MOZART: Making Orchestra Zippy; Automatic Reliable Transcription

Group 6
David Berol (dberol@andrew.cmu.edu)
Scott Storck (sstorck@ece.cmu.edu)
Michael Wenske (mwenske@ece.cmu.edu)

18-551 Digital Communications and Signal Processing Design
Spring 2004

Table of Contents

Introduction

Music is an integral part of human culture. Knowledge of music was taught first by training and memorization, and then, around 1300, through a crude form of text. Over time, the art of notating music has evolved to the current form of sheet music. This physical form of music makes it possible to teach and reproduce any composition in a standardized format.

To read sheet music requires considerable skill, but transcription or the act of listening to music and writing down its notation, is an even more difficult challenge. Manually constructing musical notation from an auditory piece of music is laborious and time consuming. It would save composers significant time and money if a computer could do this automatically. However, it is difficult for a computer to do this as well. Currently there is no robust, completely accurate product available, making it a hot topic of research.

This task is made considerably more difficult when a composition contains multiple notes, possibly from multiple instruments, being played simultaneously. This is called polyphonic sound. While the few available commercial automated transcription products perform reasonably well on monophonic sound, they all produce very poor results for polyphonic. Despite the current inability to perfectly transcribe music with a computer, new algorithms are making automatic transcription considerably more reliable. Advances in digital signal and general purpose processors have made it possible for these increasingly complex algorithms to be implemented.

Reliable polyphonic transcription will open up a world of possibilities in many different fields:

Musicians

- Learn to play a piece despite only having an audio recording of it.
- Quickly create sheet music by simply performing a piece, even if they do not know how to write sheet music.

Music education

- Allow students to easily study the structure of any composition.
- Help a student evaluate the accuracy of their performance by enabling a computer to automatically compare their performance with the sheet music.

Musical databases

- Storing music in notation form vastly reduces the space required.
- Easily search within a piece of music for a given melody or other characteristic.

Manual transcription

- Give a human transcriber a good head start towards producing perfect sheet music, vastly reducing the time and expense required.

Past 551 projects dealing with transcription did not attempt to handle polyphonic music, and were only moderately successful with monophonic. A spring 2000 group implemented a time-domain approach using wavelet filters and zero-crossings to determine note frequencies. The spring 1998 group used a discrete wavelet transform (DWT) frequency domain approach. In spring 1997, another group used the short time Fourier transform (STFT) frequency domain approach. In addition to their constriction to monophonic sound, these projects required pauses between each note and input signals with unrealistically low noise. Pitch detection remained poor despite these limitations.

The vast majority of musical pieces contain polyphonic sound. The difficulty in transcribing it lies in correctly identifying the various simultaneous frequencies (notes) in the music signal. Since there are multiple frequencies present at any given time, a strictly time domain analysis, as done in the 2000 project, simply cannot be used. A frequency domain approach must be used. When a real instrument plays a given note, it actually produces many other "harmonic" (false) notes whose frequency is some integer multiple of the real note's. As a result, the greatest challenge is determining the frequencies that are the actual notes and discarding all of the others. Polyphonic music makes this task significantly harder because some of the real notes may happen to have the same fundamental frequency as a false note. Polyphonic sound can occur in several ways:

- One instrument, several notes at one time, i.e. piano chords.
- Multiple instruments, one note each, i.e. flute duet.
- Multiple instruments, each playing multiple notes, piano and guitar chords.

Our goal was to create polyphonic capable music transcription software, and it was achieved in two stages. First, we implemented monophonic transcription of real instrumental sounds without the aid of delays between the musical notes. After that, we extended the program to do polyphonic transcription with accuracy comparable to (or better than) available commercial products. A method involving many new algorithms, presented by Luis Martins and Anibal Ferreira at the Audio Engineering Society's 112th Convention and described in detail in Martins' MSc thesis, was used to make this success possible.

The Solution

We found an algorithm by Luis Martins and Anibal Ferreira presented at the Audio Engineering Society 112th Convention in May of 2002 [1] that should allow us to meet our aforementioned goals. Martins and Ferreira describe a frequency domain approach that has been successful for monophonic transcription and worked reasonably well with polyphonic transcription.

The Martins and Ferreira method is divided into two phases: on-line processing and off-line processing. The purpose of the on-line processing stage is to identify musical notes. The off-line processing stage then takes these results and fine-tunes them, eliminating false notes and breaking apart notes that were falsely combined. We will give a brief overview of the concepts and goals of these two pieces of the algorithm.

On-line Processing

The on-line processing stage is broken down into three sub-stages: frequency analysis, harmonic analysis and harmonic structure tracking.

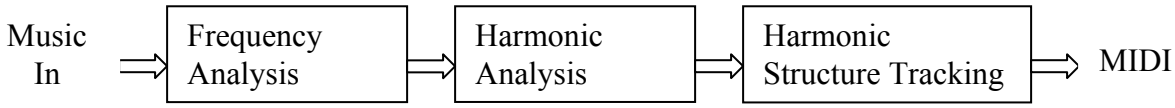

Fig. 1: On-line processing flow

In the frequency analysis portion of the algorithm, we divide the input signal up into N-point discrete samples referred to as frames. In the case of the Martins and Ferreira paper this size was 1024. Each frame is constructed so that it will overlap with the previous frame by fifty percent. Identification of the frequencies present in each frame is then obtained by an N-length sine window and N-point Odd Discrete Fourier Transform (ODFT) as outlined in [2]. This process should identify all the frequencies of each note present in a given frame.

An increased frequency resolution beyond that of a normal STFT is essential to correctly identify the notes present in each frame. The ODFT combined with interpolation of partial frequencies accomplishes this. The ODFT begins by taking an input frame with 50% overlap from the previous frame and multiplying it by a sine window:

$$h(n) = \sin\frac{\pi}{N}\left(n+\frac{1}{2}\right), \quad 0 \le n \le N-1$$

Eq. 1: Sine Window function

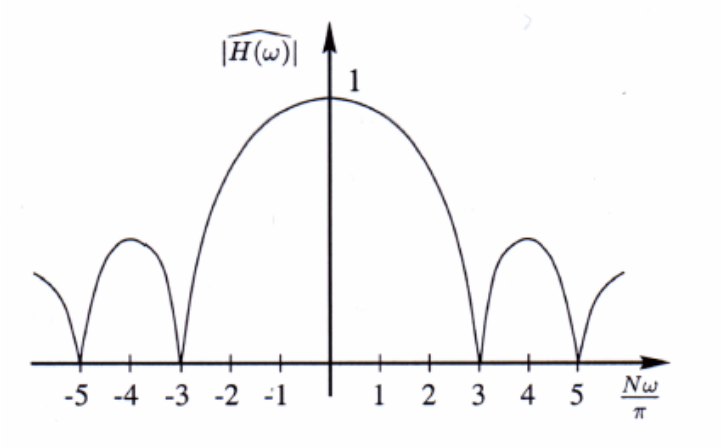

Fig. 2: Sine Window impulse response

A complex phasor is then multiplied by each of these values and this data is sent into an FFT. The Magnitude of the FFT is then taken:

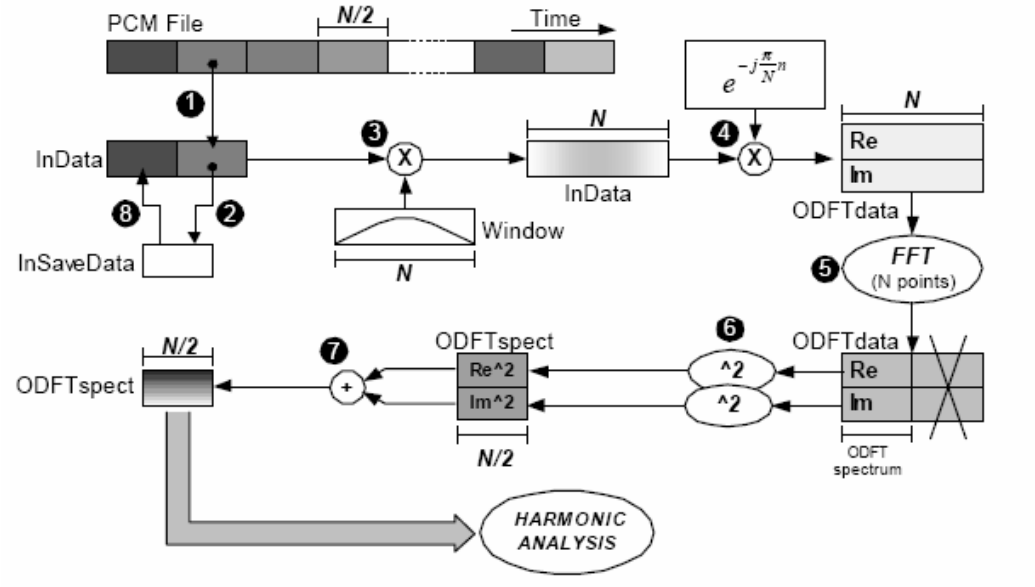

Fig. 3: ODFT flow

The harmonic analysis portion of the algorithm analyzes each frame and attempts to find harmonic structure among the frequencies present. It does this through an iterative process of identifying peaks in the |ODFT| output and looking for integer relations between them. Spectral peaks are identified by iterating through the |ODFT| output and looking at the current value, the previous value, and the next value. If the current data point is greater than the previous value by some threshold and lower than the next value by some threshold, a spectral peak has been found. The fundamental frequency of each harmonic is determined by looking at the ODFT output frequency and its corresponding fractional frequency (interpolated from values to the left and right of the peak). Each fractional frequency is calculated by taking advantage of the sine windowing done in the ODFT. The fundamental frequency in each frame is identified by

the ODFT output plus the fractional frequency identified multiplied by 2PI divided by the ODFT size. The fractional frequencies are identified as follows:

$$\Delta\ell \simeq \frac{3}{\pi}\arctan\frac{\sqrt{3}}{1+2\left[\frac{|X_O(\ell-1)|}{|X_O(\ell+1)|}\right]^{1/G}}$$

Fig. 4: Frequency interpolation function

where each Xo is a point from the ODFT output. Intuitively a higher fractional frequency is generated if the relative weighting between the next value of the ODFT and the previous value of the ODFT is high, and a smaller fractional frequency is generated if the ratio is low.

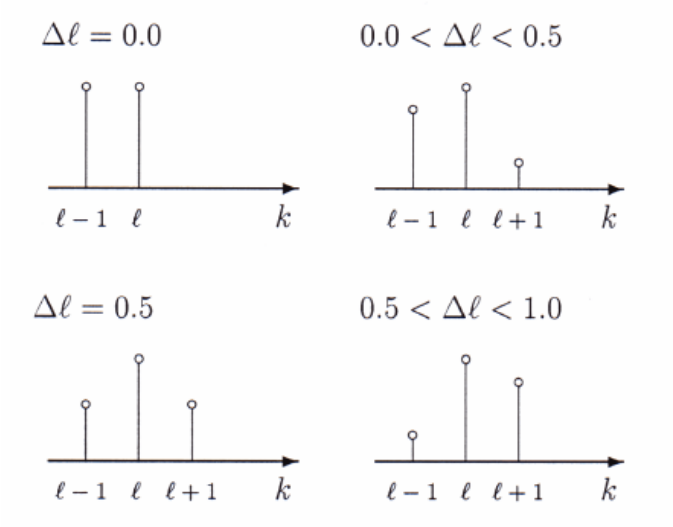

Fig. 5: Frequency interpolation example situations

Integer analysis performed on the fundamental frequencies identified by this process results in harmonic structures being identified encompassing some of the higher frequency peaks to the right of the fundamental, which are called the partial frequencies. Each harmonic structure in a frame returns the fundamental frequency and a power value for that harmonic.

The next step is to track the harmonic structures over time in an attempt to identify notes. The harmonic structures are tracked by the formation and tracking of "trajectories". Each trajectory is comprised of a start frame, stop frame, vector of fundamental frequencies, vector of power, and interpolation vector. There are two lists of trajectories: the first contains the candidate trajectories and the second, the validated trajectories. In every frame, we take all of the harmonic structures that are found and append

> TRAJECTORY	
START FRAME	
STOP FRAME	DUR = STOPFRAME - STARTFRAME + 1
[F0(1), F0(2),...,F0(DUR)]	F0 = FUNDAMENTAL FREQUENCY VECTOR
[P(1), P(2),..., P(DUR-INTERPOLS)]	P = POWER VECTOR
[INTERP(1),...,INTERP(INTERPOLS)]	INTERPOLS = NR. OF INTERPOLATED GAPS

Fig. 6: Trajectory data structure

them to a candidate trajectory and if no candidate trajectory is found, a new one is created. Assignment of a harmonic structure to a candidate trajectory is based on the fundamental frequency of the structure. If the fundamental frequency of a harmonic structure falls within some error range of the fundamental frequency of a candidate trajectory, the harmonic structure is appended to that candidate trajectory by adjusting the stop frame and adding the fundamental frequency and power value of the harmonic structure to the frequency and power vectors in the candidate trajectory. If a harmonic structure matching a candidate trajectory cannot be found in a given frame, the candidate trajectory ends and becomes validated. However, if the candidate trajectory is smaller than the smallest possible note length (user defined), it is thrown out. Additionally, it is possible, in a real music sample, for a harmonic structure to be absent from a frame that it should be in. This can lead to the premature termination of candidate trajectories. To correct for this, candidate trajectories are not ended until a minimum pause length (absence of that fundamental frequency) has been exceeded. When such a gap occurs, the missing frames are added to the interpolated list of the trajectory and are used in off-line processing.

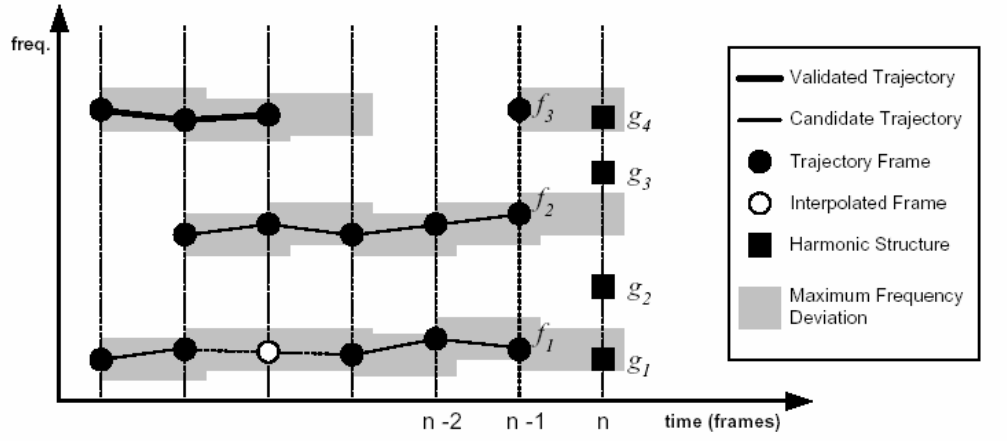

Fig. 7: Harmonic structure tracking

At this point in the algorithm, we have a list of notes that we believe are present in each frame grouped together into trajectories. From these trajectories, it is relatively easy to infer which notes are present, their length and location in time, and their relative loudness. Just from this information we are be able to obtain reasonable results for a monophonic input. However, for polyphonic sound, there will be false notes that need to be eliminated by further processing of the data.

Off-line Processing:

The off-line (post) processing consists of three sub-stages: transient detection, trajectory on-set time adjust, and trajectory clustering and pruning.

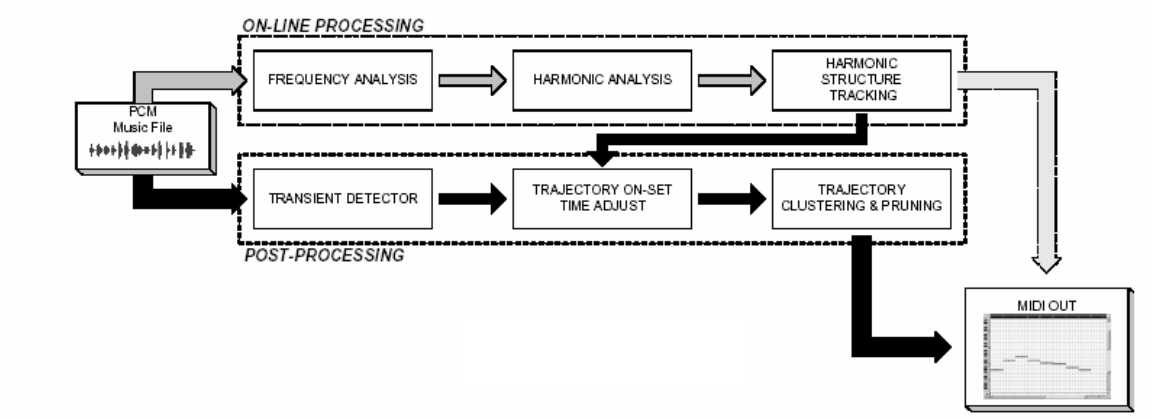

Fig. 8: On-line and Off-line (Post) processing flow

The off-line processing portion of the algorithm operates over the entire input signal in the time domain. The first stage of this process is transience detection. This procedure makes an attempt to identify the onset of musical notes through energy analysis. The first step in this process is applying a high pass filter to the input data. The coefficients we used for this filter are the same as given in [2]. The summation of the filtered data over each frame is then taken. Analysis is then performed on the summation of a frame and its previous frame. If it meets the criteria below, a spectral peak is identified:

$$E(s) = \sum_{k=1}^{N/2} \left| f\left(k + \tfrac{N}{2} s\right) \right|, \quad s = 1, \ldots, ns$$

$$\log_{10}\left(\frac{E(s)+1}{E(s-1)+1}\right) > transthres$$

(Where f is the filtered input data, ns is the number of frames, and N is the ODFT size)

Eq. 2, 3: Transient detection functions

Once the transience is established, we are able to adjust the validated trajectories. A few cases can result from this: the onset (start times) of the trajectories can be moved forwards or backwards to bring them in line with the transient if it is within a certain tolerance. It is also possible to break a validated trajectory into separate notes, if they would be of valid duration, when a transient coincides with interpolated frames.

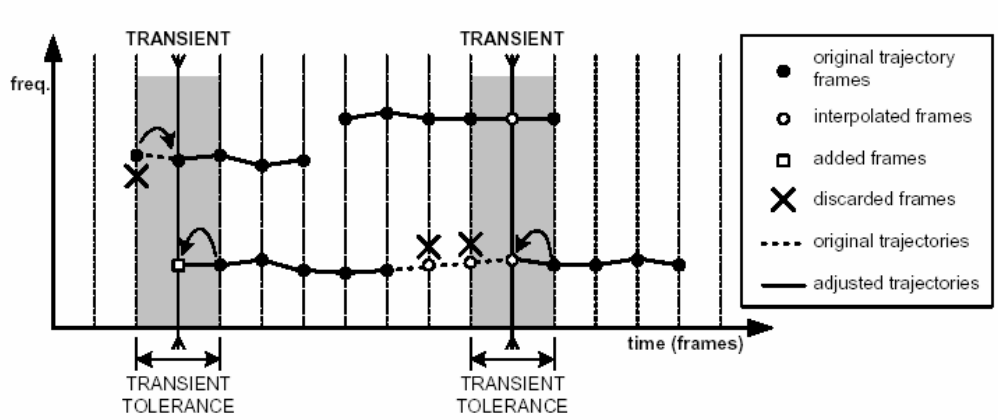

Fig. 9: Trajectory on-set adjust and splitting

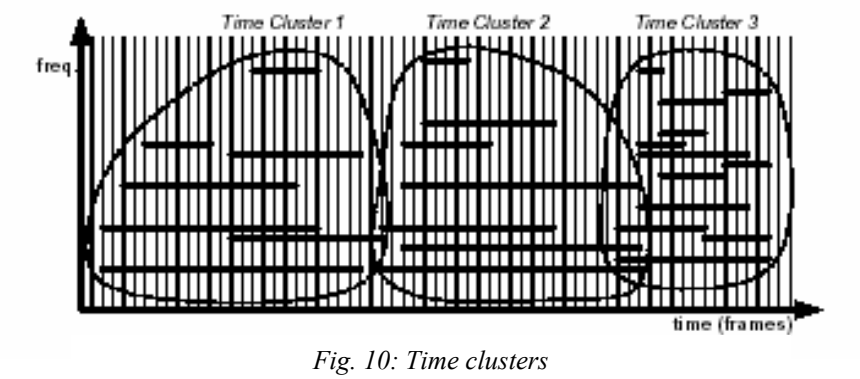

Fig. 10: Time clusters

The next step in the algorithm is to organize the trajectories into time clusters. The time clusters are formed by identifying the longest trajectories and then grouping shorter trajectories that occur simultaneously with them into the same cluster (see picture to right). This is done by analyzing the start and stop times of each trajectory and dynamically keeping track of the longest trajectory in each cluster. A small tolerance of overlap is granted for beginning and ending time clusters. Also, an improvement we made over the original Ferreira algorithm was to add an extra criteria for putting short (probably false) trajectories into surrounding time clusters in certain cases so that they would not falsely generate their own time cluster (in which case their elimination would be impossible).

Within each time cluster, we find harmonic clusters by grouping the trajectories that are separated by octave intervals (power of two multiples). Once the harmonic clusters are defined, we can attempt to eliminate false notes by looking at the power and duration of each trajectory. In each harmonic cluster, we evaluate which trajectory is the strongest (based on average power from the power vector, and duration). We also specify the relative weighting between the power and duration considerations in the calculation on the strength of a trajectory. We found that much better results were found when we considered the power to be relatively more important than the duration. The strongest trajectory is accepted as a valid note. Trajectories that fall within a certain tolerance range of the strongest will also be accepted as valid notes, while those that do not fall within this range will be eliminated. This ensures that at least one trajectory will remain from each harmonic cluster.

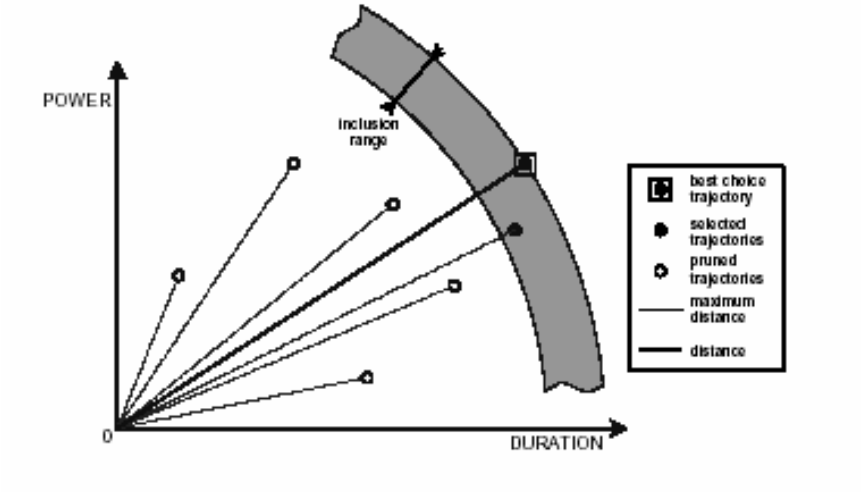

Fig. 11: Trajectory pruning

We then compare the remaining trajectories inside each time cluster, looking for weak trajectories that escaped harmonic pruning. The same elimination procedure (based on power and duration) with different thresholds is then applied to the remaining trajectories to remove any remaining false notes.

The result of the off-line processing should be the elimination of false notes while maintaining valid notes. Additionally, trajectory analysis should have increased the accuracy of the start and stop times of the individual notes.

What We Did

System Overview

Input to the system is a PCM digital audio file sent from the PC to the EVM over HPI one frame at a time. All online processing steps are then performed on the C67 and the results (trajectory structures) are sent back to the PC for the offline processing stages of the algorithm. We choose to structure our resources this way because the C67 could potentially be used to process the input data in real time from a musical instrument. The offline processing, however, still could not be done in real time because it requires information over the entire length of the input.

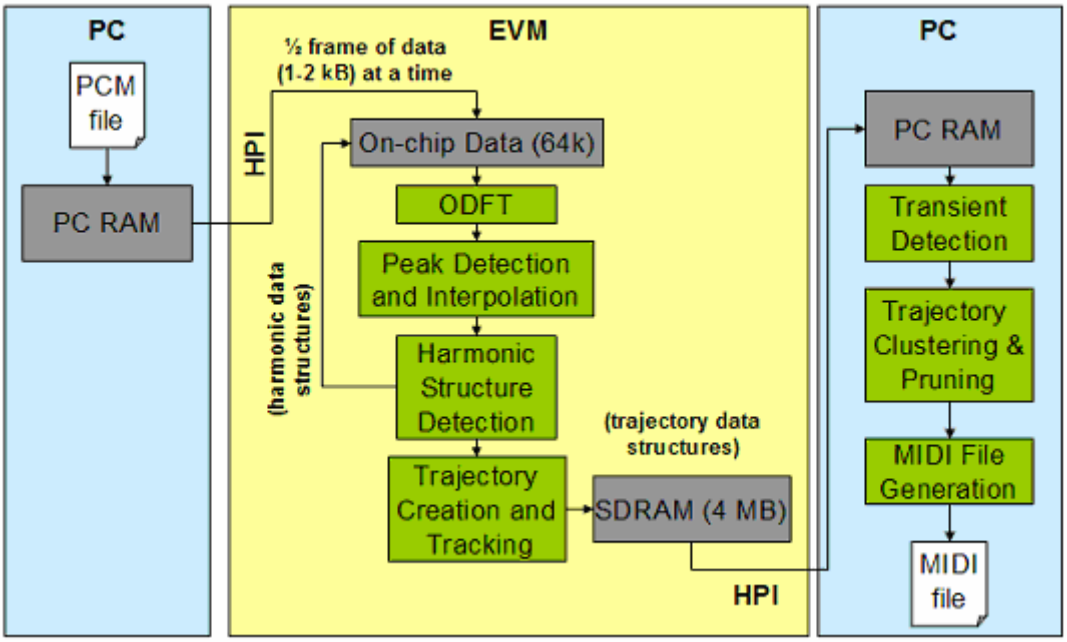

Fig. 12: Partitioning of functionality

Memory Usage

Great care was taken in writing the on-line processing in order to make all of the code fit into on-chip program memory and all of the frequently used buffers into on-chip data memory. We had to use the heap (in SDRAM) in order to store our dynamically allocated trajectory data structures, but everything else was made to fit into on-chip data.

Memory	Usage
On-Chip Program	44960 bytes (68.6 %)
On-Chip Data	58856 bytes (89.8 %)
SDRAM 0	4 MB allocated for heap (100%)

Table 1: Memory usage

Performance

After initially getting the program working on the EVM, its performance was reasonable; but not even close to real-time. We made a series of improvements that resulted in the program transcribing pieces in less time than it took to listen to them. Although a real-time implementation was not one of our original goals, it appears that our project could perform well enough to make this happen.

The first step we took was to remove all of the 'printf' and other 'stdio' functions from our code. This made it possible to fit the program itself in on-chip memory. We next instructed the compiler to perform optimizations, including inner loop un-rolling. Both of these steps produced improvements, but the execution time, especially for longer files was still much worse than real-time.

We next attempted to replace our radix-2 FFT (written in C) with the assembly radix-4 FFT from TI. This was ineffective because we commonly use a frame size of 2048 (which is not a power of 4). The radix-4 FFT would require us to use 4096 instead, and we simply could not afford to use that much more memory.

Finally, we rewrote many of our functions (ODFT and peak detection) in order to make them reuse the same buffer for everything. This required a little bit of tricky coding: even though the input PCM data samples were all shorts (2 bytes), we stored them in a long integer (4 byte) array so that we could later treat this same buffer as an array of floats (4 bytes). All of these modifications allowed us to vastly reduce the size and amount of global variables and fit everything into on-chip memory (rather than having to use DMAs/paging).

The speed improvement from this was, as expected, enormous. The program can now read the file, perform both the on-line and off-line processing, and write out a MIDI file in less time than it takes to listen to the input file.

	Program execution (s)	
Input Length (s)	Before	After
30	382	22
28	348	21
10	74	6

Table 2: Run-time performance improvement

Available Software

We were able to obtain the Matlab code from Ferreira and Martins (through email correspondence) for their implementation of the algorithm. The harmonic structure detection code from this, however, was a precompiled C file that we could not see the source for because they are in the process of developing a commercial product. We generally found the Matlab code to be difficult to read and understand. Because of this,

and the fact that the MSc thesis [2] was quite clear and detailed, we decided to develop our own C code without trying to follow the Matlab implementation. The Matlab implementation, though, did prove helpful in giving us something to check our results against. It was particularly useful when evaluating our ODFT results for validity, but used little else until the final comparison of MIDI outputs. Thus, with the exception of the FFT code and some of the EVM setup code, every line written is original code developed by us, based on our interpretation of [2] and our modifications to it.

Results and Analysis

ODFT Performance

The first crucial stage in our project was constructing the ODFT and correctly identifying spectral peaks. We evaluated the results of the ODFT by comparing it with Martins' Matlab implementation of the PCM to MIDI algorithm:

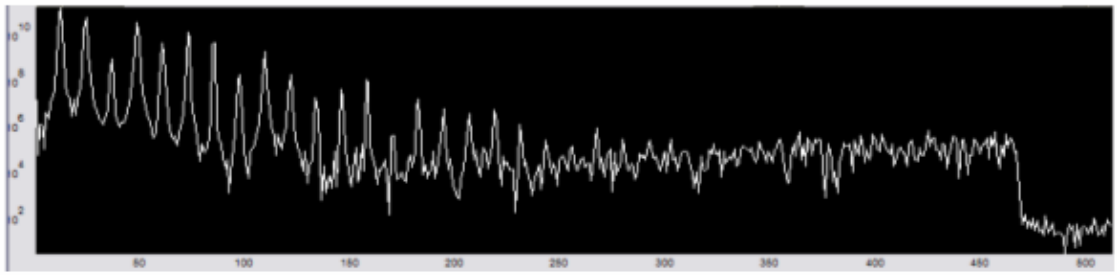

Fig. 13: Martins |ODFT|

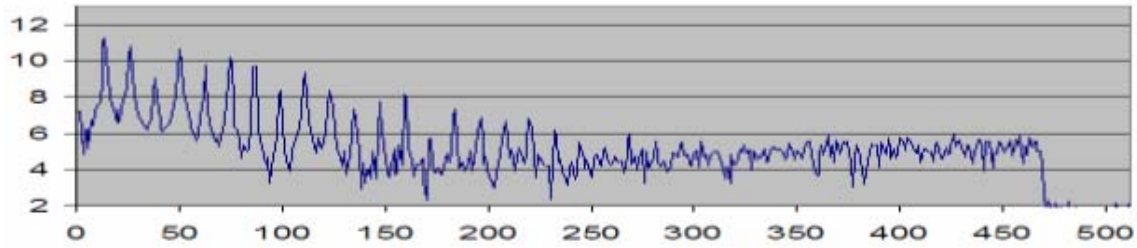

Fig. 14: Group 6 |ODFT|

As can be seen from the two graphs above, our |ODFT| output is identical to the Martins |ODFT| output. There arose several complications in developing the ODFT mostly resulting from a change in endian notation from the PC to UNIX and Martins' zero-padding the PCM input file without noting it in his paper.

Once the ODFT was verified, the spectral peaks had to be identified. A comparison between the Martins implementation and our implementation was again made:

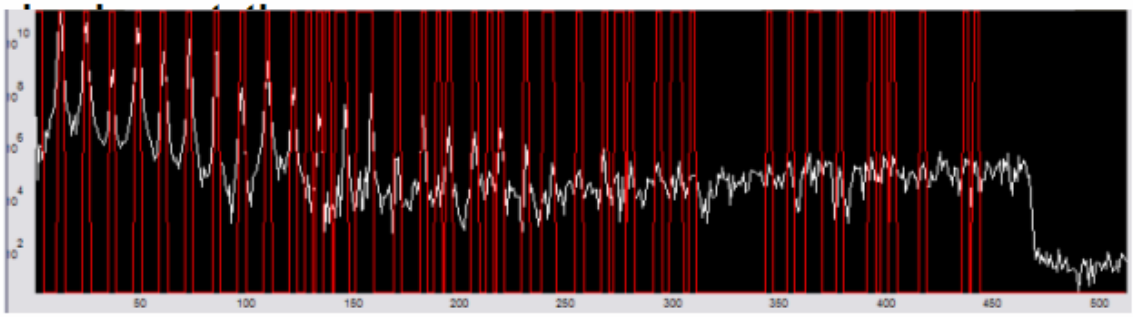

Fig. 15: Martins peak detection

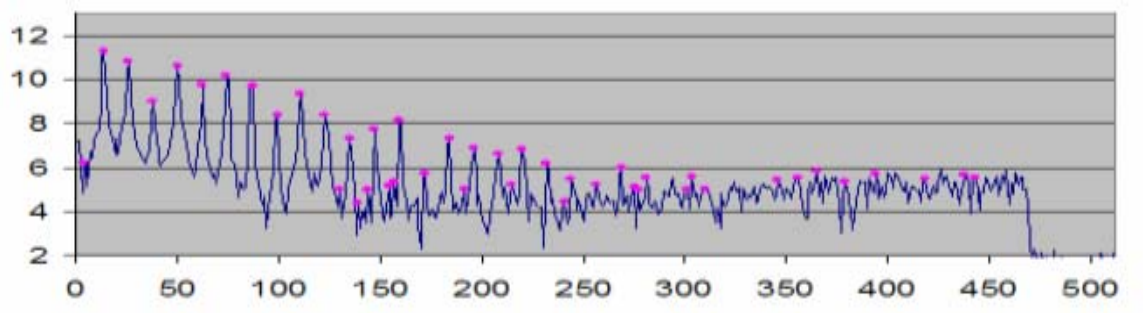

Fig. 16: Group 6 peak detection

As can be seen from the above graph, our peak detection correctly identifies spectral peaks and is very similar to Martins. Differences in the number of peaks detected can be accounted for by adjusting the peak detection threshold.

Range and Resolution

We have used frame/ODFT sizes of 1024 and 2048 while testing the program. We switch between the two different sizes due to tradeoffs between time and frequency resolution:

N	Freq. Res. (Hz)	Time Res. (s)
1024	43.1	0.02
2048	21.5	0.05

Table 3: ODFT frame size effects

With the appropriate frame size selected, the program is capable of resolving notes from about 60 Hz (MIDI octave 3) all the way up to about 4 kHz (MIDI octave 8). Lower frequency notes are more difficult to detect due to the fact that the differences between note frequencies become very small in the low octaves. Therefore, a frame size of at least 2048 is always necessary for notes below about 260 Hz. For higher frequency

notes, fine frequency resolution becomes less important and either frame size can be used. However, on certain files, the program performs better with worse time resolution because it will not detect as many false notes, improving the performance of the off-line processing. This behavior varies depending on the idiosyncrasies of the individual file, so, if we had trouble getting good results with one frame size, we simply switched to the other.

Final Demo

For our final demonstration, we created a graphical user interface program to make it easy for a user to execute our program on the EVM and view the transcription output. The program has a simple but very useful interface and was written in Java.

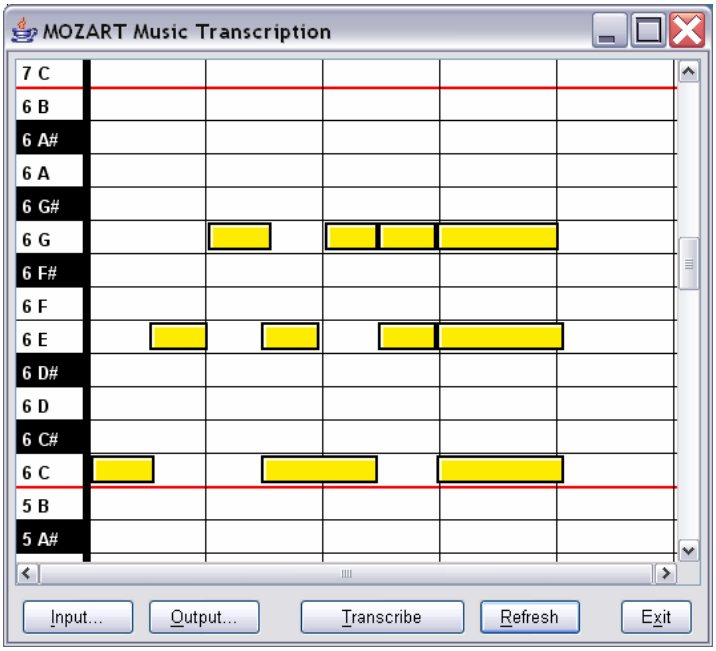

Fig. 17: MOZART GUI screen shot

To use the program, the user simply clicks the 'Input…' button, and selects a PCM file from the file selection dialog box that pops up. Next, the 'Output…' button is clicked and a new or existing MIDI file is selected. Clicking the 'Transcribe' button starts the C++ program that loads the EVM program and does the actual transcription. Once that process terminates, the Java program automatically updates its "piano roll" display of the output file's content. The piano roll shows pitch on the y-axis and time on the x-axis. The yellow bars represent the transcribed notes in the MIDI file. The vertical black bars denote quarter-note durations, and the horizontal red bars denote octave intervals.

Input Files

Single channel (mono) PCM audio files with 16-bit resolution and a 44.1 kHz sampling rate were used as input. These are essentially the same as a WAV audio file, but with the descriptive headers removed from the front of the file, so that it just contains the raw sampled data.

We tested recordings produced by a fairly high-quality wavetable synthesis of original MIDI files. This made it easy to compare the "before" and "after" piano rolls to determine the accuracy of our program. Wavetable synthesis uses samples of real instruments to accurately reproduce their harmonic, resonant, and noise characteristics. Instruments tested included grand piano, flute, organ, recorder, and violin. The program does not try to determine what kind of instrument it is transcribing, and seems to generally perform well regardless of what instrument we selected.

Monophonic Sound

Our project performs extremely well (sometimes perfectly) on monophonic sound. Because we know up-front that only one note will be played at a time, the program can be instructed to detect more false notes in order to make certain that all the real notes are found. During post-processing, it is quite easy to determine which one note is the real one. However, the program most often did not detect any false notes at all in monophonic files and the trajectory pruning during off-line processing was unnecessary.

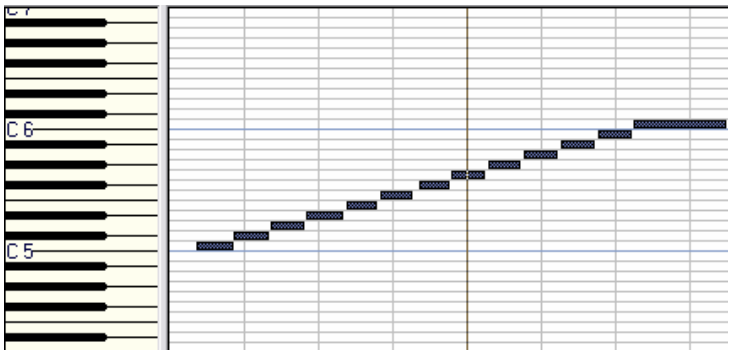

Fig. 18: Original Chromatic scale

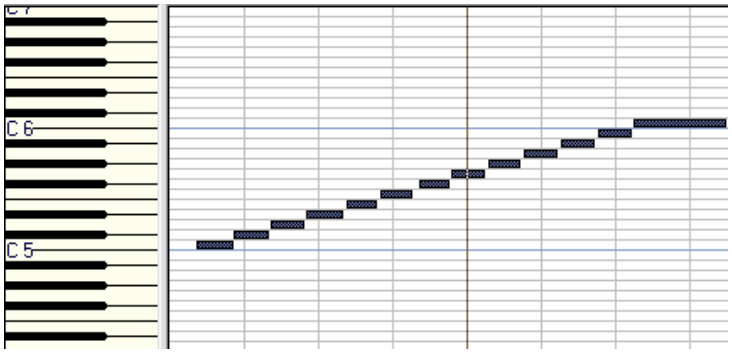

Fig. 19: Transcribed Chromatic scale (no errors)

Polyphonic Sound

While the project could not achieve the same level of accuracy for polyphonic music, it does do a very good job transcribing it as well. Selections that do not have a wide octave range perform best, but the program has handled pieces spanning three octaves and with multiple tempo changes with very reasonable results. The off-line processing, especially the trajectory cluster and pruning, make this degree of success possible.

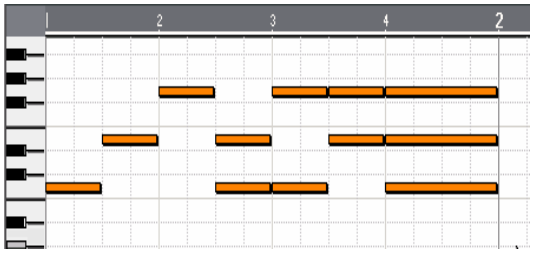

Fig. 20: Original arpeggio

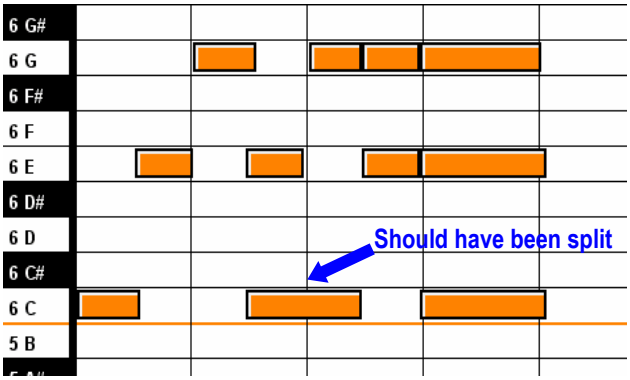

Fig. 21: Transcribed arpeggio (one missing split, no pitch errors)

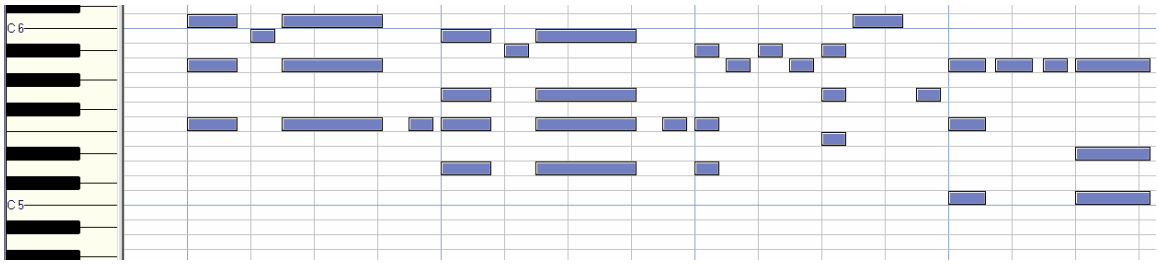

Fig. 22: Original "Family Guy" theme

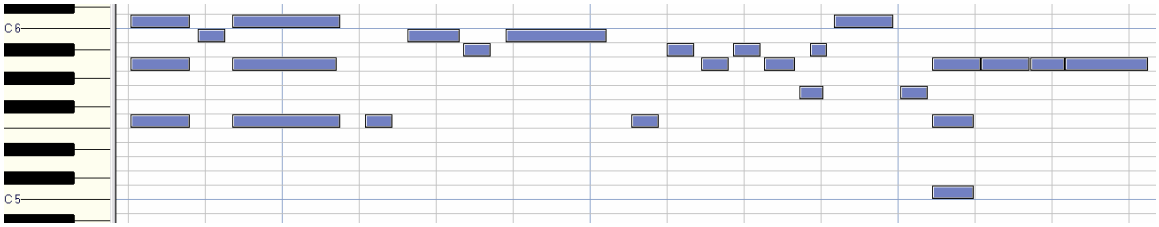

Fig. 22: Transcribed "Family Guy" theme

The most noticeable difference between the original and transcribed versions of the "Family Guy" theme are the missing trajectories that would represent some of the lower notes in a few of the chords. Generally speaking, lower notes are harder to detect because they require a higher frequency resolution. This can result in them not being detected during on-line processing, or, more likely, being mistaken for false notes and pruned during off-line processing due to the presence of more easily detected (and therefore more powerful) notes in the same harmonic/time cluster. Additional tweaking of the pruning settings may have made a better result possible, but it is not always practical to do this level of tuning for such a large input file, especially when the overall result is still quite good. Unfortunately, program settings that work perfectly for one part of a song may produce undesirable results in a different part.

Conclusions

The program is highly configurable, which is both a blessing and a curse. We have empirically developed a few sets of configuration parameters that work well for certain types of files, but the program will often require tuning by the user if it is to perform optimally. Fortunately, simply telling the program: 1. Whether the input is mono or polyphonic, 2. What the high and low octaves are, and 3. Whether a 1024 or 2048 frame size should be used, is more than enough to make it perform very well on any input file.

One limitation that would be easy to address given additional time would be the way in which trajectory data is returned to the PC from the EVM. Currently, the validated trajectories are stored on the EVM until on-line processing is complete. It would make more sense to transfer these back incrementally in order to prevent the heap from completely filling up during the processing of very large files.

Overall, we are extremely pleased with the performance of the project. We set out with the goal of transcribing polyphonic music produced by realistic instruments and without any gaps between the notes. Our program accepts realistic input files, processes them very quickly (faster than it takes to play them), and produces output that is much better than the two commercial transcription products that we tested. The program can often produce flawless transcriptions of monophonic audio files, and does a reasonably good job even on complicated polyphonic pieces. Even on music files where the piano roll shows many errors, the re-synthesized output MIDI usually sounds quite close to the original input, which means that it could be very useful in assisting a human doing a manual transcription.

References

[1] Luis Gustavo P.M. Martins and Anibal J.S. Ferreira. "PCM to MIDI Transposition". Audio Engineering Society, 112th Convention, May, 2002, Munich, Germany.

[2] Luis Gustavo P.M. Martins. "PCM to MIDI Transposition". MSc Thesis, Department of Electrical and Computer Engineering, Universidade do Porto, Portugal, 2001.

[3] Anibal J.S. Ferreira. "Spectral Coding and Post-Processing of High Quality Audio". Ph.D. thesis, Department of Electrical and Computer Engineering, Universidade do Porto, Portugal, 1998.

[4] Anibal J.S. Ferreira. "Accurate Estimation in the ODFT Domain of the Frequency, Phase and Magnitude of Stationary Sinusoids". 2001 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, 21-24 October 2001, New Paltz, NY.