

Music Transcription for the Lazy Musician

18-551 Spring 2000
Final Project Report
Group 7
May 8, 2000

Michelle Kruczuk
kruczuk@andrew.cmu.edu

Ernest Pusateri
pusateri@andrew.cmu.edu

Alison Covell
covell@andrew.cmu.edu

1. The Problem

Many times musicians find it useful and enjoyable to view the score of a piece of music while it is being played. Other times, musicians come up with melodies in their minds, but do not have the time to sit down and transcribe the tunes in their heads to paper. A musical transcription device that could take music being played and produce the musical score would be very useful in the situations described above. The transcription device will also be useful for students practicing. They will be able to tell when they are playing incorrect notes without having to be told. Most people have problems determining the pitch by directly listening to a musical score. This is, in large part, because each source in a musical performance produces a signal with a fundamental frequency as well as harmonics that, in the case of certain instruments, may have an even stronger presence than the actual fundamental. There are even cases where the fundamental does not exist at all, although these sounds are primarily made with electronic synthesizers. Our objective is to automate the process of musical transcription for a single melody played by any instrument to eliminate the need for musicians to transcribe their own music.

2. Background

In most cases, musical sound is represented by the fundamental frequency, also known as the pitch of the note, as well as several harmonics. The harmonics of a note are simply sinusoidal waves with frequencies at integer multiples of the fundamental. For example, a sound that has a fundamental frequency of 100Hz, will have its first harmonic at 200Hz, the second at 300Hz, then 400Hz, 500Hz, etc... In instruments such as the oboe, the fundamental frequency exists, but the amplitude is significantly lower than that of its harmonic components, and for several sounds that electronic synthesizers are able to produce, the fundamental is not there at all. Our ears account for this already, and although the fundamental may only be slightly audible, we still hear the note as though it is being played at the fundamental frequency. This makes it more difficult to determine the fundamental frequency because of course, the Fourier Transforms are not able to pick

up on this and insert the fundamental. Therefore, the fundamental frequency cannot be determined by taking the Fourier Transform and simply finding the maximum peak. These variations on harmonic components need to be taken into account when trying to find the correct fundamental frequency.

Once it has been determined, each fundamental frequency translates to a certain note in standard Western Music notation. The standard notation consists of a treble and bass clef, with 5 lines each. Each line and each space between represents a single note, and a certain frequency. A sample of standard notation in the treble clef is shown below.













Sharps and flats are also included, so that each note is exactly one half step away from the next. The relationship between one note, and the note one half step above is determined by a logarithmic scale, so that each note is exactly $2^{(1/12)}$ above the next. An example of a chromatic scale, and the related frequencies is shown here.

A	$220 \cdot 2^{(0/12)} = 220\text{Hz}$
A#	$220 \cdot 2^{(1/12)} = 233\text{Hz}$
B	$220 \cdot 2^{(2/12)} = 247\text{Hz}$
C	$220 \cdot 2^{(3/12)} = 262\text{Hz}$
C#	$220 \cdot 2^{(4/12)} = 277\text{Hz}$
D	$220 \cdot 2^{(5/12)} = 294\text{Hz}$
D#	$220 \cdot 2^{(6/12)} = 311\text{Hz}$
E	$220 \cdot 2^{(7/12)} = 330\text{Hz}$
F	$220 \cdot 2^{(8/12)} = 349\text{Hz}$
F#	$220 \cdot 2^{(9/12)} = 370\text{Hz}$
G	$220 \cdot 2^{(10/12)} = 392\text{Hz}$
G#	$220 \cdot 2^{(11/12)} = 415\text{Hz}$
A	$220 \cdot 2^{(12/12)} = 440\text{Hz}$

Besides representing the frequencies as notes on a clef made of lines and spaces, another important aspect is representing the duration of the musical sound. In musical scores, this is done with another standard, whereby the shape of a note or rest (space of time in which no musical sound is played) indicates how many beats it will receive. This number of beats can be used to determine the actual number of seconds a particular note

takes by examining the speed at which the music is played, usually specified in beats per minute (bpm). A chart showing the number of beats assigned to a particular note or rest is shown below:

	Note	Rest	Duration
Eighth			0.5 beats
Quarter			1.0 beat
Half			2.0 beats
Dotted Half			3.0 beats
Whole			4.0 beats

As you can see, a whole note typically receives 4 beats, so a piece of music played at a tempo of 120bpm, would have 30 whole notes in one minutes.

Most professional musicians can play 16th notes at speeds of up to 120bpm. We intend to go one step beyond this to 32nd notes. At 8 32nd notes per beat, and 120bpm this corresponds to 960 32nd notes in one minutes, roughly one 32nd note every 16th of a second, which is much faster than most people can play.

When the fundamental frequency is combined with its respective length and translated into standard music notation, a musical score is produced. An example of a very basic musical score is presented here:



There are many other components of a musical score that are not addressed in this project. These components include more expressive parts of music, such as loudness, accents and changes in tempo. The basic score our algorithm implements in the end only includes which notes were played and how long they were held.

3. Solution

3.1 Previous Projects

Several groups have done projects in the past that involve some form of pitch detection algorithm implementation. The most recent project, implemented in spring 1999, involved a pitch-correction implementation that required the detection of the incoming pitch, which was computed using an autocorrelation algorithm. Once they found the pitch being sung they compared it to known pitches to see how far out of tune the singer was. Finally, they added or subtracted from the original pitch till it matched one of the known pitches and outputted the new waves to a speaker.

With the algorithm they used their system had a slightly delay in the actual correction of the pitch from the time the singer started each note. There was also some amount of noise introduced into the system they believed to be due to the limited accuracy of their algorithm. This group seemed to achieve reasonable results, but put several limitations on their system. They only considered pitch correction of the human voice, which has a limited range and harmonic distribution. This would make their algorithm fail for certain kinds of instruments, and for sounds that exceed the human vocal range.

The next group we investigated implemented a voice-to-midi system in the previous year. Their algorithm was similar to ours in that it looked at the frequency response of a signal and used harmonic ratios to determine which frequency is most likely the fundamental. They then take the frequencies and write them to a midi-formatted file to produce a playable piece of music. They wanted to attempt a real-time solution of this, but were not able to implement this. The output of their system also had a great deal of high pitched glitches and noise.

The most obvious feature we attempt to improve upon with this project is the real-time implementation. This group again limited themselves to only a few octaves of the audible range of hearing. They also did not implement their system to work for a variety of musical instruments.

Another group of the same year also attempted a pitch detection to midi system. They planned to implement this using wavelet transforms. One major problem with this we discovered as we were researching pitch detection algorithms is the lack of good resolution at extreme ends of the audible range of sounds. This group also did not implement their system in real time.

In the spring 1997 section of the course, another group implemented music to transcription system, also using wavelet transforms, and a similar ratioing algorithm to the second mentioned project. Once again their implementation was not done in real time and had a greatly limited frequency range. Their system had various glitches and blips, but was still able to distinguish most of the notes in their input files. While it mostly worked for piano and string instruments, their system failed for brass instruments. Their output also was not able to combine several notes of the same frequency into one longer note, resulting in a cluttered and difficult to read musical score.

Our most obvious improvement over the previous projects presented here is the implementation of a pitch detection algorithm that can be implemented in real-time. Another major limitation we attempt to overcome is the dependence of most systems upon the type of instrument that is being played. More specifically we try to eliminate the need for music to have a normally structured harmonic spectrum, or for the fundamental to even need to be present. Finally we intend to eliminate the highly constricted range that all the previous projects have placed upon their algorithms by being able to detect all frequencies occurring on natural instruments.

3.2 Solutions Considered

3.2.1 Modal Distribution

The first solution that we considered for our problem was to use the Modal Time-Frequency Distribution. This distribution allows one to see the strength of frequency components in narrow bands. Thus, it is well suited to the purpose of detecting a musical

tone's fundamental and harmonic components. This approach is very computationally intense, however, and would not have allowed our implementation to work in real-time.

3.2.2 Wavelet Transform

The second solution considered was the wavelet transform. Heisenberg's uncertainty principle dictates that one must always compromise between time and frequency resolution. In other words, as one obtains more time resolution, one loses frequency resolution and visa versa. The wavelet transform allows this compromise to be made differently at different frequencies. This is a valid approach for musical signals because as one moves up the musical scale, the note frequencies get farther apart. Thus, less frequency resolution is required at higher frequencies than at lower ones. If one is willing to compromise time resolution at lower frequencies, the wavelet transform is an excellent approach. We were hoping to obtain the same time resolution across the entire musical scale, however, so this approach was discarded.

3.2.3 Combining Different Window Lengths

The third approach we considered was an algorithm that we devised ourselves. This algorithm involved using two different window sizes, one very large and one very small. The result of the FFT for the very small window would be used to filter the result of the large window FFT. The goal of this algorithm was to combine the fine frequency resolution of the large window with the fine time resolution of the small window. While this algorithm had promise, we felt it would be more prudent to use an already existing algorithm that had been proven to work.

3.3 Final Solution

3.3.1 Algorithms

3.3.1.1 Pitch Detection

The algorithm that we decided to use comes from a paper by Anssi Klapuri called “Wide-band Pitch Estimation for Natural Sounds Sources with Inharmonicities.” The purpose of this algorithm is to detect the pitch of the tone being played in one time window. In other words, the algorithm says nothing about how to combine pitch estimates for many subsequent windows into a musical notation representation. I will describe this algorithm in detail in the following paragraphs.

The algorithm consists of two main parts. First independent pitch estimates are made in different frequency bands. Then, these estimates are combined into one global estimate. The algorithm for finding the independent pitch estimates are based on a simplified version of a loudness perception model proposed by Moore [3]. This model says that the loudness of a signal can be determined by the following formula:

$$L = \int_1^{40} X(e)^2 de$$

In this formula e represents the frequency in ERBs. The formula to convert a frequency from Hz to ERBs is as follows:

$$e(f) = 21.4 \log \frac{-4.37 f + 1000}{1000}$$

Klapuri approximates the formula for L with:

$$L = \sum_{h=1}^H E_h X(e_h)^2$$

In this equation $X(e_h)$ is the value of the power spectrum at harmonic h . E_h is the approximate width of the f_0 Hz wide frequency range around the partial. This can be approximated with the formula:

$$E_h = \frac{d}{df} e(f_h)$$

After studying the behavior of this function, Klapuri concluded that a function of $f_0/bandw$ could be used as an approximation to this value. He came up with the following formula:

$$E_h = a_2 \left(\frac{f_0}{bandw} \right)^2 + a_1 \left(\frac{f_0}{bandw} \right) + a_0$$

Klapuri trained the system to obtain values for a_2 , a_1 , and a_0 . The values he obtained were $a_{2,1,0} = \{-0.4, 1.2, 0.2\}$. We found that these values did not work well for our implementation. This is described in the implementation section of this report. To obtain pitch estimates in a band, L values are computed for every pitch that could have its fundamental or any of its harmonics within that band.

After obtaining loudness values within each band, the second part of the algorithm is performed. All of the bands are searched for the largest L values. These are put into a candidate list in descending order of loudness. Starting with the first candidate, all of the frequency bands, except the band from which the candidate came, are searched to determine whether that frequency has L values in those bands. If so, the L value is added to the candidate's L value. If one of the L values added to a candidate is in the candidate list, it is removed. After this process is complete, the candidate with the highest L value is taken as the pitch estimate.

Two voice pitch detection is implemented as follows: After the candidate with the highest L value is taken as the first pitch, the components in the original FFT corresponding to the first pitch's fundamental and harmonics are set to zero. The L values are then recomputed, and another pitch estimate is made.

It should be noted that the algorithm described by Klapuri also takes into account inharmonicity. We did not implement this part of the algorithm, however, and thus it will not be described.

3.3.1.2 Duration Algorithm

The algorithm we used to determine duration is based on the algorithm used by [4]. The steps in the algorithm are as follows:

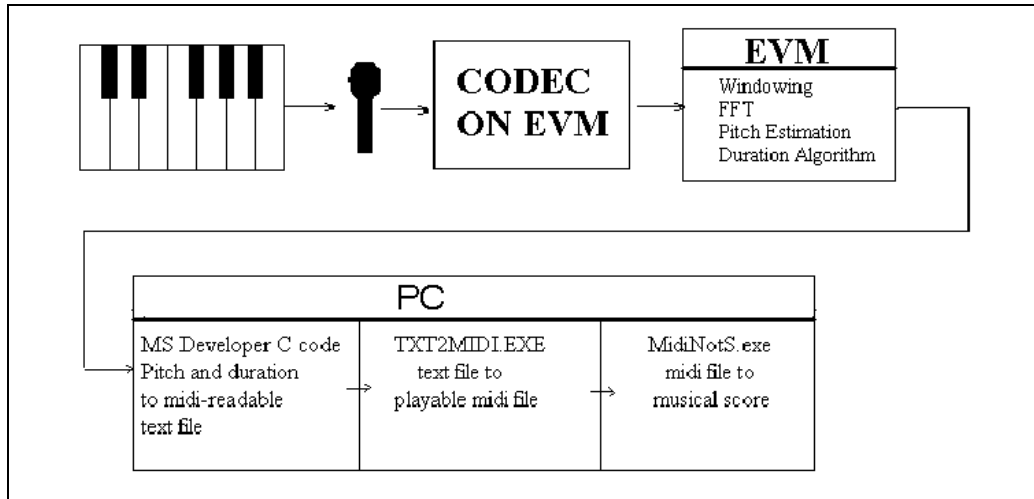
1. The length of the first note of the piece (in musical beats) is entered by the user.
2. Set window counter to 0.
3. After determining pitch of the first window, set window counter to 1.
4. Obtain the pitch of the next window. If it is the same as the previous window, increment the window counter, and repeat step 4. Otherwise move to step 5.
5. Use the value of the window counter and the value entered by the user to determine the number of windows per sixteenth note.
6. Set window counter to 0.
7. After determining pitch of the next window, set window counter to 1.
8. Obtain the pitch of the next window. If it is the same as the previous window, increment the window counter, and repeat step 8. Otherwise move to step 9.
9. Divide the value of window counter by the number of windows per sixteenth note. If the value is greater or equal to 1, the pitch is assigned this duration. Otherwise, discard the pitch as noise. Go to step 6.

3.3.2 Implementation

3.3.2.1 Data Flow

The general data flow of the musical note transcription system began with the user entering the musical length of the first note on the PC interface. For example, if the first note were a half note, .5 would be entered as its musical duration. This value would be sent to the 'C67 EVM to later be used in the duration algorithm. Next, on the EVM the pitch estimation algorithm would be performed on windows of samples of the piece of music coming from the codec. After the pitch estimation algorithm produced the fundamental frequency corresponding to the note played, the duration algorithm would be run on the EVM to find the length of the note played. After these two pieces of information, pitch and duration, were calculated on the EVM, they were sent back to the PC and stored in an array of pitches and corresponding durations for the piece of music being played. This array was then compiled into text MIDI format on the PC so that after

using the TXT2MIDI and MidiNotS programs, the transcribed notes could be viewed or heard. A data flow graph of the transcription system methodology is shown below:



3.3.2.2 Parameters

In order to achieve musical transcription in real time, the pitch detection and duration algorithms described in the algorithms section above were done on the EVM, as shown in the data flow diagram. Several parameters were chosen to fit our purposes in implementing the algorithms. First, the sampling rate parameter given to the system was 44100 samples per second. Secondly, the windows passed to the FFT were of size 4096, overlapping by half, in order to provide decent frequency resolution needed to detect low notes, while detecting notes of relatively short duration as calculated below:

$$\text{Frequency Resolution} = \frac{44100 \text{ samp./s.}}{4096 \text{ samples}} = 10.77 \text{ Hz.}$$

$$\text{Minimum Note Duration Detectable} = \frac{2048 \text{ samp.}}{44100 \text{ samp./s.}} = 46.44 \text{ ms.}$$

$$\text{32nd Note Duration at 120 bpm} = \frac{60 \text{ s./min.}}{120 \text{ beats/min.}} * \frac{1 \text{ beat}}{8 \text{ 32nd notes}} = 62.5 \text{ ms}$$

As can be seen from these calculations, the best frequency resolution using the 4096 point FFT was 10.77 Hz, but the nature of the pitch estimation algorithm assists in distinguishing lower frequencies with differences less than 10.77 Hz, so this FFT size was ideal. Also, it can be seen from the calculations that using windows of 4096 overlapping by half allows notes as fast as 32nd notes at 120 bpm to be detected, which is quite a fast tempo in music.

The 4096 point FFT was taken using the Texas Instruments radix-4 assembly FFT routine. This routine did not support the use of interrupts. Therefore, just before the assembly routine was called, interrupts were disabled, and just after the routine, interrupts were enabled again. This caused a certain number of samples from the codec to be lost, as shown below:

$$\text{Length of FFT} = 86000 \text{ cycles} * 6 \text{ ns./cycle} = .516 \text{ ms.}$$

$$\text{Number of Samples Lost} = \frac{.516 \text{ ms.}}{1/44100 \text{ samp./s}} = 23 \text{ samples}$$

This number of samples lost was relatively small since the window size to find each fundamental was 4096.

The next step was to compute the loudness values used to determine the most likely pitch candidate for the pitch estimation algorithm. The lowest frequency to detect was designated 55Hz since the sounds below this are too irregular to detect, and the upper limit for detection was designated 21,096Hz, which is approximately the upper limit of human hearing. The bands for the pitch estimation calculation were chosen in groups of 2/3 octaves ranging from 55Hz to 21,096Hz. 2/3 octaves were chosen since if the band is too narrow, the loudness value for many fundamentals will only be calculated with one harmonic. In such a case, the pitch estimation for each band would not be based on enough information.

Other parameters needed in the pitch estimation algorithm were the a0, a1, and a2 parameters used to calculate Eh values. In the original implementation of the algorithm by Anssi Klapuri, a0 = 0.2, a1 = 1.2, and a2 = -0.4 parameters were chosen by automated training with musical instrument samples. Since automated training was not

implemented in this project, values which gave the best results in general for all test samples were chosen of $a_0 = 0.2$, $a_1 = 1.2$, and $a_2 = -0.2$.

Also, in the pitch estimation algorithm, when the pitch candidates were being searched to obtain the best fundamental estimate, the number of most-likely pitch candidates to save from the possible pitch estimates across all bands was chosen. For this project, 10 most-likely pitch candidates were saved from the vector of possible candidates since this number seemed to achieve good results across the set of musical test samples.

Finally, an important parameter for the duration algorithm was the length of the musical note to which all notes were quantized. For this project, any note shorter than a sixteenth note was disregarded, and all note lengths were recorded in terms of sixteenth notes. For example, a quarter note would be represented by 4 sixteenth notes. Additionally, if the number of sixteenth notes was not exactly the length of any musical note duration, the note was rounded to the closest musical note duration. For example, if the note were found to be five sixteenth notes long, the note duration would be rounded to a quarter note.

3.3.3 Optimization

3.3.3.1 Speed

Because we wanted to create a real-time music transcription system, and the algorithm was being implemented on the EVM, optimization of the EVM code was an integral part of achieving our goals. We calculated the total number of allowable cycles for a real time implementation, using the fact that the window size was 4096, overlapping by half, as such:

$$\frac{1 \text{ cycle}}{6 \text{ ns.}} * \frac{2048 \text{ samp.}}{44100 \text{ samp./s.}} = 7.74 \text{ million cycles}$$

A chart showing the approximate initial number of cycles in the parts of our code consuming the most cycles, and the amount of improvement achieved with each optimization for these parts is shown below:

Improvement vs. Cycles	FFT	Loudness Values	Overall
Optimization on, most data and code external	1.2 Million	33 Million	60 Million
Transferred large data structures into internal buffer when needed	400,000	15 Million	25 Million
Moved program code to internal memory	86,000	6 Million	8 Million
Moved RTS libraries to external memory	86,000	24 Million	30 Million
Changed processor to C6700	86,000	12 Million	15 Million
Eliminated unnecessary divides and print statements	86,000	1.2 Million	3 Million

When the EVM code was initially profiled, the optimizations option was checked in Project settings, but most of the data was in external memory since it could not be fit in internal memory. At this point, the total number of cycles was around 60 million, which was exceeding the limit of 7.7 million cycles by quite a lot. Therefore, the memory management system, described below, was implemented, and the data structures were transferred from external memory into an internal memory buffer using DMA just before they were needed. The computation was done with the data in internal, and then the results would be shipped back to external using DMA so that the internal buffer could be used for the data of the next computation. This memory management scheme brought significant improvements in performance, but the total number of cycles was still 3 times too big for real time.

Next, with the help of the teaching assistant, Pete, an error was found in the linker command file, whereby the program code was being placed in external memory. After fixing this and getting the program code placed in internal memory, the number of cycles was reduced to almost the amount needed for real time, but improvement was still needed.

Therefore, the RTS libraries were moved to external memory so that there would be room in internal memory to bring in more data needed in the algorithm computation. However, when these libraries were moved to external, the number of cycles went up by

a surprisingly big amount, making the total number of cycles again over 3 times too big for real time.

The RTS libraries contained many functions, including routines for floating point adds, multiplications, divisions, and also those needed for the McBSP. When the assembly code for the project was viewed, it could be seen that many calls were being made to floating point functions, instead of using the floating point assembly instructions for the 'C67. Therefore, upon examination of the Project settings, it could be seen that the processor type was not set. The processor setting was changed to C6700, and the project was recompiled. This change reduced the total number of cycles by about 2, but the number of cycles was still about 5 million too big for real time.

Finally, looking at the assembly code for the project revealed that every time a divide appeared in the code, a function call to the off-chip RTS libraries was made, slowing down the code dramatically. Thus, as many divides as possible, as well as print statements, which were also suspected to be slowing things, were eliminated from the code. These final optimizations brought the total number of cycles down to 3 million, well under the 7.7 million needed for a real time implementation.

3.3.3.2 Memory

Since the amount of data needed to implement our pitch estimation algorithm was much more than could be stored in internal memory all at once, memory management was a very important optimization in order to achieve a real time implementation. The general memory management technique was to designate an internal memory buffer, to which data would be transferred from external memory using DMA just before it was needed in the algorithm. After the data was used in the algorithm, and its results were stored in internal memory, the results would be shipped back to external memory using DMA, and new data would be brought into the internal buffer.

The particulars of the memory management for the algorithm are as follows. After the first musical note duration was passed over to the 'C67 EVM using a synchronous mailbox write, the pitch estimation algorithm was begun on the EVM.

The first step to implementing it was to pass windows of the samples coming from the codec into an FFT to obtain the power spectrum. Three external data buffers of

size 4096 (the FFT size) were used to store 2048 samples coming from the codec interleaved with 2048 zeros. After one of these external data buffers was filled up with real samples from the codec, it and the previously filled buffer would be sent to internal memory using a DMA transfer. Also, at this time, the twiddle factors, which were computed and stored in external memory before the pitch estimation algorithm began, were transferred to internal memory using DMA. The FFT was taken with all of the data structures in the internal memory buffer. The result of the FFT was stored in internal memory, and the power spectrum of the resulting FFT was computed, and stored in an internal buffer, as well. At this time, the power spectrum result was shipped to external memory using a DMA transfer.

The second phase in the pitch estimation algorithm was computing the loudness values for each frequency band. Therefore, the internal buffer was now designated to store the structure for the loudness, frequency, and E_h values per frequency band (as explained above in the algorithms section). Bands of the power spectrum result were shipped in one at a time from external memory using a DMA transfer, and each of these was used in the loudness values calculation to fill in one band of the loudness/frequency structure in internal memory. After the internal loudness/frequency structure was filled in, it was searched in order to find the most likely fundamental for the note being played, and the pitch estimation was complete.

Finally, the duration of the fundamental was calculated as explained above using the ratio of the number of samples of the current fundamental to the number of samples of the initial fundamental. Thus, the duration algorithm required no significant data structures or transfer.

Once both the fundamental and its duration were found on the EVM, they were stored in a structure, which was passed back to the PC, using a synchronous mailbox transfer.

4. Results

4.1 Monophonic Pitch Detection

Below is a chart of basic music we used to test our algorithm, and the results for different ranges in the musical scale. For the single note tests, we compiled a group of files for each instrument that consisted of just a few windows containing the same pitch. For the chromatic scale tests, instruments were played from their lowest capable pitch to their highest with little or no rests between played at 120bpm. Most notes were detectable by our algorithm, but in certain cases noise was apparent in the results. To account for these in inaccuracies, we decreased the percentage of correct notes detected by the percentage of noise added. In other words, if a note was correctly detected, but an extra, incorrect, tone was also detected it only counted as .5 when the total number of successes were summed.

Instrument	a1-b2	c2-f4	f5-above
Midi-clarinet (single notes)	NA	100%	100%
Midi-violin (single notes)	50%	100%	100%
Midi-piano (single notes)	75%	90%	100%
Real clarinet (chromatic scale)	100%	90%	50%
Real clarinet (chromatic scale) 2x fast	100%	100%	100%
Real clarinet (chromatic scale) 3x fast	100%	100%	100%
Midi-violin (chromatic scale)	75%	94%	53%
Midi-violin (chromatic scale) 2x fast	97%	100%	100%
Midi-violin (chromatic scale) 3x fast	98%	100%	100%
Midi-flute (chromatic scale)	93%	100%	86%
Midi-flute (chromatic scale) 2x fast	100%	100%	100%
Midi-flute (chromatic scale) 3x fast	15%	100%	100%
Sawtooth wave (function generator)	100%	100%	100%

As can be seen by the table, most of our basic tests cases had very good results. There are various properties or music that contribute to the poor results in various ranges of certain instruments. For example, the real clarinet chromatic scale had poor results in

the upper range due to the increased amount of distortion as the instrument pushed the boundary of its natural range and fell somewhat out of tune. In the upper range of the midi instrument chromatic scales, the notes began to smear together, causing some harmonic distortion when one note changed into another. It is interesting that the accuracy of our results for the chromatic scales actually increased as we increased the tempo. This is probably because, with the midi tones, the attacks and decays of each of the notes are shortened when the length of the tone is shortened. It is these attacks and decays that seem to cause the most inaccuracy in our system.

In our demo we used a real cello and trombone, which we inputted to the codec through a microphone. The algorithm performed rather well on the trombone, detecting most notes with some noise. The trombone also revealed the ability of the system to detect notes at various volumes. On soft and normal volumes it performed fine, but when a very loud note was played some noise was also detected with the note. This is due partially to the way we threshold noise, and partially to the distortion of the sound when it is played extremely loudly. The cello did not perform so well. Upon looking closely at the output file, it was determined that the notes were being correctly detected, but noise was introduced in the attack and decay of most notes. This is due to multiple items. Firstly, the cello has a great deal of reverberation within its body even after a note has stopped. Secondly, if the note is not struck cleanly and starts just slightly out of tune, the results will not be as expected. When notes were played cleanly and reverberations kept to a minimum, the algorithm performs just as well as with any instrument. Our algorithm will work with string instruments, but the notes have to be very clean and in tune.

The results we obtained are consistent with those obtained by Klapuri [2]. In his tests, he obtains almost 100% correct detection from c2-c4. The discrepancies between our results and his, especially in the real clarinet and midi violin cases, are probably due to the differences in how our tests were run. Klapuri tested his algorithm on very short samples of individual instruments, not continuous music. Thus, the distortion introduced by attacks and decays was not present in his results.

4.2 Polyphonic Pitch Detection

Although the algorithm we were using was designed for single pitch detection, Klapuri adapted the algorithm to perform detection of two pitches [2]. We implemented this adapted algorithm, but did not implement a duration algorithm or any post processing to go along with it (unlike in the monophonic case).

To test the performance of this adapted algorithm, we felt it would be most informative to use an instrument and range on which single pitch detection worked well. This way we could evaluate the algorithm's polyphonic possibilities separately from the monophonic limitations we had already discovered. Thus, the test data consisted of a midi clarinet in the range from c2 to c4. A chromatic scale was played from c2 to c4, while c2 was held. Then the same chromatic scale was played while c3 was held.

Running this test, the program succeeded in detecting the two pitches in at least one window during the duration of each note 87.5% of the time. The number of windows in which the pitch was detected was very irregular, however, and would have made it difficult to create a successful duration algorithm. It would also be very difficult to determine which windows' pitch estimates were correct and which were noise caused by irregularities in the two tones.

In Klapuri's tests, he averaged about 80% correct detection on two pitches. His tests were run on single windows, however, not continuous music. Thus, he had only one chance to guess the correct pitches, while we had the whole duration of the note. This is significant because, even with midi tones, the harmonic characteristics of the tone will change during its sounding. He also ran his tests on many different instruments, while we picked an instrument that performed well with the single pitch detection algorithm.

The reason for the poor performance of the algorithm in polyphonic pitch detection is partially due to the way in which the information from the first pitch is removed from the FFT before an attempt is made to find the second pitch. The FFT values corresponding to the first pitch's fundamental and harmonics are simply set to 0. This has disastrous effects when the first pitch shares harmonics with the second. In the test we ran, none of the octaves were detected, probably because the higher pitch shares all of its harmonics with the lower one. Both fourths and one of the fifths were also missed, again because the two tones in these intervals share many harmonics.

5. Schedule & Task Distribution

	Alison	Ernie	Michelle
Week 1: 2/28/00	Research pitch Detection	Research pitch Detection	Research notation Software
Week 2: 3/6/00	Explored various algorithms.	Explored various algorithms.	Explored various algorithms.
Week 3: 3/13/00	Began implementing Klapuri's algorithm.	Began implementing Klapuri's algorithm.	Began implementing Klapuri's algorithm.
Week 4: 3/20/00	Implement code to compute power spectrum.	Implement code to determine pitch from L values.	Implement code to compute L values.
Week 5: 3/27/00	Spring Break	Spring Break	Spring Break
Week 6: 4/3/00	Port code to EVM.	Port code to EVM.	Port code to EVM.
Week 7: 4/10/00	Non-real-time implementation.	Non-real-time- implementation.	Non-real-time- implementation.
Week 8: 4/17/00	Real-time implementation and testing. Implement duration algorithm.	Real-time implementation and testing	Real-time implementation and testing. Research MIDI file format.
Week 9: 4/23/00	Real-time implementation and testing. Integrate duration algorithm.	Real-time implementation and testing. Implement midi file conversion.	Real-time implementation and testing. Implement midi file conversion.

6. References

- [1] Klapuri, Anssi “Wide-band Pitch Estimation for Natural Sound Sources with Inharmonicities”
- [2] Klapuri, Anssi “Pitch Estimation Using Multiple Independent Time-Frequency Windows”
- [3] Moore, Glasberg, Baer, “A Model for the Prediction of Thresholds, Loudness, and Partial Loudness,” *J. Audio Eng. Soc.*, vol. 45, No. 4, April 1997
- [4] Nedel, Ng, Yamaguchi, “Automated Music Recognition and Transcription,” *18-551 Final Report*, May 1997

7. Web References

- Radix-4 FFT Assembly file (cfftr4.asm)
■ <http://www.ti.com/sc/docs/tools/dsp/ftp/c67x.htm>
- Program txt2midi (Used to put our transcribed notes and durations into midi format)
■ <http://www2.iicm.edu/diditest.nametest>
- MidiNotate (Displays a midi file in musical notation)
■ <http://www.notation.com/midinotate.htm>