nfft)/(nfft - \text{overlap}))$ .

It is recommended that we overlap our windows so that we may identify prominent frequencies that were not detected in each window individually. For example, consider a periodic frequency  $f$  within a window  $w_i$ . If the end of this window cut off  $f$  before it could repeat, it may not be detected as prominent.

Figure 4 shows the technique that is applied to each windowed signal. First, an FFT is applied to  $w$ , creating a vector of the same length, namely  $nfft$ . Let's denote this new vector as  $w_{fft}$ . The vector  $w_{fft}$  contains the complex coefficients of the sinusoids, with frequencies 0 to  $nfft - 1$ , which, when summed together, create  $w$ . The details of how these coefficients are generated will be discussed later. The magnitude or strength of frequency  $x$  within  $w$  is  $|w_{fft}[x]|$ . Consider a particular frequency  $0 \leq f < nfft$ . If  $f$  occurs consistently within  $w$  or if  $f$  maintains a high magnitude throughout  $w$ ,  $|w_{fft}[f]|$  will be large. An important property of the FFT is if  $w$  is

real,  $|w_{fft}[x]| = |w_{fft}[nfft - x]|$ . Therefore, since  $w$  is real,  $|w_{fft}|$  is symmetric i.e. only half of the data points are unique. Thus, the algorithm takes  $|w_{fft}|$  and reduces  $w_{fft}$  to  $\frac{nfft}{2}$  samples, denoted  $fftmag$ .

At this point, the algorithm multiplies this reduced  $fftmag$  by a filter matrix<sup>2</sup>. The algorithm chooses the filter matrix based upon  $pow$ . The table below shows the size of the filter matrix used for a given  $pow$ :

| $pow$ | $\frac{nfft}{2}$ | $nbands$ | filter matrix size |
|-------|------------------|----------|--------------------|
| 9     | 256              | 21       | $21 \times 256$    |
| 10    | 512              | 23       | $23 \times 512$    |
| 11    | 1,024            | 23       | $23 \times 1024$   |
| 14    | 8,192            | 29       | $29 \times 8192$   |

Multiplying these two matrices is equivalent to multiplying  $fftmag$  by each row in the filter matrix. This gives us a “prism effect”, as suggested by Figure 4, where each window is split into 21, 23 or 29 frequency bands. When multiplying a  $nbands \times \frac{nfft}{2}$  filter matrix and a  $\frac{nfft}{2} \times 1$  vector  $fftmag$ , we are left with a data-reduced  $nbands \times 1$  windowed signal. Let’s denote this new vector as  $w_{dr}$ .

So what have we done? This new  $w_{dr}$  now contains the magnitude or strength of a *range* of frequencies. The FFT gave us the strengths of every *individual* frequency within  $w_i$ . The filter matrix breaks apart these frequencies into  $nbands$  frequency bands where each band contains a range of frequencies. The first row contains the lowest frequencies while the last row contains the highest frequencies. The filter matrix assures that much of the high frequencies have been filtered out since groups of frequencies are lumped together. The final  $w_{dr}$  contains only  $nbands$  entries.

This process of taking a window,  $w_i$ , of length  $nfft$  continues until  $i = \text{floor}((ns - nfft)/(nfft - overlap))$ . We have established that each reduced  $w_i$  contains  $nbands$  elements. The end result is what we call an *audio matrix* of size  $nbands \times \text{floor}((ns - nfft)/(nfft - overlap))$ . It is important to note that as the overlap increases the number of samples in the audio matrix increases, thus improving the time resolution of the audio matrix. As  $nfft$  increases,  $nbands$  increases and the number of samples in the audio matrix also increases—however, with larger values of  $nfft$ , the algorithm can detect a wider range of frequencies. As in creating a spectrogram, there is always a trade-off between time resolution and frequency resolution.

---

<sup>2</sup>Sethares and Scheirer [2] provided us with various filter matrices

Let's put all of this together. Consider a particular 4 minute signal,  $s$ , sampled at 44.1 kHz with  $pow = 9$  and  $overlap = 212$ . By Equation 1, the number of samples in  $s = 4 \times 60 \times 44,100 = 10,584,000 = ns$ . By Equation 2, we will be taking  $\text{floor}(\frac{10,584,000-512}{512-212}) = 35,278$  windows. Each window contains 21 entries or samples. Thus, our audio matrix is of size  $21 \times 35,278$  and contains 740,838 samples! This a major reduction! We managed to reduce a 4 minute song of  $1.0584 \times 10^7$  samples to an audio matrix of 740,838 samples.

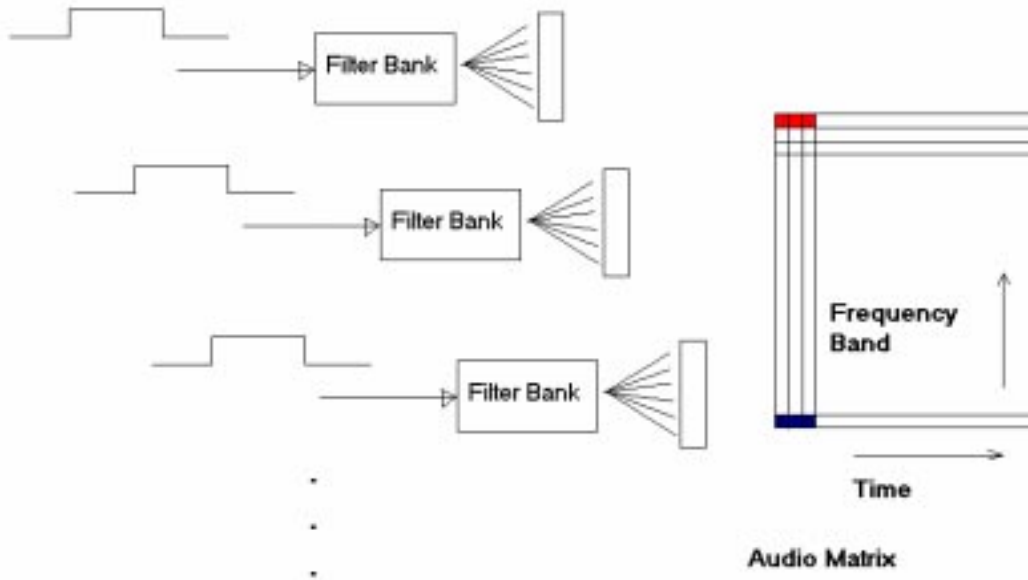


Figure 3: windowing and the creation of the audio matrix

## 2.2 Analysis

MATLAB is known for its quick matrix and vector manipulation. It is often the case that looping can be replaced by using matrices. We determined that it is faster to use matrices and vectors (when possible) to perform tasks. Thus, the same result can be obtained faster by replacing iterative algorithms with matrix manipulation. The old data reduction algorithm (Appendix A.1) uses looping for sliding the window along the signal to build the audio matrix. Since a 48-second song contains millions of samples, thousands of iterations were needed. Thus, the old data reduction algorithm was

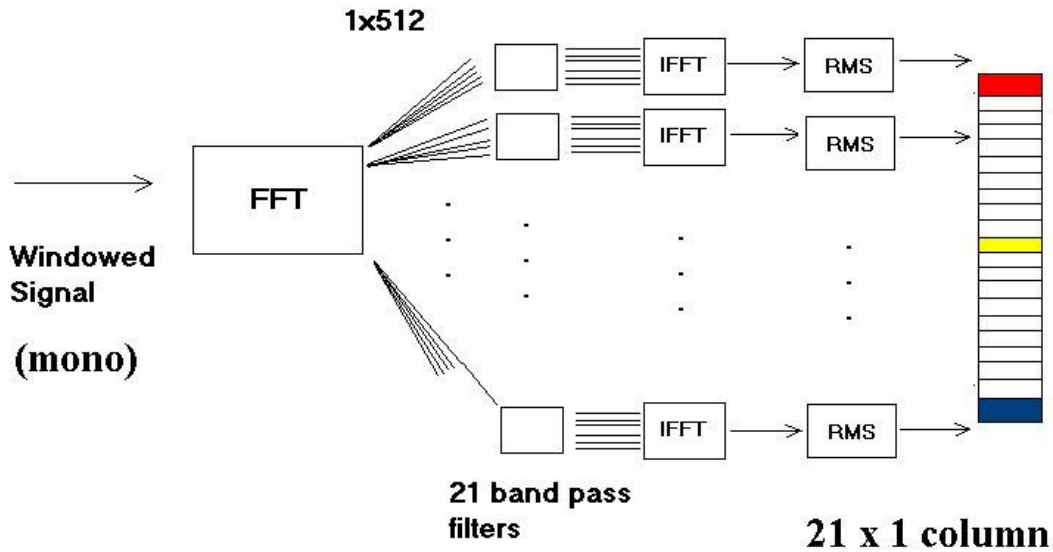


Figure 4: processing of each window

slow. It was intuitive to write the algorithm in this manner since windows are taken iteratively along the signal. However, we created a faster, more efficient algorithm (Appendix A.2) by replacing this iterative behavior with matrix manipulation.

The running time of the algorithms varies depending on the architecture, the speed of the processor, the operating system and the amount of memory in the PC. Both algorithms were run on my PC (AMD 1.4 GHz, Red Hat Linux 7.3, 128 MB RAM) and Dr. Waksman's PC (Intel 2.0 GHz, Windows 98, 1.0 GB RAM). The old data reduction algorithm ran in linear time  $[\theta(n)]$  on both PCs. The function  $\theta(n)$  (order  $n$ ), where  $n$  is the number of samples in the song, describes the behavior of the algorithm as  $n \rightarrow \infty$ . If an algorithm runs in  $cn$  time, where  $c$  is a constant cost, then the algorithm is linear or order  $n$ . If two different algorithms are  $\theta(n)$ , they have the same running time as  $n \rightarrow \infty$ . In reality, however, the number of samples are finite in number. Therefore, the constant cost,  $c$ , is important in the running time.

The new data reduction algorithm ran in linear time as well. However, we can see by Figure 5 that the new data reduction algorithm has a lower constant cost and thus ran much faster. Surprisingly, when running on my PC, the new data reduction

algorithm actually took longer after  $3.8 \times 10^6$  samples! After  $3.8 \times 10^6$  samples, it is evident, by Figure 6, that the algorithm begins to exhibit polynomial behavior  $[\theta(n^2)]$ . At first, this result may seem odd since this behavior is not exhibited on Dr. Waksman's PC. MATLAB automatically attempts to store matrices in memory. If the matrix is too large, it is stored on hard disk. Obviously, there is more overhead, and thus a higher data seek time, when accessing data on the hard disk. Since the old data reduction algorithm examines a small window of a signal at a time, it is possible to store the window in memory for calculations. However, the new data reduction algorithm attempts to store the entire signal in memory. When approaching this many samples, my computer was no longer able to completely store the entire signal in memory. Thus, data access time became more expensive and took longer than the old data reduction algorithm. Dr. Waksman's PC was successfully able to store the entire signal in memory, so this polynomial behavior never surfaced.

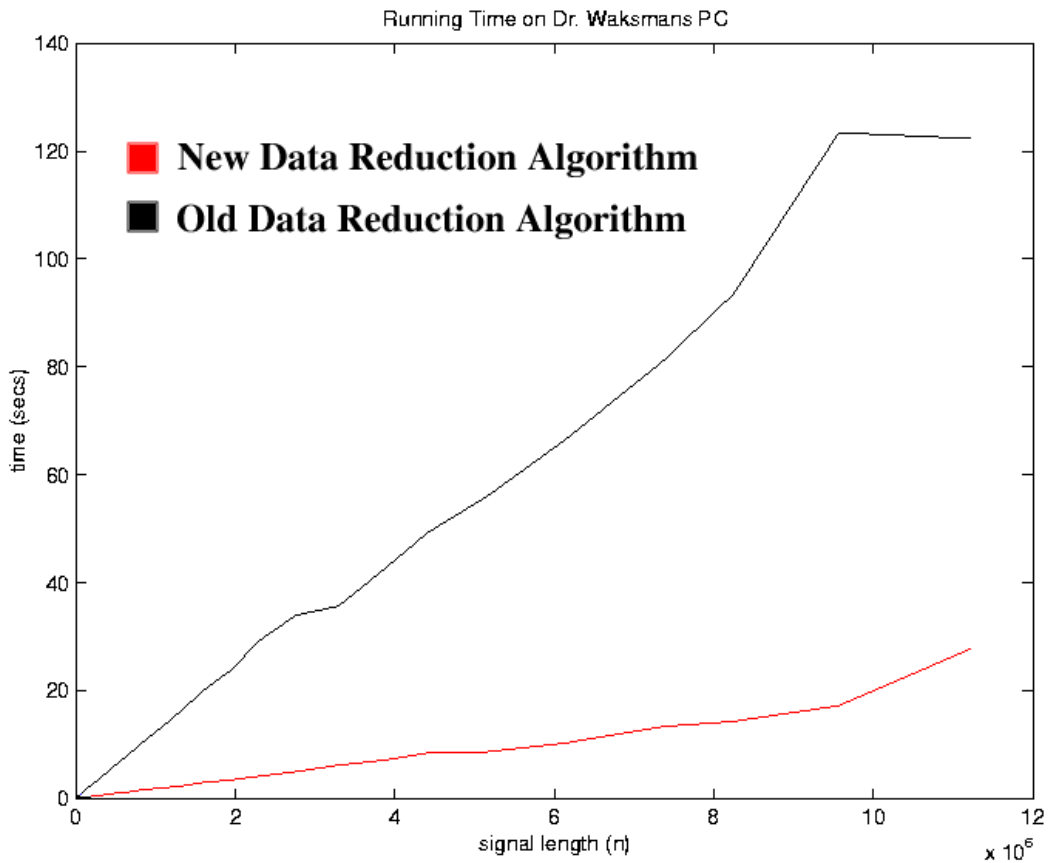


Figure 5: Running time of data reduction algorithms on Dr. Waksman's PC

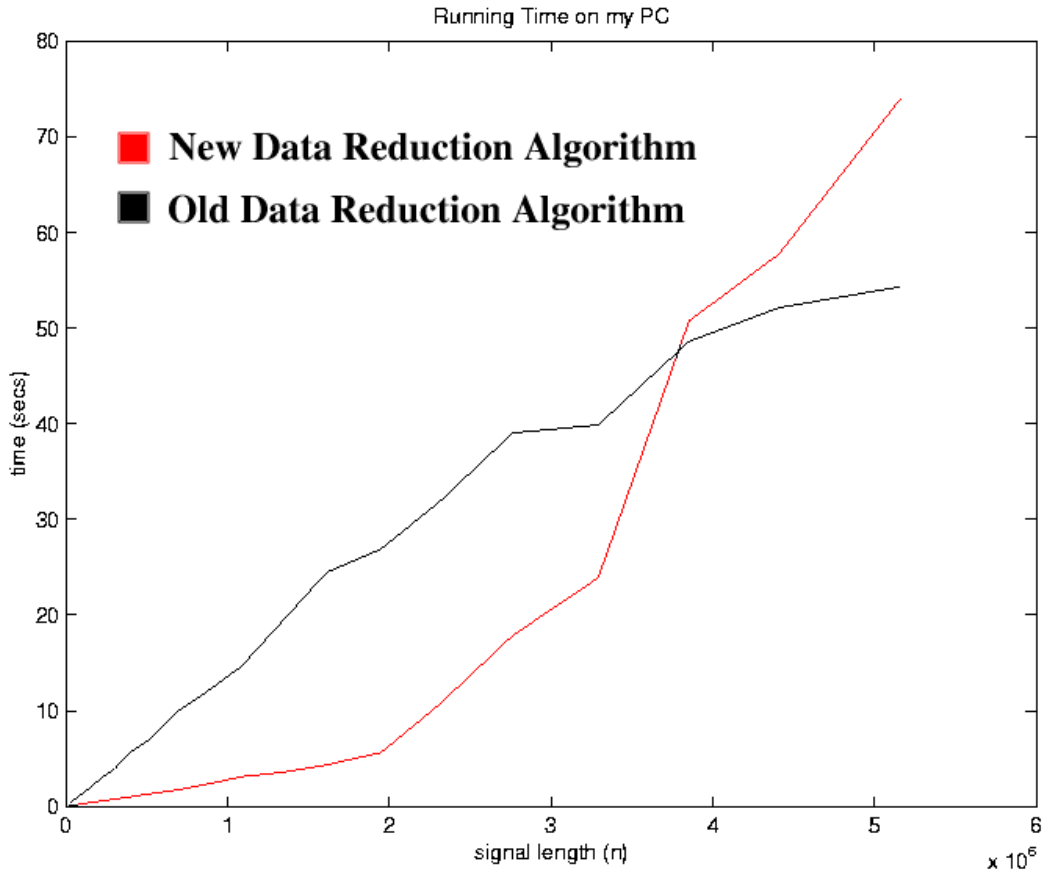


Figure 6: Running time of data reduction algorithms on my PC

### 2.3 Example

Popular music is a prime candidate for data reduction and analysis. Generally, popular music maintains a steady rhythm throughout the entire song. The fact that the rhythm is maintained is important (the reason for this will be discussed later). ZZ Top's *Sharp Dressed Man* has a very periodic, heavy rhythm throughout the song. Thus, I felt that this would be a good choice for analysis. The prominent beat of the song is introduced immediately when the song begins. Therefore, in this example, I felt that the first 7 seconds would suffice. Figure 7 shows a plot of this 7-second excerpt. As you can see, the waveform of the original signal is very complex. This waveform contains  $7 \times 44,100 = 308,700$  samples. So, we need to reduce the amount of data. Since we are examining such a short clip, choosing  $pow = 9$  ( $nfft = 512$ ) is

sufficient. An *overlap* = 212 was used to reduce the data. We have constructed an audio matrix of size  $21 \times 1027$  containing 21,567 samples! This is quite a reduction.

Figure 8 is an image of the audio matrix that was created for this example. The *y*-axis corresponds to the 21 rows or 21 frequency bands of the audio matrix. In this case, each frequency band spans roughly half an octave. The first band contains the magnitude or energy of the lowest frequencies from the original waveform. The 21<sup>st</sup> frequency band holds the magnitude of the highest frequencies. The *x*-axis corresponds to the 1,027 columns of the audio matrix. We have established that the audio matrix preserves time. Therefore, the *x*-axis indicates the energy of the various frequency bands as the song progresses from 0 to 7 seconds. The image is color-coded in rainbow order depending on the magnitude or energy of a particular frequency band. The color red exhibits a frequency band of high energy while yellow implies a frequency band of mild energy. Blue indicates a frequency band of low energy.

We have established that this data reduction algorithm successfully reduces the number of samples. However, how does it hold up when preserving periodicity in the original waveform? Let's dissect the image. First, the graph has red spikes in the first and second frequency bands. Since the spikes are equidistant, this implies that there exists a low frequency, *periodic* burst of high energy that is maintained throughout the 7 second clip. This certainly seems to relate to the rhythm of the song. The burst of energy occurs roughly twice a second. This seems reasonable for rhythm emanating from a bass drum. There is more activity at the top of Figure 8 in the 18<sup>th</sup> through 21<sup>st</sup> frequency bands. Since these yellow spikes are also equidistant, this implies that there exists a high frequency, periodic burst of mild energy that is maintained throughout the 7 second clip. This burst occurs twice for every red spike or 4 times a second. This could possibly be the hi-hat cymbals, which are high pitched and thus high frequency. Finally, the image reveals a startling result. Between 3 and 4.5 seconds, we notice a yellow "blob" roughly in the center of the graph. In the original song, the singer makes a brief sound into the microphone after roughly 3 seconds. Therefore, this "blob" is actually the singer's voice signature for that particular noise. We can see that voice is extremely complex as opposed to rhythm.

So far, we are making speculations based on the graph. If we could physically listen to the audio matrix, then we could verify these speculations. Thus, we wrote an algorithm (Appendix A.5) to output the audio matrix into a wave file using filtered white noise to fill in the rhythm bands. The details of how this was done will be explained soon. The results were interesting. The rhythm was preserved almost perfectly. In addition, the voice was somewhat preserved! We did indeed hear the singer's voice after roughly 3 seconds! Other elements of the original song, such as

the guitar, were lost almost completely.

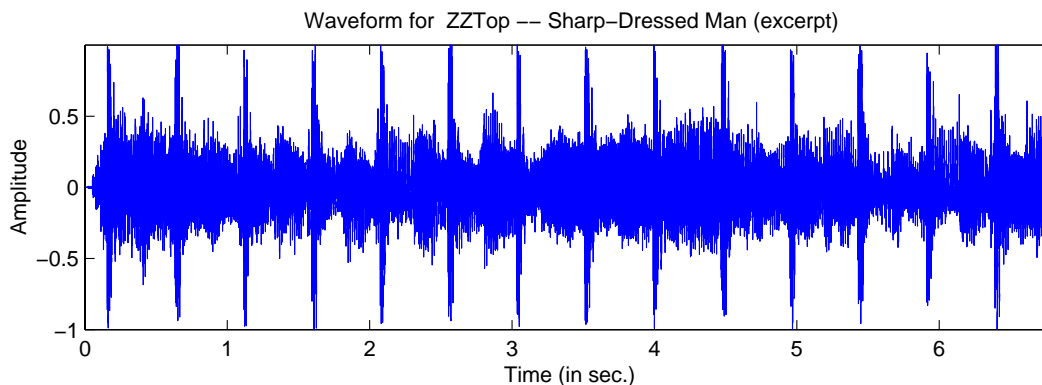


Figure 7: Original waveform for ZZ Top - “Sharp Dressed Man”

## 2.4 Listening to the audio matrix

We can obviously listen to the original signal [track 2 on the CD]. How can we listen to the data reduced signal, i.e. the audio matrix? Since the audio matrix preserves time, it is possible. The algorithm in Appendix A.5 is written to take the rows of the audio matrix and output audible tracks. However, there are several problems with playing the audio matrix verbatim.

First, the audio matrix contains magnitudes well above 1. As we can see by Figure 7, the sinusoidal waves of the original track go between  $\pm 1$ . Thus, the magnitudes of the audio matrix need to be scaled to run between  $\pm 1$ . If we simply divide every value by the maximum magnitude, we will be left with values running between  $\pm 1$ .

Secondly, Figure 9 points out another problem. We can see that with an audio matrix of size  $21 \times 1027$ , each track or row has 1,027 samples. By Equation 1, this implies that if we created a wavefile for each track at 44.1 kHz, the wavefile will be .02 seconds! This is a problem considering the original track was 7 seconds. Thus, we need to resample. That is, we need to figure out how many samples we need in the wavefile for every one sample in the audio matrix. In mathematical terms, we need to determine  $x$  such that:

$$\frac{x \times 1027 \text{ samples in audio matrix}}{44100 \text{ samples per second}} = 7 \text{ seconds}$$

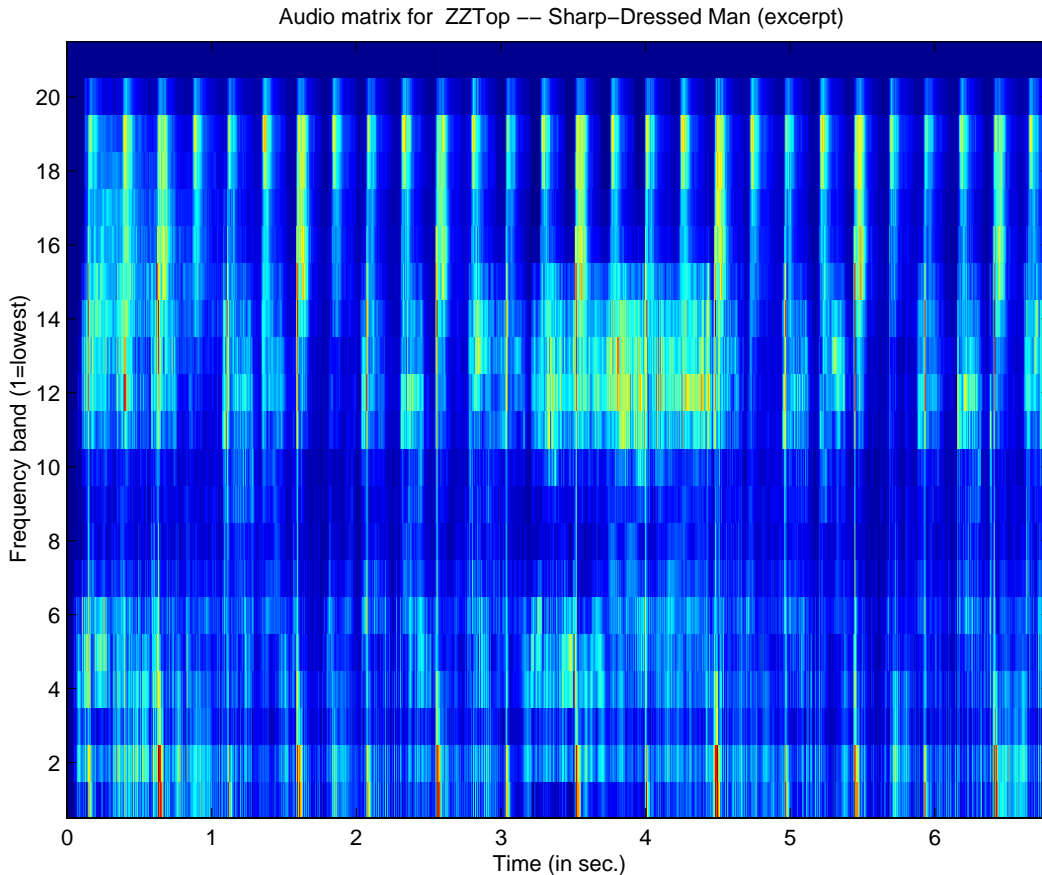


Figure 8: Image of the audio matrix for ZZ Top - “Sharp Dressed Man”

In this case,  $x$  is approximately 301 samples in the waveform for every one sample in the audio matrix. This resampling should stretch the waveform to 7 seconds. Figure 9 and Figure 10 were created by this process since these plots are of a 7 second waveform. Figure 9 is a plot of all of the rows of the audio matrix summed together and scaled, together with the negative of the sum vector. We can see that the resulting waveform is hollow or simply an *envelope*. If we played the waveform in Figure 9, we would hear nothing. We can resolve this problem by filling in the envelope with random values i.e. white noise. The result is an audible waveform as seen in Figure 10. However, listening to this waveform [track 3 on the CD] doesn’t give much idea of the rhythmic structure of the piece, since all the high frequency information has been lost.

The algorithm in Appendix A.5 solves all of the above problems. It creates 21 tracks or wavefiles [tracks 4 thru 24 on the CD] (for each row in the audio matrix) of the

same length as the original song. Track 1 corresponds to the lowest frequency band in the audio matrix while track 21 corresponds to the highest frequency band. It is difficult to determine any rhythmic patterns in any one individual track. So, the algorithm first creates a track for each band by filling in with filtered white noise of the appropriate frequency range. Then all of the tracks are combined. It is in this track [track 25 on the CD], that we notice that the rhythm is preserved almost perfectly and that the voice of the singer is somewhat preserved. The song is quite noticeable in this track. We can actually sing along if we wanted to! It is important to note that each track will have a hissing sound throughout it. Remember, this is because the envelopes were filled in with white noise so that we can hear the sound.

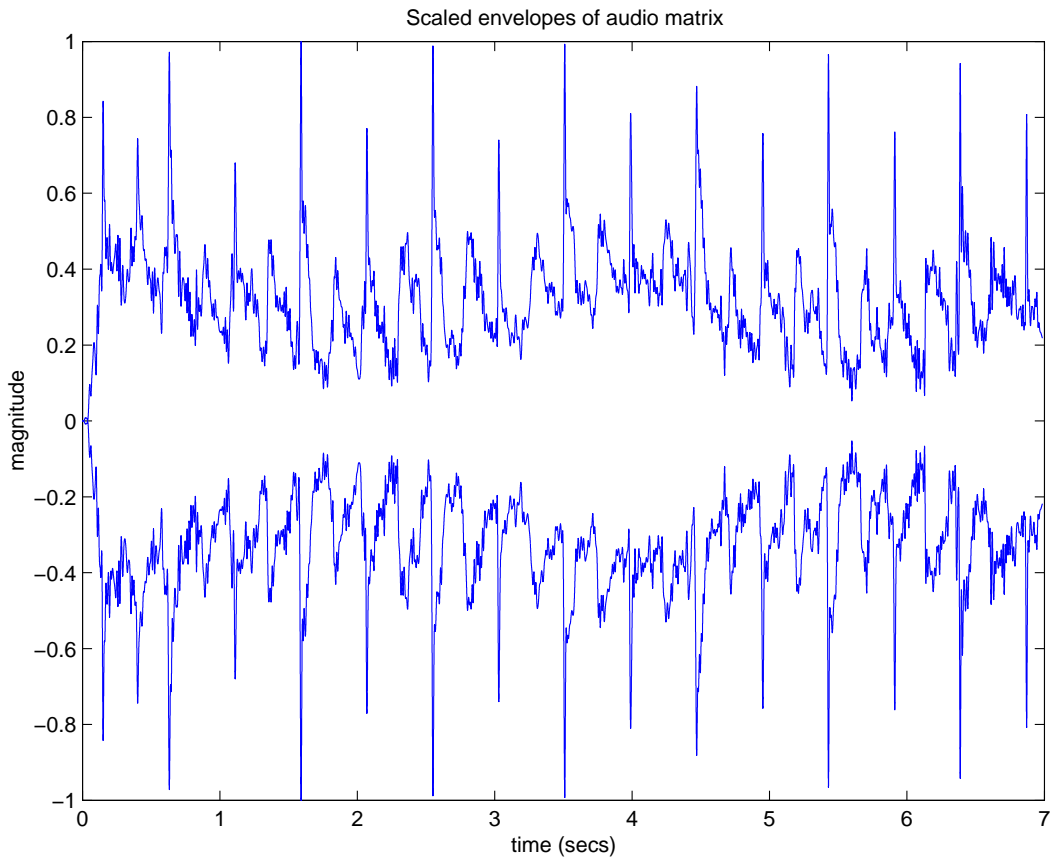


Figure 9: Scaled envelopes of the audio matrix

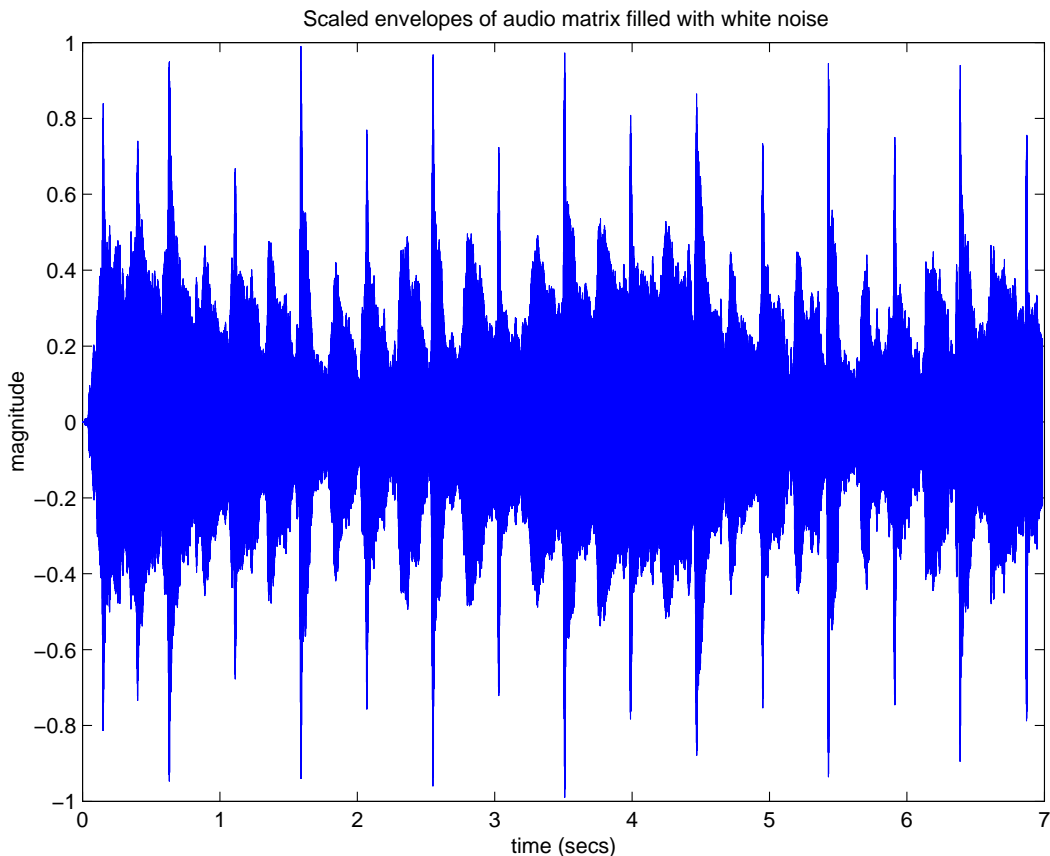


Figure 10: Scaled audio matrix envelopes filled with white noise

## 2.5 Conclusion

The data reduction algorithm is good! Not only was the primary rhythmic content preserved, but in the case of ZZ Top's *Sharp Dressed Man* the voice was preserved! We managed to identify the rhythm with a much smaller amount of data.

If  $pow = 9$ , the frequency resolution allows only 21 bands, while 29 bands are possible if  $pow = 14$ ; but the time resolution is smaller with  $pow = 14$ . Also, the audio matrix is smaller if  $pow = 9$ . However, even with a larger power the data is dramatically reduced. Also, in the new data reduction algorithm, an  $overlap = 212$  seems to work well. This value was determined experimentally.

The CD contains various songs that were data reduced and analyzed. Since a CD is limited to 99 tracks, two songs are listed with their corresponding frequency bands.

Tracks 51 thru 66 consist of an original 04 and the sum of all of its frequency bands applied to white noise.

## 3 The Discrete Fourier Transform (DFT)

### 3.1 Overview

The Discrete Fourier transform creates a vector  $X[k]$  of complex numbers from a vector  $x[n]$  of period  $N$  using the following formulas:

$$\begin{aligned} \text{DFT:} \quad X[k] &= \sum_{n=0}^{N-1} x[n] e^{-w} \\ \text{Inverse DFT:} \quad x[n] &= \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^w \\ &\text{where } n, k = 0 \dots (N-1) \in \mathbf{Z} \text{ and } w = \frac{2\pi jnk}{N} \text{ (where } j = \sqrt{-1}) \end{aligned} \tag{4}$$

It appears that the Inverse DFT does not resemble a sinusoid. However, Euler's formula tells us that  $e^{ja} = \cos a + j \sin a$ . It is important to note that  $w$  is complex in the DFT and Inverse DFT. Since  $x[n]$  is real,  $X[k]$  is complex. Consider a discrete signal  $x$  of length  $N$ . Upon closer examination of the Inverse DFT, Euler's formula tells us that we can represent  $x$  as a finite sum of sinusoids (note that if  $x[n]$  is real, the  $j \sin w$  terms must cancel)! So, in fact, every periodic signal is actually *composed* of many sinusoids! Well, if this is the case, then what do all of the variables in the formula correspond to?  $N$  is simply the length of our discrete signal while  $n$  is an index into the signal. That is,  $x[0]$  is the first element in the signal and in general,  $x[n-1]$  is the  $n^{\text{th}}$  element in the signal.  $X[k]$  is frequency spectrum for the original signal. When looking at the Inverse DFT, we can see that  $X[k]$  specifies the amplitude for the sinusoid with frequency  $k$ . That is,  $|X[k]|$  tells us how much of a sinusoid with frequency  $k$  that we need to represent the original signal.

When plotting a discrete signal, the graph is a amplitude vs. time plot. However, when plotting  $|X[k]|$ , the graph is a magnitude vs. time graph. Therefore, when applying a Fourier transform, we gain a frequency spectrum of the signal but lose the concept of time.

The utility of the DFT as well as these phenomena will become more apparent in the following examples.

## 3.2 Examples

### 3.2.1 Sample Problem

Consider a discrete signal,  $x = \{0, 1, 1, 1, 0, 0, 1, 1, 1, 0\}$  represented by Figure 11. [Note that this sequence has frequency 2]

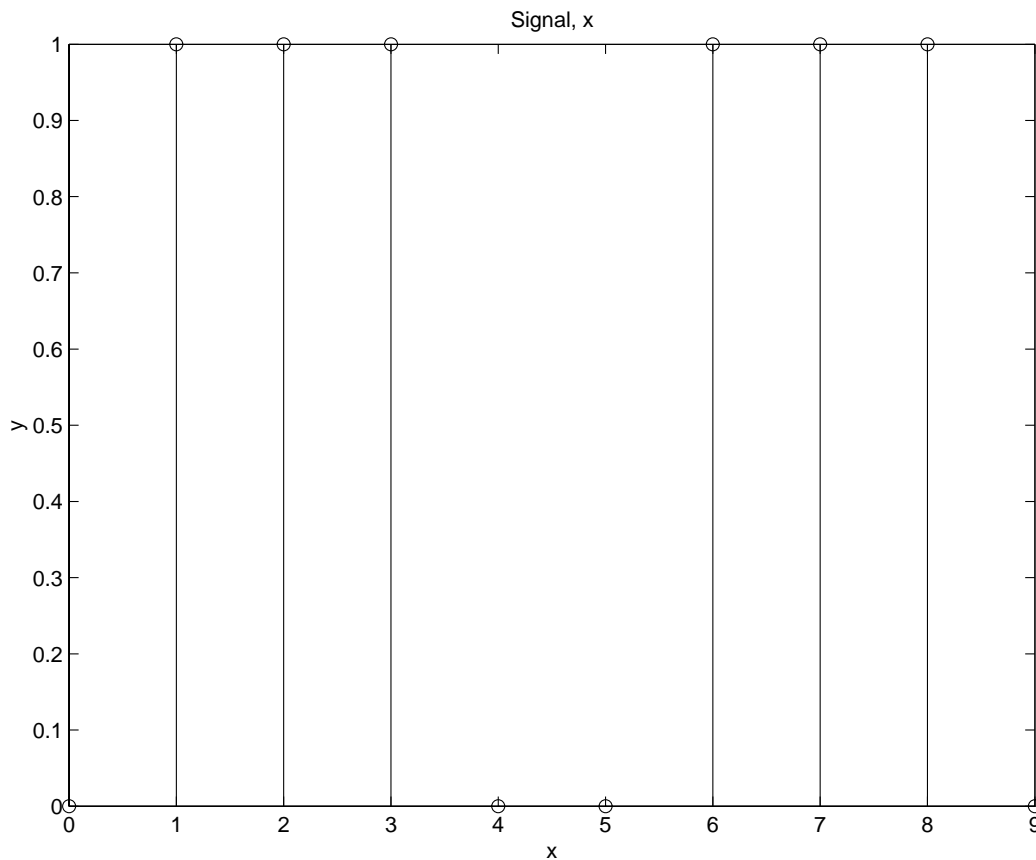


Figure 11: Square Wave Example

The DFT formula in this case looks like:

$$X[k] = \sum_{n=0}^9 x[n] e^{-\frac{2\pi jnk}{10}} \text{ where } k = 0, 1, \dots, 9$$

We can use Euler's formula ( $e^{jx} = \cos x + j \sin x$ ) to rewrite  $e^{-\frac{2\pi jnk}{10}}$ :

$$X[k] = \sum_{n=0}^9 x[n] [\cos(\frac{\pi nk}{5}) + j \sin(\frac{\pi nk}{5})]$$

Thus:

$$\begin{array}{ll} X[0] = 6 & \\ X[1] = 0 & \\ X[2] = -2.618 - 1.9021j & |X[0]| = 6 \\ X[3] = 0 & |X[1]| = |X[9]| = 0 \\ X[4] = -0.382 - 1.1756j & |X[2]| = |X[8]| = 3.2361 \\ X[5] = 0 & |X[3]| = |X[7]| = 0 \\ X[6] = -0.382 + 1.1756j & |X[4]| = |X[6]| = 1.2361 \\ X[7] = 0 & |X[5]| = 0 \\ X[8] = -2.618 + 1.9021j & \\ X[9] = 0 & \end{array}$$

When applying the inverse DFT to these complex coefficients we will recover our original signal,  $x$ . We can see that the prominent magnitude corresponds to  $X[2]$  or a frequency of 2. This implies that we need more of a sinusoidal wave with frequency 2 in order to represent the original signal  $x$ . When finding the prominent frequency, ignore  $X[0]$ , because it is simply the average of the elements of  $x$ .

When looking at Figure 12, we notice that the magnitudes of the frequencies are symmetric. The DFT always has this symmetric property. This is because, if  $x[n]$  is real,  $X[k] = X[-k]^* = X[N - k]^*$ .  $x[n]$  is real in this case, so the magnitudes are symmetric. Also, notice that only multiples of 2 have nonzero coefficients.

The vector  $X[k]$  is of the same length as our original signal  $x[n]$ . In order to create  $X[k]$ , the DFT takes  $N$  calculations for each of the  $N$  elements in  $X$ . Therefore, the time complexity of this formula is  $\theta(n^2)$ . This isn't very efficient. The *Fast Fourier transform (FFT)* generates the same values as the DFT but much faster. In fact, the FFT is a  $\theta(n \log n)$  algorithm! The FFT has many uses. For example, it can be used to multiply two polynomials together quickly. It is also used in wide variety of fields ranging from mathematics to computer science to quantum mechanics. But, how can we use Fourier Analysis to aid us in detecting rhythm?

We can see by Figure 7 that music has a more complex waveform that is not perfectly periodic. However, in the case of most popular music, the track is "periodic enough" to be analyzed by the DFT. This is because popular music is recorded to a metronome track that keeps the drummer perfectly in time. The DFT is typically

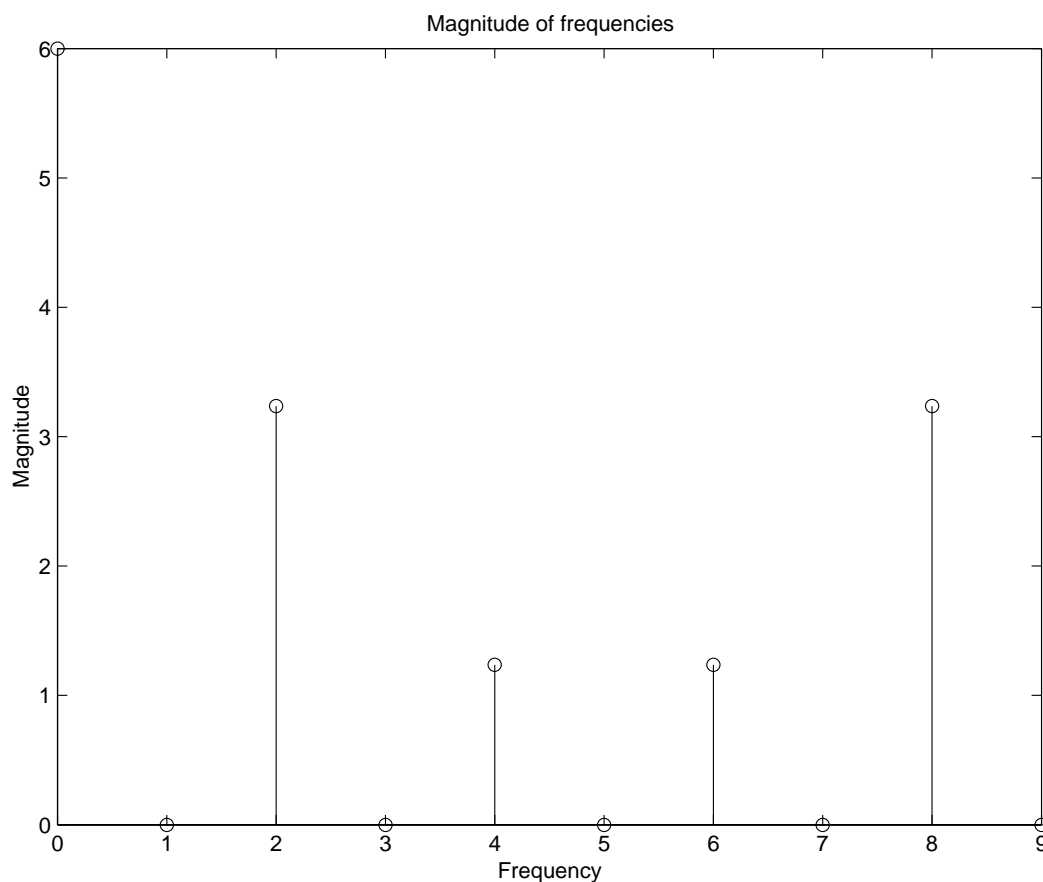


Figure 12: Frequency magnitudes of our signal,  $x$

applied to periodic signals, but in practice, we can still gain relevant information from approximately periodic signals.

### 3.2.2 Analysis of ZZ Top after data reduction

How can we apply Fourier Analysis to our data reduced audio matrix? When looking at Figure 8, we notice that we can only guess how often the particular low frequency, burst of high energy occurs. If we apply the FFT to the rows of the audio matrix, we can determine the frequency of these periodic bursts of energy. Therefore, we can determine the beat of the song and verify that those red spikes in Figure 8 are indeed the rhythm. Since we can listen to the audio matrix and hear the rhythm, we know it exists in the data reduced audio matrix.

Figure 13 is a plot of the magnitudes of the FFTs of each row of the audio matrix superimposed. It was generated using the algorithm in Appendix A.3. The various colors correspond to the frequency bands of the audio matrix. The color red relates to the highest frequency band (21, treble) and yellow to a mid frequency band (11, midrange). Blue corresponds to the lowest frequency band (1, bass). The height of a spike in the graph relates to the strength or magnitude of a periodic frequency band. The  $x$ -axis relates to the frequency of each band, i.e., how many times that beat occurs within the song.

Now that we know how the graph is designed, let's analyze it. We can see that the prominent spike is high frequency because it is red. Using Equation 1, we can relate a prominent frequency spike to the number of times a minute that this burst of energy occurs. We can rewrite Equation 1, to obtain:

$$\frac{\text{number of samples (ns)}}{\text{number of samples per second}} = \text{the length of the song in seconds}$$

Thus:

$$\begin{aligned} \text{beats per minute} &= 60 \times \text{frequency} / \left( \frac{\text{number of samples (ns)}}{\text{number of samples per second}} \right) \\ &= \frac{60 \times \text{frequency} \times \text{number of samples per second}}{\text{number of samples (ns)}} \end{aligned} \tag{5}$$

Since the original signal was a CD track, it was sampled at 44.1 kHz. In this example, the original signal contained 2,154,391 samples. Therefore, the prominent frequency 203 occurs:

$$\frac{60 \times 203 \times 44,100}{2,154,391} = 249.3 \text{ times a minute or } 4.15 \text{ times a second.}$$

This enforces what we saw in Figure 8. We saw a periodic high energy, high frequency component occur roughly 4 times a second. Therefore, this prominent spike in the DFT graph is the hi-hat cymbals. This is not the rhythm of the song. When we

listen to the song, we tap our foot according to the beat which, in this song, was generated by the bass drum. The bass drum is not high frequency. Thus, the blue, low frequency spike at around 101 beats per minute seems to be the best candidate for the primary beat. The frequency of this spike is half that of the prominent spike; that is, this prominent burst of energy occurs twice for every one low frequency burst. This low frequency burst occurs roughly twice a second since the prominent frequency occurred roughly 4 times a second. This blue spike appears to be the periodic, low frequency burst of high energy seen in Figure 8. We have already established that rhythm falls between 0 and 6 Hz. Therefore, two beats per second (or 2 Hz) is a perfect frequency for rhythm!

As mentioned earlier, it is important that the rhythm is maintained throughout the song. Rhythm, in terms of the audio matrix, is a *recurring* periodic burst of energy. The FFT detects periodicity i.e. reports on the frequency of a particular burst of energy. Classical music typically exhibits frequent changes in rhythm and tempo. Therefore, a graph of the Fourier analysis will not point to any particular frequency. As far as the FFT is concerned, a large single hit of the bass drum does not necessarily constitute a spike or high energy. As mentioned earlier, a spike is caused by a frequent periodic element in the music or an element that is not so periodic but maintains high magnitude. Therefore, a graph of a piece whose tempo changes often would not have any noticeable spikes.

## 4 Conclusion

We have discussed a few methods for quickly and efficiently detecting rhythm. Not only do our algorithms detect the primary beat, but they also give clues about the time signature, which is characterized by the relationship between the different strengths of pulses and the relationship between the bass and the treble. In popular music, we are able to detect the time signature of a particular work.

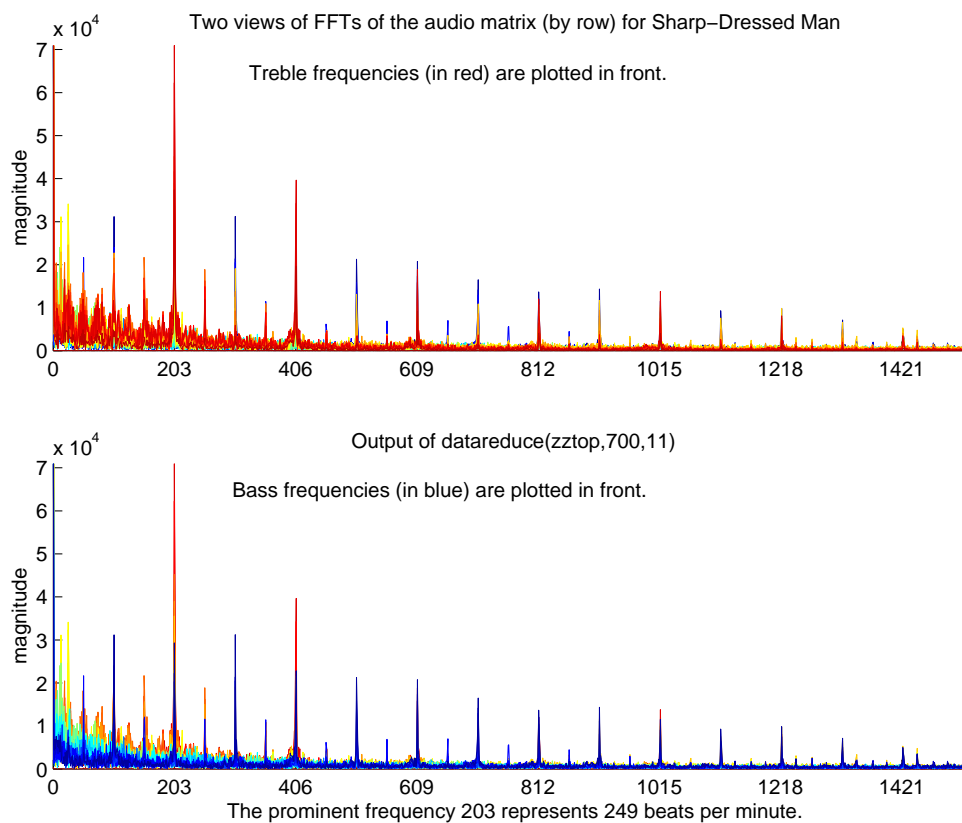


Figure 13: DFT of data reduced ZZ Top - “Sharp Dressed Man”

## References

- [1] William A. Sethares and Thomas W. Staley. “Periodicity Transforms.” *Transactions on Signal Processing*, vol. 47, no. 11, November 1999.
- [2] William A. Sethares and Thomas W. Staley. “Meter and Periodicity in Musical Performance.” preprint, 2001.
- [3] Eric D. Scheirer. “Tempo and Beat Analysis of Acoustic Musical Signals.” *Journal of the Acoustical Society of America*, vol. 103, no. 1, January 1998.

# A Algorithms

## A.1 Original Data Reduction Algorithm

```
function datareduce(infile,overlap,pow)

%DATAREDUCE  Makes audiomatrix for infile
%  DATAREDUCE(INFILE,OVERLAP,POW)
%
%  infile=name of .wav or .au file to process
%  overlap= number of samples till next fft
%  pow = exponent:  uses 2^pow samples to compute the FFT.
%                The values are 9, 10, 11, or 14.
%
%
%  Program by William Sethares (7/1/99)
%  Modified by Rachel Hall (5/7/02)
%  Modified by Hall, Waksman, Flannick (10/25/02)

if nargin==0, error('You must specify a file name. '), end
if nargin < 3, pow=9; end
if nargin < 2, overlap=300; end

[path,infile,ext]=fileparts(infile);
if isempty(path), path='/home/rhall/music/soundfiles/'; end
if isempty(ext), ext='.wav'; end
if (length(ext) ~= 3) & (length(ext) ~= 4)
    error('Soundfiles must be in .au or .wav format. '), end
if length(ext) == 3; if ext ~= '.au';
    error('Soundfiles must be in .au or .wav format. '),
else    au=1; end
end

if length(ext) == 4; if (ext ~= '.wav') & (ext ~= '.WAV') ;
    error('Soundfiles must be in .au or .wav format. '),
else    au=0; end
end
```

```

PLOT = 1; % plot define

if (PLOT)
    clf
end
pathin=path; % input path (directory)
pathout=path; % input path (directory)
nfft=2^pow; % length of FFT

% output file name
outfile=[pathout, '/', infile, num2str(overlap), '.mat'];

% fullfile = input file name.
fullfile=[pathin, '/', infile, ext]

% It has 2 rows (stereo).
% selects the appropriate size matrix (filtmat) that will
% separate out frequency bands.
earfile=['/home/joe/research/ear', num2str(pow), '.mat'] ;

if au==1 [dummy,fs]=auread(fullfile,1);
        siz=auread(fullfile,'size');
else [dummy,fs]=wavread(fullfile,1);
        siz=wavread(fullfile,'size');
end

lengthfile=siz(1); duration=lengthfile/fs;

% iterations = the number of points in the sample
% after data reduction
iterations=floor((lengthfile-nfft)/overlap)

comm=['load ', earfile];
eval(comm);

[rf,cf]=size(filtmat);
audmat=zeros([rf,iterations]); % audmat contains the output
window=hanning(nfft); % hanning window vector of

```

```

                                % size nfft
for j=1:iterations
    if au==1
        ss=auread(fullfile,[round((j-1)*overlap+1)
round((j-1)*overlap)+nfft])';
    else
        ss=wavread(fullfile,[round((j-1)*overlap+1)
round((j-1)*overlap)+nfft])';
    end
                                % ss = nfft samples from the
                                % original file
    ss=ss(1,:);                    % consider only one channel
                                % of the stereo input
    s=window.*ss;                  % apply the window to ss,
                                % producing s
    ffts=fft(s);                   % ffts = the FFT of s
    fft2=ffts(1:nfft/2);           % fft2 = the first half of
                                % the FFT of s
    fftmag=abs(fft2)';             % fftmag = the magnitudes of
                                % the entries of fft2
    thiscol=filtmat*fftmag;         % thiscol is the result of
                                % multiplying filtmat by
                                % fftmag. The data is now
                                % separated into bands
                                % of about 1/3 octave.
                                % The entry in each row is
                                % the energy in that band
                                % over the duration of
                                % nfft points.
    audmat(:,j)=thiscol;           % thiscol is assigned to the
                                % current column of audmat.
    if round(j/1000)==j/1000      % the program prints out
        disp(j)                    % every 1000th iteration
    end

end

effsr=fs/overlap;                 % effsr is the effective
                                % sampling rate

```

```

[raud,caud]=size(audmat);
lenpiece=round(caud/2);           % lenpiece is half the
                                   % number of columns in audmat

% effssf is a vector indicating the frequencies
effssf=0.5*effsr*(0:lenpiece)/lenpiece;

% dftaud will contain the magnitudes of the FFTs
% of the rows of audmat
dftaud=zeros([rf,lenpiece+1]);

for j=1:raud
    ww=audmat(j, :);
    wwfft=fft(ww);
    abswwfft=abs(wwfft(1:lenpiece+1));
    dftaud(j, :)=abswwfft;
end

if (PLOT)

    % plots the columns of dftaud, superimposed.

    plot(0:lenpiece,dftaud')
    axis([0 lenpiece 0 max(max(dftaud(:,3:end)))'])
    [peak,indx]=max(max(dftaud(:,floor(3*duration):end)));
    indx=indx+floor(3*duration)-2;
    xtick=indx*(0:floor(lenpiece/indx));
    set(gca,'XTick',xtick);
    title(['DATAREDUCE('' infile '' , ' num2str(overlap) ...
           ', ' num2str(pow) '' ' ' date]
    xlabel(['The prominent frequency ' num2str(indx) ' represents ' ...
           num2str(60*indx/duration) ' beats per minute.'])
    ylabel(['magnitude'])

end

comm=['save ',outfile];           %save the environment's variables
eval(comm);

```



## A.2 New Data Reduction Algorithm

```
function datareduce(infile,overlap,pow)

%DATAREDUCE  Makes audiomatrix for infile
%  DATAREDUCE(INFILE,OVERLAP,POW)
%
%  infile=name of .wav or .au file to process
%  overlap= the overlap of the sliding window
%  pow = exponent:  uses 2^pow samples to compute the FFT.
%                The values are 9, 10, 11, or 14.
%
%  This program applies filters generated by William Sethares
%  This program utilizes the quick matrix manipulation of {\sc Matlab}
%  instead of for loops
%
%  a file called <infile><overlap>.mat is created in the same
%  directory as the input filename. It contains all of the
%  environment variables created and used in this program INCLUDING
%  the audio matrix
%
%
%  Written By Joseph E. Flannick
%              Rachel Hall
%              Adlai Waksman

% ----- PROCESS THE PROGRAM ARGUMENTS -----
if nargin==0, error('You must specify a file name. '), end
if nargin < 3, pow=9; end
if nargin < 2, overlap=212; end % = 2^9 - 300

[path,infile,ext]=fileparts(infile);
if isempty(path), path='/home/joe/research'; end
if isempty(ext), ext='.wav'; end

if (length(ext) ~= 3 & length(ext) ~= 4)
    error('Soundfiles must be in .au or .wav format. '), end
```

```

PLOT = 1;          % plot define

au = 0;
if length(ext) == 3
    if (ext~='.au') & (ext~='.AU')
        error('Soundfiles must be in .au or .wav format. '),
    else    au = 1; end
end

if length(ext) == 4 & (ext ~= '.wav') & (ext ~= '.WAV') ;
    error('Soundfiles must be in .au or .wav format. '),
end

% ----- READ IN OUR FILES AND SET UP -----

% output file name
outfile=[path,'/',infile,num2str(overlap),'.mat'];

% fullfile = input file name. It has 2 rows (stereo).
fullfile=[path,'/',infile, ext] ;

% selects the appropriate size matrix (filtmat)
% that will separate out frequency bands.
earfile=['/home/joe/research/ear',num2str(pow),'.mat'];

comm=['load ',earfile];          % Load the earfile
eval(comm);

if au==1
    [signal, fs]=auread(fullfile); % read the input .au file
else    [signal, fs]=wavread(fullfile); % read the input .wav file
end

clear au fullfile                % relieve memory usage!!
signal = signal(:, 1);           % strip the signal to 1 channel
nfft=2^pow;                       % length of FFT
ns = length(signal);              % length of the signal

```

```

% number of cols in audmat, sigmatrix etc
ncol = floor ((ns-nfft) / (nfft - overlap));

sigmatrix = zeros(nfft, ncol);           % create our signal matrix
colindex = 1 + (0:(ncol-1)) * (nfft - overlap); % set up the column indexing
rowindex = (1:nfft)';                   % set up the row indexing
if ns < (nfft + colindex(ncol)-1)
    signal(nfft+colindex(ncol)-1) = 0;    % zero-pad the signal
end

window=hanning(nfft);                   % hanning window vector of size nfft

% ----- CREATE THE AUDIO MATRIX -----

% shove the signal into a matrix
sigmatrix(:) = signal(rowindex(:, ones(1, ncol)) + colindex(ones(nfft,
1), :) - 1);

clear rowindex colindex signal          % relieve memory usage!!
winmatrix = window(:, ones(1, ncol));    % create a window matrix so we
can multiply
clear window                             % relieve memory usage!!

% take the fft across the 1st dimension
fftmat = fft(winmatrix.*sigmatrix, [], 1);
clear sigmatrix winmatrix                % relieve memory usage!!

% get rid of 1/2 the rows b/c they are symmetric
fftmat = fftmat(1:(nfft/2), :);
fftmag = abs(fftmat);                    % use the magnitudes of the fft
clear fftmat                             % relieve memory usage!!

% create the audio matrix by applying the filter
audmat = filtmat * fftmag;
clear fftmag                             % relieve memory usage!!

% ----- GRAPH THE DFT -----
[rf,cf]=size(filtmat);
duration=ns/fs;

```

```

[raud,caud]=size(audmat);
lenpiece=round(caud/2);
dftaud=zeros([rf,lenpiece+1]);

% lenpiece is half the
% number of columns in audmat
% dftaud will contain the
% magnitudes of the FFTs
% of the rows of audmat

dftaud = abs(fft(audmat, [], 2)); % create the dft matrix for graphing
dftaud = dftaud(:, 1:lenpiece+1);

if (PLOT)

    % plots the columns of the dftaud dftaud, superimposed.

    plot(0:lenpiece,dftaud')
    axis([0 lenpiece 0 max(max(dftaud(:,3:end)))'])
    [peak,indx]=max(max(dftaud(:,floor(3*duration):end)));
    indx=indx+floor(3*duration)-2;
    xtick=indx*(0:floor(lenpiece/indx));
    set(gca,'XTick',xtick);
    title(['DATAREDUCE('' infile '' , ' num2str(overlap) ', ' ...
           num2str(pow) ') ' ' date'])
    xlabel(['The prominent frequency ' num2str(indx) ' represents ' ...
           num2str(60*indx/duration) ' beats per minute.'])
    ylabel(['magnitude'])

end

% ----- SAVE THIS ENVIRONMENT'S VARIABLES IN A .MAT FILE -----
comm=['save ',outfile];
eval(comm);

```

### A.3 Plot DFT

```
function plotdft(infile, bass_in_front)

%PLOTDFT Plots the DFT of the audio matrix
% PLOTDFT(INFILE,BASS_IN_FRONT)
%
% infile=name of .mat file to process
% bass_in_front = boolean (0 or 1) to plot the bass or treble in front
%     if bass_in_front = 1, the function plots the bass in front.
%     Otherwise, the function plots the treble in front
%
%
% This program expects that the .mat file it is reading in was produced
% by the datareduce function.
% If this is the case, the filename should look like <name><overlap>.mat
%
% The graph plotted is a frequency vs. magnitude graph. The colors
% correspond to the particular frequency bands of the audio matrix:
%
% Blue represents low frequencies (bass),
% Yellow represents mid frequencies and
% Red represents high frequencies (treble)
%
% NOTE: This program references the following:
% 1. an external function rainbow
% 2. a variable called 'dftaud' expected to be found in the .mat file
% 3. a variable called 'duration' expected to be found in the .mat file
% 4. a variable called 'lenpiece' expected to be found in the .mat file
%
%
% Written By Rachel Hall
%           Joseph E. Flannick
%
%

if nargin==0, error('You must specify a file name. '), end
if nargin < 2, bass_in_front=0; end
```

```

[path,infile,ext]=fileparts(infile);
if isempty(ext), ext = '.mat'; end
if ext ~= '.mat', error ('The file must be a .mat file'); end

% ----- READ IN THE FILE -----
outfile=[path,'/' infile, ext];
comm=['load ',outfile];
eval(comm);

rdft = size(dftaud,1);

set(gca,'NextPlot','replacechildren')

% ----- plots the columns of dftaud, superimposed. -----

if bass_in_front
    set(gca,'ColorOrder',flipud(rainbow(rdft)))
    plot(0:lenpiece,flipud(dftaud)')
else
    set(gca,'ColorOrder',rainbow(rdft))
    plot(0:lenpiece,dftaud')
end

% ----- LABEL THE GRAPH -----

axis([0 lenpiece 0 max(max(dftaud(:,3:end)))'])
[peak,indx]=max(max(dftaud(:,floor(2*duration):end)));
indx=indx+floor(2*duration)-2;
xtick=indx*(0:floor(lenpiece/indx));
set(gca,'XTick',xtick);
title(['DATAREDUCE('' ' infile '' , ' num2str(overlap) ', ' ...
        num2str(pow) '' ' ' date])
xlabel(['The prominent frequency ' num2str(indx) ' represents ' ...
        num2str(floor(60*indx/duration)) ' beats per minute.'])
ylabel(['magnitude'])

```



## A.5 Create wave tracks from the audio matrix

```
function write_fbands(infile, overlap)

%WRITE_FBANDS  Creates .wav tracks for each band in the
%              audio matrix
%  WRITE_FBANDS(INFILE,OVERLAP)
%
%  infile=name of .mat file to process
%  overlap=overlap used to generate the .mat file
%
%  This program expects that <infile> was produced by
%  datareduce
%  It expects the filename to be of the form:
%  <infile><overlap>.mat
%
%  It creates the tracks with the filename:
%  <infile><overlap>_<frequency band>.wav
%  and a sampling rate of 44.1 kHz (mono)
%
%  Written by Joseph E. Flannick

% ----- PROCESS COMMAND LINE ARGUMENTS -----
if nargin < 2, overlap=212; end
if nargin < 1, error('You must specify a file name'), end

format compact

[path,infile,ext]=fileparts(infile);
if isempty(ext), ext='.mat'; end

in_matfile=[path, '/', infile, num2str(overlap),'.mat']
comm=['load ',in_matfile];
eval(comm);

% ----- SET UP OUR VARIABLES -----
fullfile=[path, '/', infile, '.wav']
size_ff = wavread(fullfile, 'size');
```

```

stretch=20;
size_filtmat=size(filtmat);
cf = size_filtmat(2); % number of columns in filter matrix
nsamples = size_ff(1); % number of samples in song
bigfiltmat=[];
hissy=[];
snd=[];

% generate a random vector of the same length
% as filtmat x stretch x 2
hiss=2*rand(1,cf*stretch*2)-1;
hissfft=fft(hiss);
hfft=hissfft(1:cf*stretch);

nfft=2^pow;
white=2*rand(1,100000)-1; % 100000 was determined experimentally
stretchfactor=round(nsamples/caud);
hlen=stretchfactor*caud;

hissy_song = [];

% ----- CREATE TRACKS -----
for i=1:raud
    message=['Creating Track ', num2str(i), '...'];
    disp(message)

    row=intrp(filtmat(i,:),stretch);

    temp = row.*hfft;
    prod=[temp 0 conj(temp(length(temp):-1:2))];
    ifftprod=real(ifft(prod));

    bighissy=perextend(ifftprod,hlen);

    % stretch audmat to the length of the original song
    stretchaud=intrp(audmat(i,:),stretchfactor);

    soundmat(i, :) = bighissy.*stretchaud;

```

```

end

% ----- WRITE THE AUDIO TRACKS -----

maxsoundmat=max(max(abs(soundmat)))');
hissy_song=soundmat(1,:);

for i=1:raud
    message=['Writing Track ', num2str(i), '...'];
    disp(message)
    if i > 1, hissy_song=hissy_song+soundmat(i,:); end
    out_file=[path,'/',infile,num2str(overlap),'_', num2str(i)];
    wavwrite(soundmat(i,:)/maxsoundmat, fs, out_file)

end

% --- WRITE THE COMBINED TRACK (sum of all frequency bands) ---
maxsoundmat=max(max(abs(hissy_song)))');
out_file=[path,'/',infile,num2str(overlap),'_all'];
wavwrite(hissy_song/maxsoundmat, fs, out_file)

format

```

## B Data

| Number of Samples | Data Reduction on my PC |           | Data Reduction on Dr. Waksman's PC |           |
|-------------------|-------------------------|-----------|------------------------------------|-----------|
|                   | Old (sec)               | New (sec) | Old (sec)                          | New (sec) |
| <i>N</i>          |                         |           |                                    |           |
| 26,384            | 0.3533                  | 0.0948    | 0.3800                             | 0.0500    |
| 49,367            | 0.6815                  | 0.1321    | 0.6000                             | 0.1100    |
| 82,091            | 1.0844                  | 0.2161    | 0.9900                             | 0.1100    |
| 127,336           | 1.6606                  | 0.3330    | 1.5900                             | 0.2200    |
| 172,499           | 2.2646                  | 0.4448    | 2.2000                             | 0.3300    |
| 226,726           | 3.0472                  | 0.5866    | 2.8000                             | 0.3900    |
| 302,699           | 3.9312                  | 0.7717    | 3.7400                             | 0.5500    |
| 397,342           | 5.6075                  | 0.9959    | 4.8900                             | 0.7100    |
| 516,901           | 7.0123                  | 1.3226    | 6.3700                             | 0.9900    |
| 679,394           | 9.7853                  | 1.7040    | 8.3500                             | 1.2100    |
| 896,448           | 12.2876                 | 2.3942    | 11.0900                            | 1.7100    |
| 1,085,568         | 14.6505                 | 3.1134    | 13.2400                            | 1.9200    |
| 1,327,242         | 19.1121                 | 3.5245    | 16.3100                            | 2.3100    |
| 1,613,013         | 24.4517                 | 4.3441    | 20.1100                            | 2.9600    |
| 1,948,659         | 26.9017                 | 5.6054    | 23.7800                            | 3.4000    |
| 2,307,504         | 31.7901                 | 10.608    | 29.2700                            | 4.1200    |
| 2,764,652         | 39.0258                 | 17.8634   | 33.9400                            | 4.9400    |
| 3,292,951         | 39.8619                 | 23.9080   | 35.6500                            | 6.1500    |
| 3,854,422         | 48.5933                 | 50.7404   | 42.2900                            | 7.0300    |
| 4,410,015         | 52.1133                 | 57.7296   | 49.2100                            | 8.4000    |
| 5,161,954         | 54.3166                 | 73.9578   | 56.0300                            | 8.6200    |
| 6,141,154         | N/A                     | N/A       | 66.5100                            | 10.2700   |
| 7,357,798         | N/A                     | N/A       | 81.0100                            | 13.2900   |
| 8,251,130         | N/A                     | N/A       | 93.5400                            | 14.2800   |
| 9,569,804         | N/A                     | N/A       | 123.4300                           | 17.1900   |
| 11,227,392        | N/A                     | N/A       | 122.3200                           | 27.7350   |

## C CD Contents

1. **Example of Rhythm to Vibration to Pitch with square waves at 1, 5, 10, 15, 25, 50, 100 Hz**
2. **ZZ Top - “Sharp Dressed Man” Original Clip**
3. Clip with high frequencies filtered out. The result is a low frequency envelope applied to white noise
4. Frequency Band 1 (lowest frequencies)
5. Frequency Band 2
6. Frequency Band 3
7. Frequency Band 4
8. Frequency Band 5
9. Frequency Band 6
10. Frequency Band 7
11. Frequency Band 8
12. Frequency Band 9
13. Frequency Band 10
14. Frequency Band 11
15. Frequency Band 12
16. Frequency Band 13
17. Frequency Band 14
18. Frequency Band 15
19. Frequency Band 16
20. Frequency Band 17
21. Frequency Band 18
22. Frequency Band 19
23. Frequency Band 20
24. Frequency Band 21 (highest frequencies)
25. Sum of all frequency bands applied to white noise
26. **Ivo Papasov - “Hristianova Kopanitsa” Original Clip**

27. Clip with high frequencies filtered out. The result is a low frequency envelope applied to white noise
28. Frequency Band 1 (lowest frequencies)
29. Frequency Band 2
30. Frequency Band 3
31. Frequency Band 4
32. Frequency Band 5
33. Frequency Band 6
34. Frequency Band 7
35. Frequency Band 8
36. Frequency Band 9
37. Frequency Band 10
38. Frequency Band 11
39. Frequency Band 12
40. Frequency Band 13
41. Frequency Band 14
42. Frequency Band 15
43. Frequency Band 16
44. Frequency Band 17
45. Frequency Band 18
46. Frequency Band 19
47. Frequency Band 20
48. Frequency Band 21 (highest frequencies)
49. Sum of all frequency bands applied to white noise
50. Original recording combined with rhythm track
51. **Goo Goo Dolls - “Big Machine” Original Clip**
52. Sum of all frequency bands applied to white noise
53. **Blue Oyster Cult - “Dont Fear The Reaper” Original Clip**
54. Sum of all frequency bands applied to white noise
55. **Metallica - “Fade To Black” Original Clip**

56. Sum of all frequency bands applied to white noise
57. **Korn - “Alone I Break” Original Clip**
58. Sum of all frequency bands applied to white noise
59. **Van Halen - “Jump” Original Clip**
60. Sum of all frequency bands applied to white noise
61. **Tantric - “Breakdown” Original Clip**
62. Sum of all frequency bands applied to white noise
63. **Tantric - “Mourning” Original Clip**
64. Sum of all frequency bands applied to white noise
65. **Corey Taylor - “Bother” Original Clip**
66. Sum of all frequency bands applied to white noise