SILENCE OF THE LAMBS

18-551 Digital Communications and Signal Processing Systems Design

GROUP 7

James Chan (chan6@andrew.cmu.edu)
Daniel Chen (dtchen@andrew.cmu.edu)
Ri Muttukumaru (rim84@cmu.edu)

SPRING 2005

TABLE OF CONTENTS

INTRODUCTION 3
Background 3
Objective 4
Complexity of our project 4
How Our Project Differs from Prior Projects 5
PROJECT SPECIFICS 6
System Overview 6
C/EVM Resources Available 7
Algorithms Used 8
L-R Center Channel Cancellation 11
Harmonic Product Spectrum (HPS) 12
Comb Filtering 15
Chen-Muttukumaru (C-M) Algorithm 16
RESULTS AND ANALYSES 19
Test signals used 19
Problems encountered, Solutions Adopted 20
EVM Memory Issues/Analyses 20
Code size/Memory Allocation 20
Optimizations Performed 21
Speed Analysis 22
GUI Design and Implementation 23
Discussion of Results of Our System 24
SUMMARY 26
REFERENCES 27
Content References 27
Code References 28

INTRODUCTION

Background

The problem of *auditory source separation* has intrigued and perplexed researchers and engineers since the mid-nineteenth century. Auditory source separation is defined as the separation of a polyphonic audio stream into its component sources. For instance, Billy Joel's "Piano Man" can be auditory-source-separated into Billy Joel's vocals and the accompanying piano instrumental.

The first researcher to uncover the proverbial tip of the iceberg was famous psychophysicist and physicist Hermann von Helmhotz, who was concerned about the grouping of harmonics in complex piano tones into coherent percepts[1]. The term *auditory source separation* was only derived much later on, in 1990 from Bregman's work on *computational auditory scene analysis* (CASA)[2].

This problem remains unsolved even today. Despite the lack of success, a multitude of interesting approaches have proposed - spatial, periodicity and harmonicity, physiological. For instance, substantial amount of research has been placed into figuring out how human hearing works in an attempt to apply knowledge gleaned onto auditory source separation systems - and not without good reason. Our ears perform the job of auditory source separation perfectly. A person can sit at a cafe table by the road, notice the Doppler effect of a passing car, the honk of the school bus and yet continue to track his friends' conversations across the table. Similarly, in the context of our project, a person is able to listen to a song and focus on the singer - his vocal inflections and other vocal subtleties.

Applications for a system capable of performing perfect auditory source separation are mind-boggling. It can be used in battlefield systems, to assist military personnel in conducting military actions in battlefield environments, "intelligent-room systems" to facilitate interactions between people or the isolation of specific audio sources in a noisy environment, such as brokers at a stock exchange. More consumer-level applications might include the ability for your average Joe to create karaoke tracks to sing-along to at home, or high-fidelity elimination of a particular instrument accompaniment so that the user can practice from the comforts of the home.

---

[1] Objects of perception

[2] *Instantaneous and Frequency-Warped Signal Processing Techniques for Auditory Source Separation*, Avery Wang, CCMRA Stanford, 1994

Objective

In this project, our group sought to devise an auditory source separation algorithm, as an eventual step towards consumer-level real-world application. We aim to design a system that can remove musical accompaniment from a polyphonic audio track, leaving behind only the vocals. One can imagine the functionality of our system as the inverse of some of the consumer-grade vocal removers available on the market, such as the ALESIS Playmate[3]. In those devices, a song is fed into the device, and the output yields an audio track with vocals removed. However, these devices either have a large variance in quality depending on the song, or require oracle knowledge in the form of input from the user[4] to obtain decent-quality output.

This is where our project tries to take this process one step further. We have designed a system that does not make use of oracle knowledge in our process to eliminate the accompanying music. By oracle knowledge, we mean specifics about the audio track, such as the exact positioning of instruments at the time of recording, musical score, genre of music and so on. While we are not eliminating vocals, the nature of music leads us to believe success in background music removal will translate into success in vocal removal.

Complexity of our project

The complexity of our project is quite high - for starters, there is still no reliable and accurate way to distinguish between the frequency components of voice and music when everything else is held equal. There is also no academic consensus as to how background music removal or vocal removal can be accomplished. These were established after several weeks of intensive research going through publications from established sources such as CCRMA of Stanford and Audio Engineering IEEE, as well as from our consultations with Professor Tom Sullivan and Professor Richard Stern here at Carnegie Mellon.

---

[3] As far as we can tell, the ALESIS Playmate performs L-R center channel elimination and cleans up the resultant signal with a variety of reverb-cancellation algorithms. The ALESIS Playmate product description can be found at

[4] Such as genre of songs, male/female vocals, e.t.c.

Although we found a detailed research thesis by Avery Wang of Stanford titled *Instantaneous and Frequency-Warped Signal Processing Techniques for Auditory Source Separation* that generated very convincing results even when separating a tenor-soprano-accompanying music piece into any of its components, the algorithm was judged to be too complicated to be implemented in our case, considering we have to work with the hardware limitations of the Texas Instruments EVM DSP board.

In the end, we decided to formulate our own algorithms based upon our own understanding and research. We also chose to reduce the scope of our project, by designing an algorithm to work only for instances in which there is **a single instrument and a single vocal**. As mentioned before, this is because as of now, **there is no way to distinguish between pitches of instruments in multi-instrument mixes without some form of oracle knowledge**, such as having access to the music scores of each instrument in the music piece. We also assume ideal harmonicity in the instruments that we pitch track, so as to further reduce the complexity of our problem, and avoid handling scenarios such as non-harmonic or in-harmonic music.

How Our Project Differs from Prior Projects

Although auditory source separation has been a topic examined by previous project groups, no previous projects ever made vocal removal their main focus.

"*Enhanced Karaoke*" (Spring '99) had an aspect of voice removal. However, it was only a minor portion of their project. They performed the L-R center channel cancellation as part of the process leading up to their pitch correction algorithm.

We also identified "*Making Mozart Zippy*" (Spring '04) as a past project that was similar in nature to ours. While pitch tracking was a significant portion of their project, music removal was not their intention. Instead, the project transcribes polyphonic music into notes, which essentially was an automatic music transcription system that attempts to transcribe all musical notes of a piece into a score . While their pitch tracking algorithm was superb, their pitch tracking algorithm would be useless in our case, since we would not be able to distinguish between the pitches of instruments and vocals anyway. Without the ability to distinguish between the pitches of instruments and vocals, background music or vocal removal would be unattainable. Once again, nothing useful could be found from past projects.

PROJECT SPECIFICS

System Overview

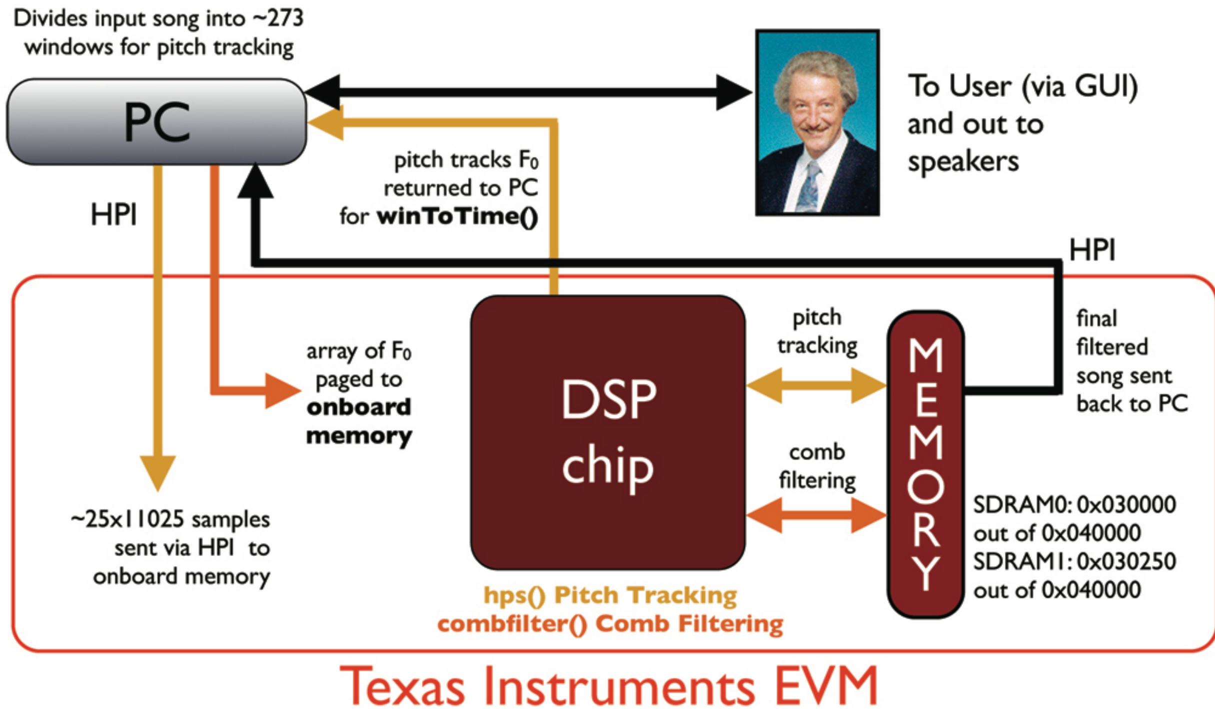

Figure 1: Block Overview of our System

The above diagram gives an overview of our entire system. Our group has assigned each part of the system to either the EVM or the PC as follows:

FUNCTIONALITY	DESCRIPTION	LOCATION
Pitch Tracking (Harmonic Product Spectrum)	Provides system with estimated pitch trajectory of musical instrument	EVM
FFT	Radix-2 STFT to aid in the pitch tracking process	EVM
Window-to-Time	Converts the original form of pitch estimates obtained from the EVM (in windows) into time-domain (in seconds)	PC

FUNCTIONALITY	DESCRIPTION	LOCATION
Comb Filter	Performs periodic 'notch' comb filtering at integral multiples of the estimated fundamental frequency of the musical instrument	EVM
C-M algorithm (Chen-Muttukumaru algorithm)	Post-processing technique to enable us to identify portions of music tracks without vocals	PC
GUI	Graphical user interface to allow users to load a wave file and apply our algorithm to obtain an output file for playback	PC

Details of each aspect of the system will be discussed in further detail in the later sections. The Harmonic Product Spectrum (HPS), Comb Filtering and C-M algorithm will be discussed under *Algorithms Used*, while the GUI will be discussed under the section *GUI Design and Development*.

C/EVM Resources Available

We were unable to locate significant C code or EVM resources relevant to our project. The only C code we found was one for STFT[5] - the assembly version provided to us during labs, as well as the FFT function we wrote for our homework were not used because they gave slightly different results from our MATLAB values.

No EVM resources were available. Perhaps the only other useful resource we found online was the MATLAB code for the pitch tracking algorithm[6]. We later based the C code for our pitch tracking off the MATLAB code. The comb filtering code was produced after conferring with Professor Richard Stern and Rajeev Ghandi.

In short, no code could be used wholesale, and significantly effort was put into understanding and writing our own code for the final product on the PC and EVM.

[5] Radix-2 (decimation-in-time) STFT C code, http://cnx.rice.edu/content/m12016/latest

[6] *Pitch Detection Algorithms*, http://cnx.rice.edu/content/m11714/latest, Connexions

Algorithms Used

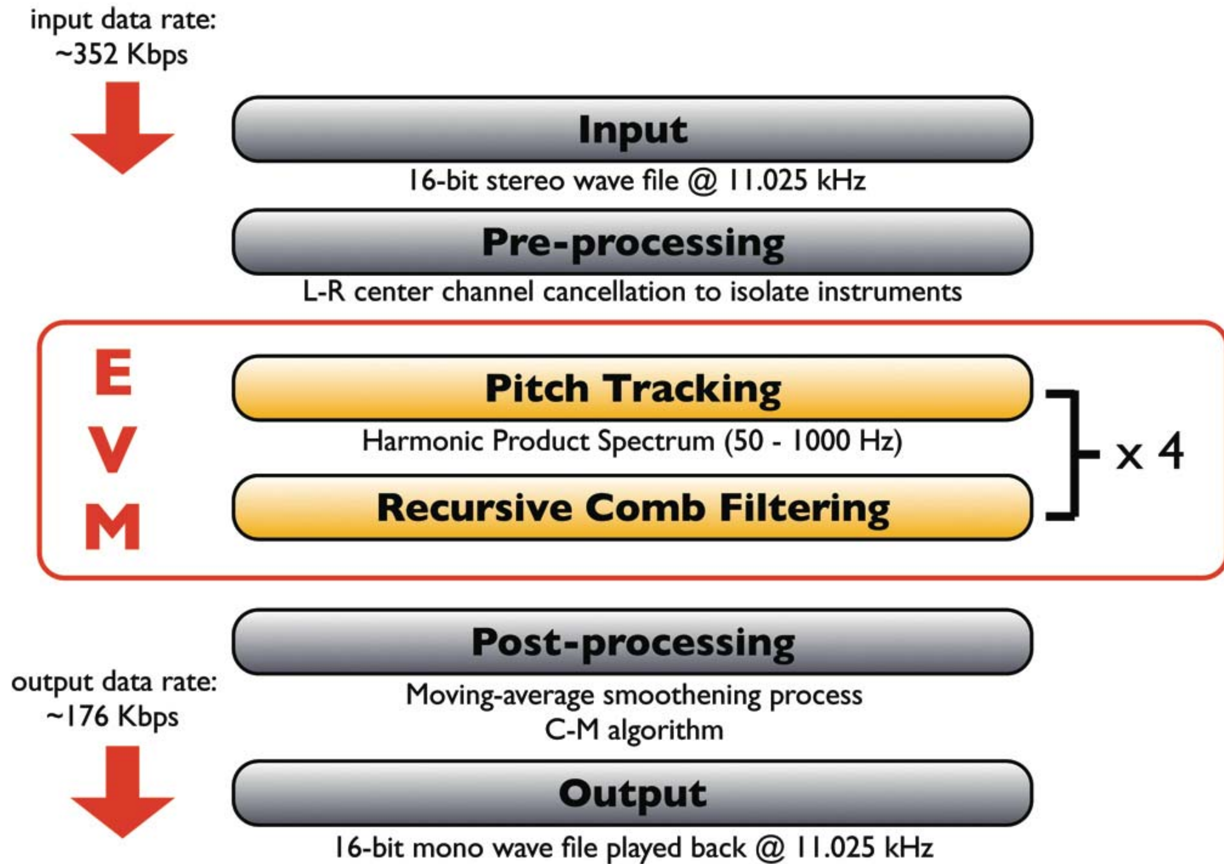

Figure 2: Algorithm Flow for our System

The figure above gives an overview of the algorithm we employed in our project. The input data rate of 352 Kbps comes from a 16-bit stereo wave file sampled at 11.025 KHz - 16*2*11025 = 352,800 bits. The output data rate is halved since output is mono. On a cursory glance, we can identify the following 4 major components:

1. L-R center channel cancellation
2. Harmonic Product Spectrum
3. Comb Filtering
4. C-M (Chen-Muttukumaru) algorithm

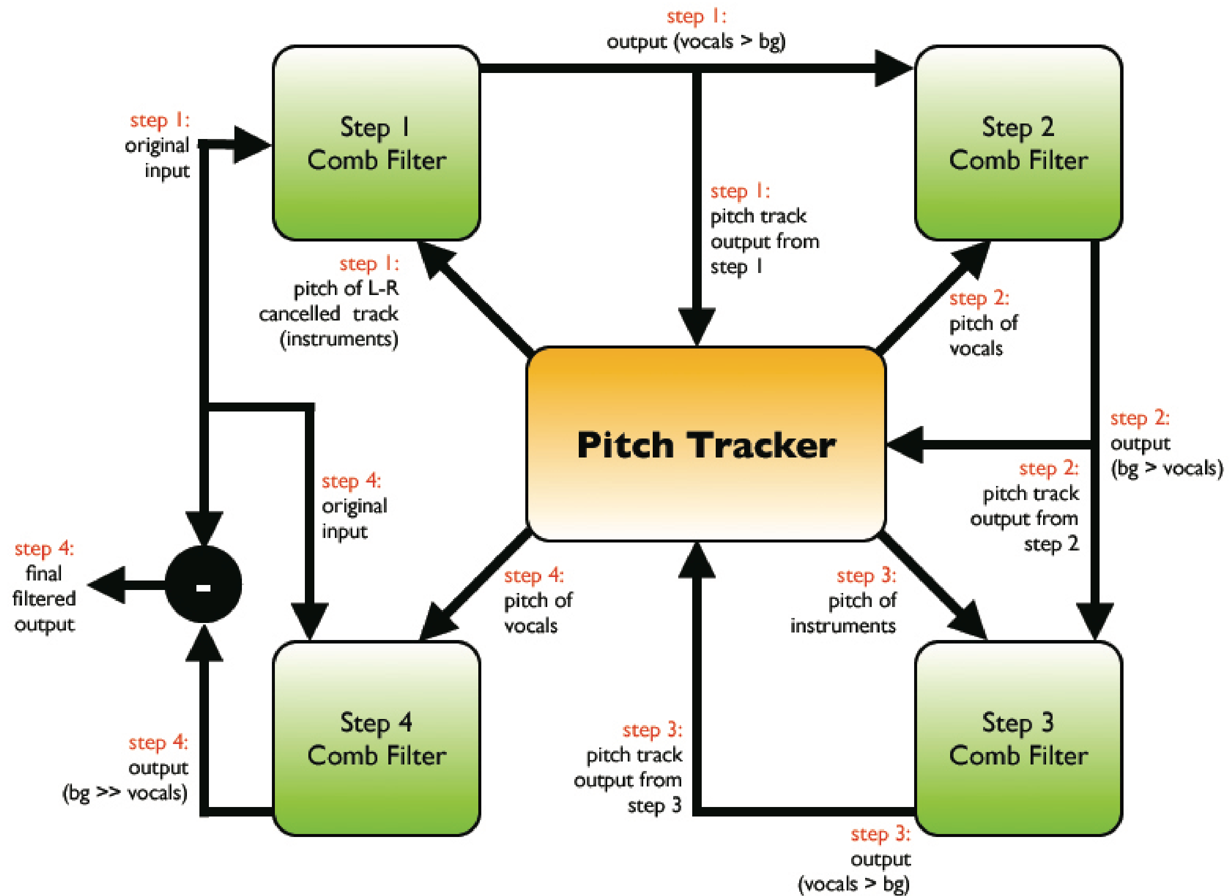

Figure 3: Flow Diagram of Pitch Tracking/Filtering Process

The algorithm begins by performing L-R center channel cancellation on the original song. The L-R cancelled track is then pitch-tracked, and the pitch info for the instruments (which is more dominant than vocals) is sent together with the original song into comb filter for Step 1 filtering, yielding an output with more dominant vocals than instruments (vocals > bg). The output from Step 1 is then pitch-tracked again.

These pitch estimates of the vocals are sent into the comb filter in Step 2, where filtering is then applied on the output from step 1. The resultant track has more dominant instruments than vocals (bg > vocals). The output from Step 2 is then pitch-tracked again.

The pitch information from tracking Step 2's output is then sent in Step 3 into comb filter, where filtering is yet again applied to the output from Step 2. This yields an output with more dominant vocals than instruments (vocals > bg). The output is sent for a final time into the pitch tracker to give us the pitch of vocals. We have established through experimentation that this process gave us the most accurate pitch tracks for voice.

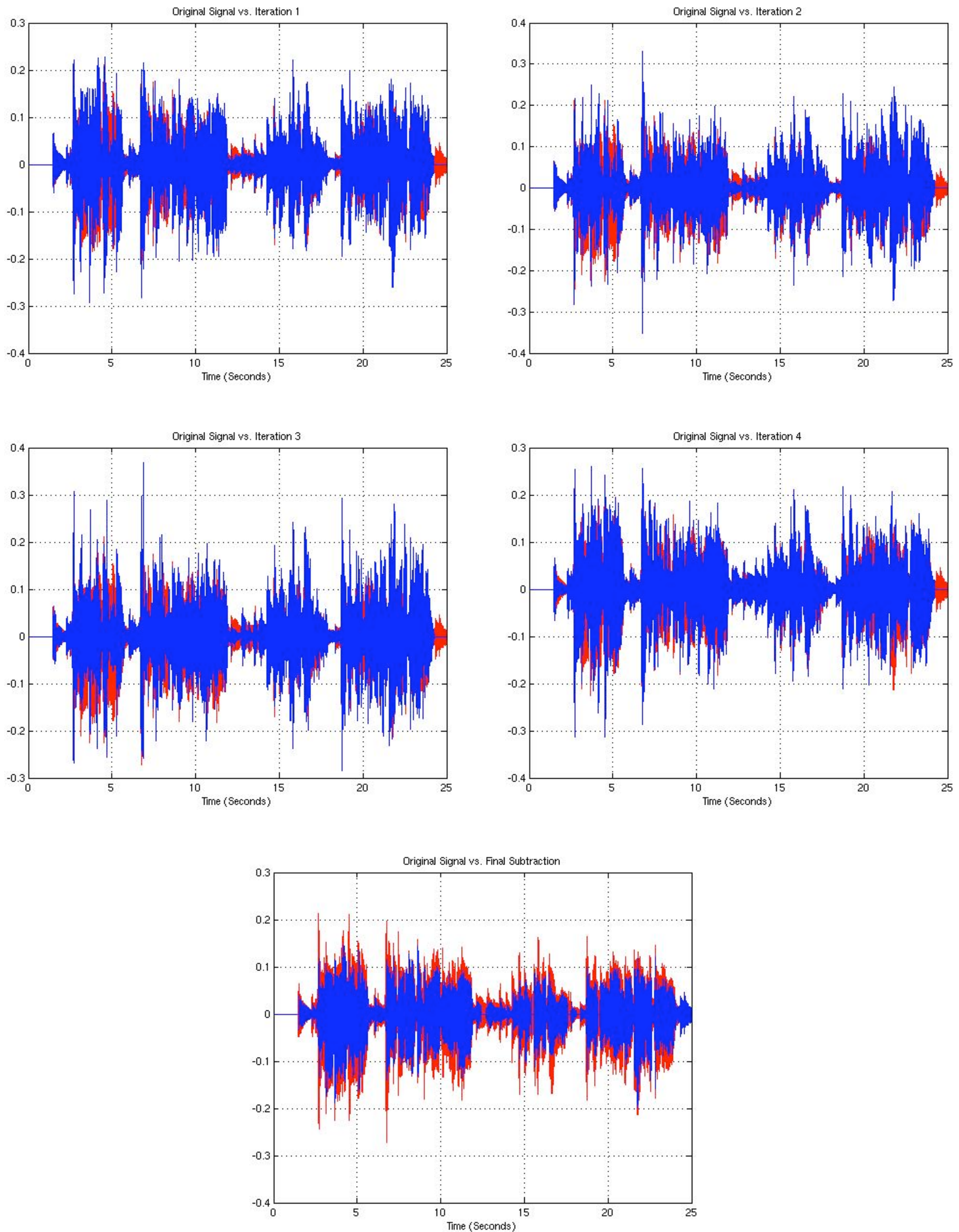

Figure 4: Comparative Time-domain plots at each step
*(**Red** =original input, **Blue** = filtered output at each step)*

In Step 4, the pitch estimates of vocals from tracking the output of Step 3 is used to yield an output with significantly more background instrumentation than vocals. This output is then subtracted against the original input signal to yield our final filtered output.

From the first 4 plots in Figure 4, it is not clear quantitatively what has changed as a result of the filtering. This is because it is difficult to quantify how much of the audio signal at one point in time is voice or background music. However, in comparing the fifth plot of the final signal against the original, there is an obvious and significant reduction in the audio track as a whole. When listening to this, it is clear that most of the reduction is, in fact, the background music being removed.

This process of repeated iterations of pitch tracking and comb filtering was developed during our initial MATLAB experimental phase. This concept was also partially inspired from an auditory source separation method we came across in *Multiple fundamental frequency estimation based on harmonicity and spectral smoothness* by A. P. Klapuri, which involves repeated pitch tracks and filtering. A single pitch track is performed for each instrument in the audio mix in Klapuri's algorithm, with the assumption that their particular pitch tracking algorithm would be able to pick out the dominant instrument each time round. To raise the amount of background music that was perceived to be removed, we developed the iterative process that was described in the previous paragraphs.

Each component of our algorithm will be discussed in detail in the sub-sections that follow.

L-R Center Channel Cancellation[7]

To obtain a sufficiently accurate estimate of the pitch of the vocals in a song, we chose to make use of the L-R Center Channel Elimination, which takes the left and right channel of a stereo channel and performs a subtraction between the two. The success of this algorithm relies on the assumption that the vocals are mixed equally between the two channels, while instruments are mixed slightly to either side, which is often the case for songs recorded in a recording studio. The difference from the subtraction yields a predominantly voice-removed track with which we can estimate instrumental pitch trajectories from.

We acknowledge that by performing such a subtraction, some instrumental energies are lost. This technique is also does not work for all songs, as explained in the previous paragraph. However, as of today, there are no consistent, reliable or low-cost methods of distinguishing between the pitches of instruments or voice when performing a 'global' pitch estimate on an

[7] No formal references could be found. This is a forum post. http://www.dsprelated.com/showmessage/1662/1.php

entire track. The usage of the L-R Center Channel Cancellation algorithm is our best compromise that would let us know for sure that the 'instrument' we are pitch tracking is in fact the musical instrument and not the singing vocal.

We chose to perform this component of the overall algorithm on the PC before sending the data over to the EVM because it is a relatively inexpensive operation. It does not make a difference either way.

Harmonic Product Spectrum (HPS)

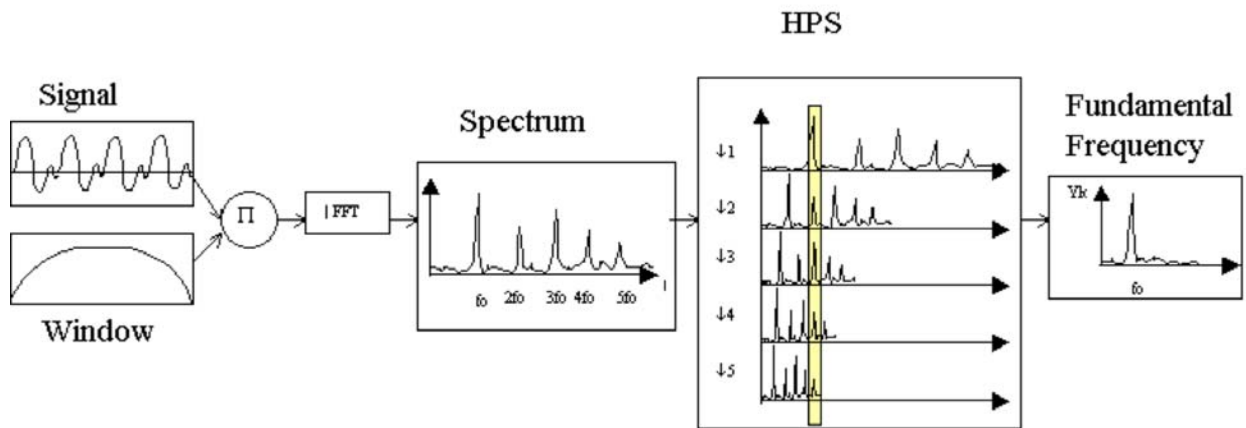

Figure 5: Diagrammatic representation of HPS algorithm

The above figure illustrates the functioning of the pitch-tracking algorithm that we selected for our project.

The pitch tracking algorithm works by dividing the input signal into a series of 4000-sample windows, with an overlap of 3000 samples. In other words, the algorithm generates a pitch estimate for every 4000-sample window and hops by 1000 samples before recomputing the pitch estimate for the subsequent frame. For a 11.025 KHz input file, 4000 samples translates into an interval of about 36 ms, which is close to the 30 ms maximum interval in which a pitch change can be picked up by the human ear.

The sampling frequency was set at 11.025 KHz for a variety of reasons. Firstly, we were only interested in tracking pitches on the musical scale, which falls between 16.75 Hz (C0) and 4978.03 Hz ($D^{\#}_8$/E^{b}_8) - our sampling frequency of 11.025 KHz would allow us to track pitches up to half the sampling frequency, or roughly 5.5 KHz. It is clear the musical scale falls comfortably within the pitch tracking range. Secondly, 11025 samples per second is a reasonable and comfortable trade-off between audio fidelity and computational complexity. The downsampling also gave us higher frequency resolution than in the case of a 4096-

STFT on 44.1 KHz input data. For these reasons, our group decided upon the 11.025 KHz sampling frequency.

During the pitch estimation process, a 4096-point STFT is performed on each window. The length of the STFT was determined through experimentation. A 4096-point STFT gave us the best results in terms of computation time versus pitch estimation accuracy. A 2048-point STFT gave us inaccurate pitch tracks, while a 8192-point STFT took longer to complete without significantly raising accuracy.

Moving on, in our instance of HPS, the FFT is then downsampled 3 more times, to give a total of 4 frequency spectrums. These spectrums are then squared, before being multiplied with one another. We square the spectrums so as to accentuate the peaks. The highest peak on the resultant spectrum gives rise to the estimated fundamental frequency of the window. The algorithm is complete once a pitch estimate is obtained for each window in the input song. A 25-second audio clip at 11.025 KHz gives rise to 273 windows, and subsequently 273 pitch estimates. However, due to the downsampling process, HPS has a tendency to overestimate pitches by an octave - this problem was taken care of when we post-process the pitch trajectory.

The algorithm works by assuming ideal harmonicity - whereby subsequent harmonics exist in integral multiples of the base fundamental frequency. By downsampling and multiplying, the subsequent harmonics of the downsampled spectrums will continue to line up with the fundamental frequency of the original spectrum, further accentuating the peak. The algorithm is also relatively inexpensive per iteration, and is immune to multiplicative noise.

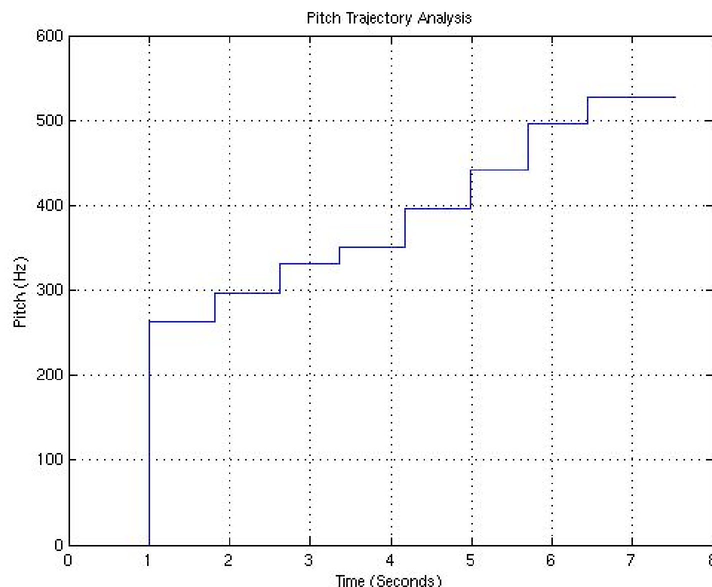

Figure 6: Pitch Trajectory of Piano playing rising C scale (4th to 5th octave)

We ran tests to verify the accuracy of our pitch tracker, by tracking the pitches of a piano recording that played a rising scale, from C (4th octave) to C (5th octave). The figure above shows how well our pitch tracker performs. We can see from the staircase-like results that the pitch tracker works perfectly. Closer examination show every step of the staircase corresponds to the pitch of the note that was played (+/- 5 Hz). Noise from the recording process also did not seem to affect the accuracy, which is an advantage imparted by the HPS algorithm.

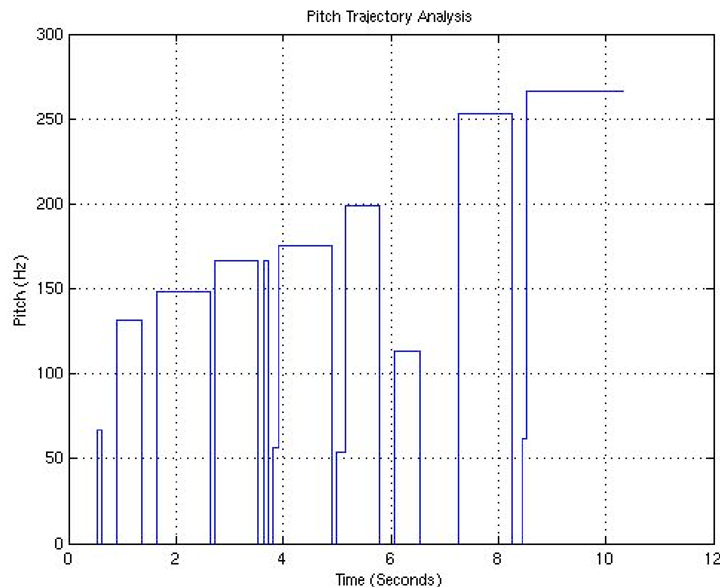

Figure 7: Pitch Trajectory of Guitar playing rising C scale (1st to 2nd octave)

We conducted a more rigorous test with stronger note attacks by testing our pitch tracker on a rising C scale played by an acoustic guitar. Once again, the general staircase trend is present, with only a few irregularities. The dips were the result of the notes dying out before the next note was done. Pitch errors due to string attacks were largely eliminated by post-processing in winToTime()[8], but these can still be seen in the form of the small steps right before t=4 and t=5 in Figure 7. On the whole, our pitch tracker continues to perform well.

To handle pitch overestimations, we reduced all pitch estimates our pitch tracker was returning by an octave. While there will be instances where HPS accurately detects the pitch, as we can see from Figure 7, frequency of such occurrences are far and few in between. Any resultant error will be insignificant on the final output to the human ear.

[8] winToTime() converts pitch estimates from window to time domain.

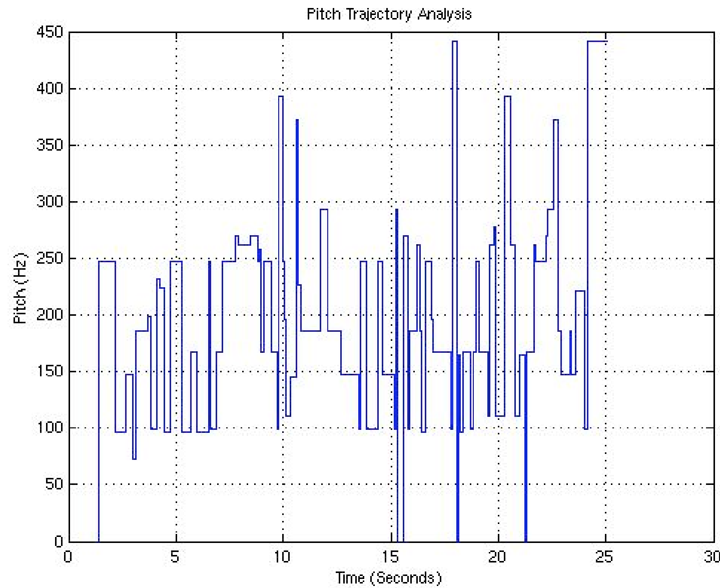

Figure 8: Pitch Trajectory of a 25-second portion of "Now and Forever" sung by James Chan

The above figure shows the pitch estimates of the guitar in a 25-second portion of one of our test tracks, "Now and Forever". Most of the pitch estimates fall between 100 Hz to 300 Hz, which ranges from A (2nd octave) to E (4th octave). While it does not appear very precise or consistent, this range in fact does include the octave range in which the song was performed in. The outliers or peaks and troughs are likely to be estimation errors induced by the attack of the guitar strings[9], or simply regions in which pitch changes occur across 2 windows. These are the edge cases in which the pitch tracker would fail.

Comb Filtering

Because of our assumption of ideal harmonicity, comb filtering became the natural filtering technique of choice. We chose the recursive 'notching' form of the comb filter, so as to reduce the harmonic components of the estimated fundamental frequency (pitch) of the background music. The filter equation is as follows:

$$y[n] = x[n] - x[n - M] + g*y[n - M].$$

where $M = \text{round}(F_S/F_O)$

$g = 0.5$

[9] Attacks of notes by instruments such as guitars or pianos often lead to pitch estimation errors regardless of algorithm. This is because some time is necessary before the note/pitch 'stabilizes'

The value M determines the location of the poles and zeroes for our filter, and it is the rounded value of sampling frequency (F_S) divided by estimated pitch (F_O).

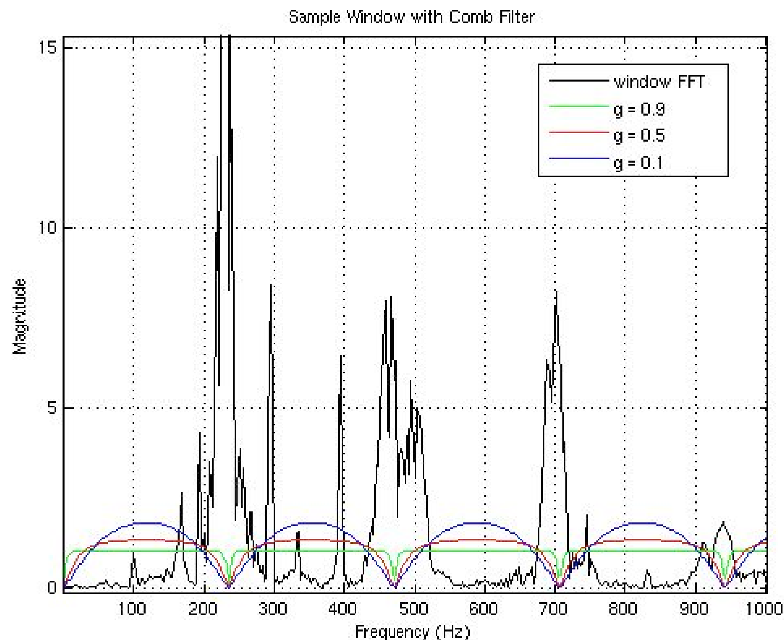

Figure 9: MATLAB plot of held spectra plots of magnitude response of comb filter and FFT of sample window

The above figure shows the FFT of a sample window, with the magnitude plots of our comb filter with a variety of different g values. A higher g value translates into a tighter notch, and vice versa. Here, we can see that a value of 0.9 would be too narrow to notch out the 4 main harmonic components, and a value of 0.1 would be indiscriminately wide. Through a series of trial-and-error, our group decided upon the value of g=0.5 for our comb filter that seemed to give us the best fidelity.

Chen-Muttukumaru (C-M) Algorithm

To further improve the quality of our output track, it was clear some form of post-processing was necessary. Despite our best efforts, comb filtering was simply unable to remove all music from portions of the input song where there are no vocals. However, it gave us enough information to identify portions of the track without vocals via some nifty post-processing work.

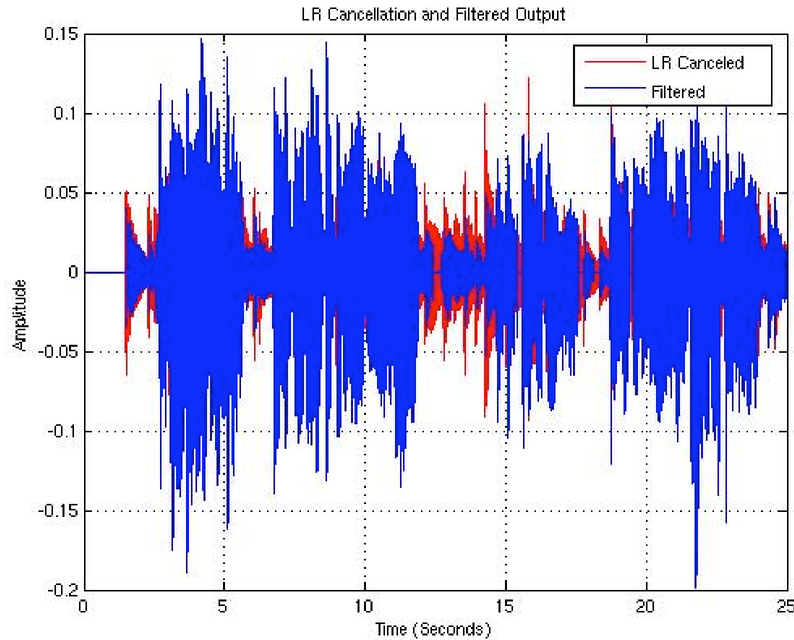

Figure 10: Pitch Trajectory of Piano playing rising C scale (4th to 5th octave)

The above figure shows 2 signals - the L-R cancelled signal of a 25-second clip (red waveform), and its comb-filtered output (blue waveform) after the series of pitch tracking as explained two sub-sections ago. Through second-by-second song analysis, we noticed a relationship between the two signals. The L-R cancelled waveform should be purely instruments, while the comb-filtered waveform should be purely vocals. It also appeared that portions of the L-R cancelled waveform that were higher in amplitude than the filtered waveform also corresponded with portions of the song where there were no vocals. From these two findings, we felt it was possible to perform a time-domain comparison between the two tracks and employ some form of tolerance checks to determine segments of the clip devoid of vocals.

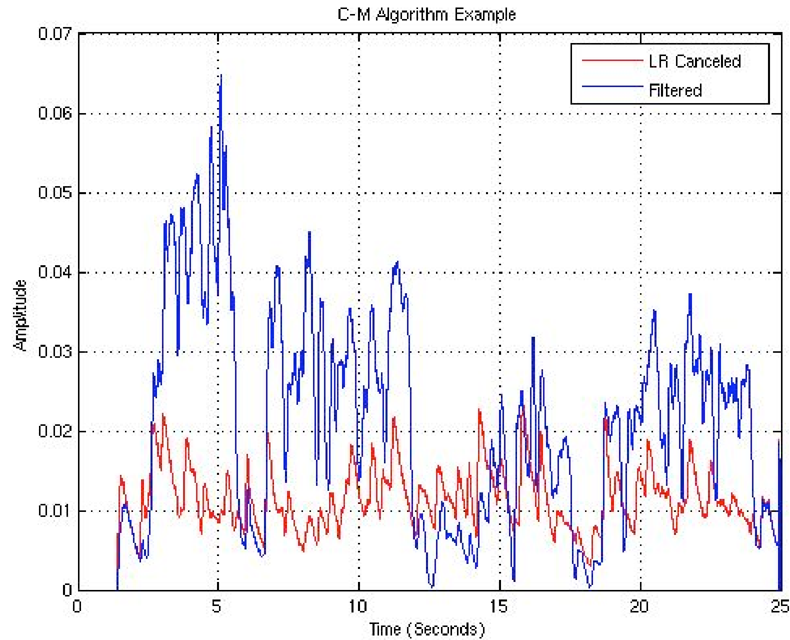

Figure 11: Overlaid time-domain plots of L-R cancelled signal and final filtered output.

To prepare the waveforms for post-processing, we applied a 1000-sample moving average smoothening process to both waveforms to smoothen out spikes and excessive troughs. We then compared waveform X (L-R cancelled) and Y (filtered). If the amplitude of X was greater than Y, we would gain that portion of Y by 0.05, so as to clip those musical portions of the filtered track without any vocals.

Because of the computational complexity of the moving-average smoothening process, we chose to perform this computation on the PC, after the filtered data is returned from the EVM.

RESULTS AND ANALYSES

TEST SIGNALS USED

A series of audio tracks were recorded via Apple's Garageband for experimental and demo purposes. Due to equipment limitations, our group had to record the instrument pieces via a MIDI keyboard and sample off Garageband's instrument library. Although the recordings are not from the actual instruments, we felt it was a reasonable compromise as test signals in our initial MATLAB experimental phase. File type was 16-bit WAVE at 44.1 KHz, and downsampled to 11.025 KHz before being fed into our system.

As mentioned under the Algorithms Used section, we used the following tracks to test our pitch tracking algorithm:

1. Instruments - Piano
2. Instruments - Acoustic Guitar
3. Instruments - Alto Sax

We then proceeded to record a series of songs for use on the eventual finished product.

1. Now and Forever, by Richard Marx
2. 爱相随 (Chinese pop song), by Emil Chau
3. 唯一 (Chinese pop song), by David Tao

In 'Now and Forever', James' vocal was mixed centrally, with bass and piano slightly to the left or right as it would be recorded if it was done at a proper studio. The bass and piano were recorded using a MIDI keyboard and real instrument audio samples from Garageband's instrument library.

In both 爱相随 and 唯一, a real acoustic guitar was used as the accompanying instrument to James' vocals.

The following commercial songs were also selected as additional test tracks, to evaluate the performance of our system on commercial songs.

4. You're My Home - Billy Joel

All test signals were downsampled from 44.1 KHz to 11.025 KHz before being passed through our system.

Problems encountered, Solutions Adopted

Our initial attempts to compile our code in Visual C with the Microsoft C compiler led to output results that were different from what we obtained on the GCC compiler. Instead of wasting time troubleshooting, we switched to an alternative open source IDE - Bloodshed Dev C, which employs a X86 port of the GCC compiler.

We were also initially unable to allocate larger variables onto the heap or stack - increasing the heap/stack limit solved that problem that.

Profiling via profiling a range or function did not always yield consistent results. When optimization was turned on, profiling the function hps() yielded abysmal cycle counts. We figured this could be because of the call to the fft() function from within the hps() function. To obtain a consistent performance indicator upon which to base our comparisons, we resorted to physically timing each function call by break-pointing and stepping over. Without optimization and paging, the fft() and combFilter() cycle counts of 35,274,127 and 91,835,560 were close to the physical timings we obtained based upon the 133 MHz speed of the DSP chip.

EVM Memory Issues/Analyses

Code size/Memory Allocation

Code Type	Code Size (in hex)
Without paging, without optimization	0xA1E0
Without paging, with optimization level 3	0xA020
With paging, without optimization level 3	0xA120
With paging, with optimization level 3	0xA5870

In all instances, the program data fit into ONCHIP_DATA without any issues.

A 25-second input song data requires 2*11025*25*4 = 2 MB (11025 samples per second, at 4 bytes per sample, one array to contain the input song and a second array to hold the final filtered output) in off-chip memory.

Optimizations Performed

The hps() function was easily the most memory intensive function in our program. The following are some characteristics of this function:

1. A 25-second clip contains roughly 273 windows, each 4000 samples in length.
2. Since hps() works only on a single window at a time, window data alone occupies 4,000*4 bytes = **16 KB** (variable name: **windx**).
3. The fft() function is called 273 within the hps() function. The 4096-point STFT requires 4096*2*4 bytes = **32 KB** (multiplied by 2 because of real and imaginary output of the STFT, variable names as **fxr** and **fxi** respectively).
4. The downsampling process then takes place after all the fft() calls.
5. Each downsampling process takes roughly 2048*4 bytes = 8KB. We require two 8 KB memory block (variable names: **page**, **fx3**) for a total of 16 KB to store each subsequent downsampled spectrum and perform the multiplication.
6. Total memory requirement for hps() is as follows:
   16 KB (**windx**) + 16 KB (**page** + **fx3**) + 32 KB (**fxr** + **fxi**) = 64 KB, which is exactly the ONCHIP_DATA capacity of 64 KB.

Because global variables are also placed in ONCHIP_DATA, we are unable to keep everything in hps() on-chip. We chose to sacrifice fx3 since it had the lowest array size and fewest accesses, hence the lowest penalty incurred for going off-chip. All other variables in hps() were either paged using dma_copy_block() or were already in ONCHIP_DATA.

We chose not to pre-fetch the subsequent 1000 samples from the next window to speed up computation because of obvious memory constraints. Also, the hps() runtime was much higher than a single copy of 1000 samples. No significant time can be saved by parallelizing this process.

We chose not to implement loop unrolling in the bottleneck function, hps(). This is because of the high number of loops we have in that function, which eliminates any speed gains by unrolling loops four- or even eight-at once. This assertion was in fact verified, where we saw no speed improvements in terms of cycles or physical runtimes after unrolling loops in hps by 4.

We chose not to optimize combFilter(), since its runtime per iteration was not the bottleneck in our algorithm. It would also be difficult to page the filtering process, due to the ini-

tial conditions in the recursive filtering process. As a result, the entire filtering process was carried out in off-chip memory.

Speed Analysis

CODE TYPE	HPS()	FFT()	COMBFILTER()
Without paging, without optimization	2 min 9.86 sec	0.74 sec	1.44 sec
With paging, without optimization	1 min 49.26 sec	0.09 sec	0.63 sec
Without paging, with optimization level 3	1 min 2.79 sec	0.58 sec	0.69 sec
With paging, with optimization level 3	53.62 sec	0.48 sec	0.79 sec

Table shows runtimes for single iteration of function. Times for fft() and combFilter() are not accurate because they were too fast to be timed accurately by hand

The times in the above table represent total time taken to complete a single call of each function. We can see from the above table that hps() is the bottleneck in our algorithm. A total of **4 calls to hps()**, **273*4=1092 calls to fft()** (one fft per window, 273 calls to fft() for each hps function) from within hps(), and **4 calls to combFilter()** are made in the entire process of our algorithm. Prior to any paging or optimization, processing a 25-second input file took our system almost 9 minutes! Paging and turning level 3 optimization gave us a final runtime of about 4 minutes.

To give further insight into the runtimes, the following table gives an estimate of the amount of computation within these three functions with paging and level 3 optimization:

FUNCTION	COMPUTATIONS
combFilter()	8.2 million cycles = 0.06 seconds (based on 133 MHz clock) 11025 samples per second, 15 cycle penalty for accessing off-chip memory twice for each array, to perform filtering from original into filtered array
hps() - fft()	9,093,142 cycles (via profile range) = 0.0683 sec
fft()	9,374,816 cycles (via profile function) = 0.0705 sec

FUNCTION	COMPUTATIONS
hps()	273*(0.0683+0.0705) = 37.90 sec < 53.62 sec We measured these times by splitting the entire hps() call into the above two components and removing the rest of the system. This was prompted by the timing difficulties we mentioned in *Problems encountered, Solutions adopted* on page 20. We postulate that by stripping down the rest of the system, the compiler has more memory to make the function faster by 15.72 sec.

GUI Design and Implementation

We designed a graphical user interface in VB 6.0. The design is as shown in the following figure.

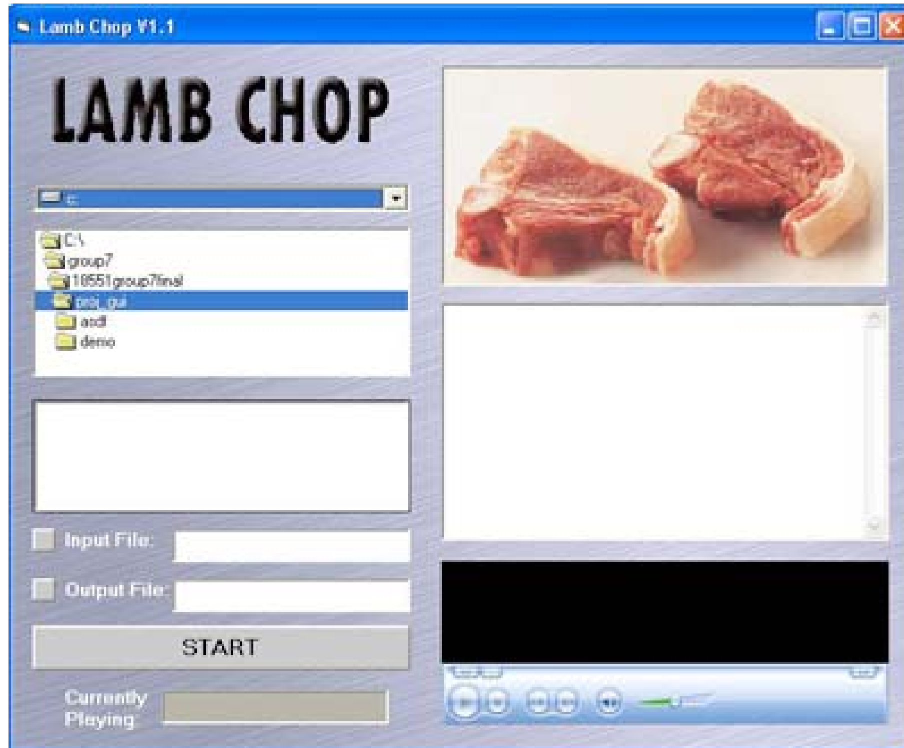

Figure 12: Screenshot of 'Lamb Chop' Graphical User Interface

The GUI allows the user to navigate between folders and select wave files as the input file into our system. It is to be noted that no downsampling is performed by our program - a downsampled clip is assumed to be fed into the system. As explained before, we have limited wave files to be a maximum of 25 seconds in duration - longer music tracks will not fit onto the EVM without significant paging. Furthermore, our algorithm already takes 3+ min-

utes to process a 25-second track - allowing longer clips would result in undesirably long computation times.

Once the algorithm completes its work, the text field on the right will show details of the process. A separate window will also pop up to indicate completion, with an 'OK' button to dismiss the separate window. The user will then be able to click on the radio button to the left of the output file text field, which loads the output file into the Windows Media Player plugin on the lower right of the GUI.

The following Figure 12, shows the organizational structure of our GUI, and how it relates to the rest of the system. Note that our C program on the PC-side performs a board reset and loads the EVM program onto the board before initiating the system.

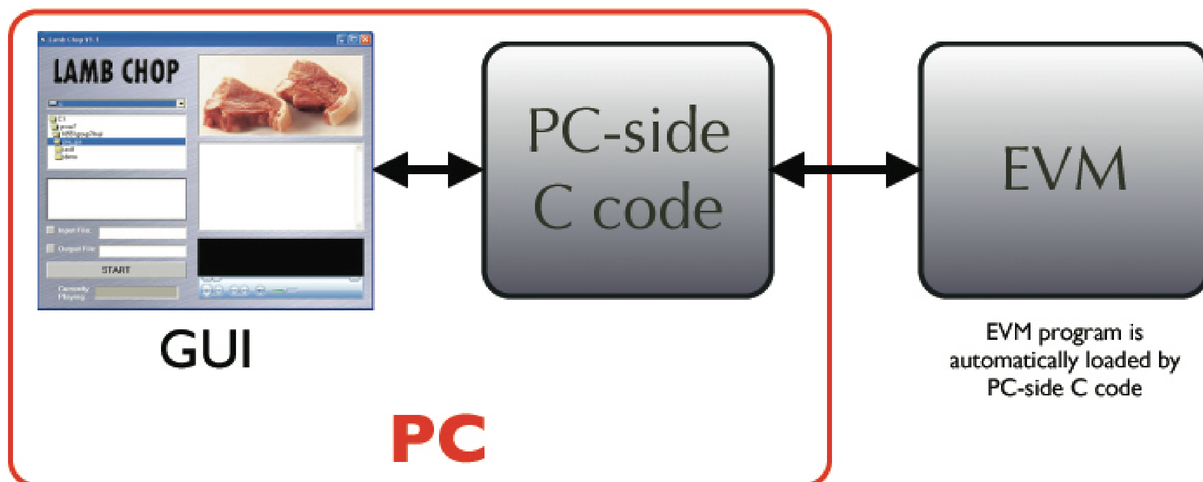

Figure 12: Block diagram depicting interfacing between GUI and rest of system.

Discussion of Results of Our System

The following tracks were used to test our system. "Now and Forever", "唯一" and "爱相随" were recorded by James Chan in Garageband, while 25-second clips of Billy Joel's "You're My Home" was extracted from one of his audio CDs.

Generally, our system worked very well. It was able to eliminate portions of the clips without vocals perfectly in "Now and Forever" and "唯一", and reduced a significant portion of the background instrument. Only the attacks of notes could be heard during portions of the clips with vocals - few notes were sustained. These two tracks embody the success that we set out to achieve in the beginning.

However, putting "爱相随" through our system gave us abysmal results. In-depth analysis showed that the L-R cancelled signal did not give a clear separation between vocals and in-

struments. In addition, the guitar strings were constantly being struck, leading to a significantly higher number of attacks than in the previous 2 tracks. These combined to cause our system to fail miserably.

We then went a step further and tested our system on commercial tracks. Most failed miserably as well, because the L-R cancellation gave our system nothing to work with. This is an important step to get our algorithm underway without oracle knowledge, because we rely on the L-R cancellation to give us relatively-isolated instruments from which to pitch track. Billy Joel's "You're My Home" worked surprisingly well, and further analysis showed a high degree of voice removal in the L-R cancelled track, which explains the success of our system.

SUMMARY

From *Discussion of Results of our System*, it is apparent that our system is far from perfect. However, it achieves what we have set out to do very well. Nevertheless, it is important to acknowledge the limitations of our system - we assume perfect harmonicity, yet sound is not always perfectly harmonic, i.e. with harmonics in perfect integral multiples. Our pitch tracking algorithm assumes perfect periodicity and as a result fails to take into account inharmonic signals or non-harmonic sound such as those produced by the saxophone.

We came across several ideas in our work that we were unable to apply or try due to hardware, time or knowledge limitations. An excellent example would be the multiple harmonic trackers that Avery Wang uses in his paper titled *Instantaneous and Frequency-Warped Signal Processing Techniques for Auditory Source Separation*. We regret that we were unable to pursue that path and implement it this time round, and hope that a future group will take up the challenge on an updated, more powerful version of the Texas Instruments EVM and duplicate Avery Wang's success. For an example of how successful his algorithm is, listen to his audio samples from
ftp://ccrma-ftp.stanford.edu/pub/Publications/Theses/AveryWangThesis/sound_demos.

Easy and reliable auditory source separation remains an open problem even today - yet if progress in the realm of DSP is anything to go by, we can be assured that an elegant solution will be found sometime in the near future. Until then, our ears shall remain king.

REFERENCES

Content References

1. *Instantaneous and Frequency-Warped Signal Processing Techniques for Auditory Source Separation*, Avery Wang, CCMRA Stanford, 1994

Comprehensive background information on history and various approaches to auditory source separation. Paper had very successful results employing multiple harmonic locked loops to track each harmonic component of an audio source it was trying to separate, but its algorithm was too difficult to implement due to hardware/time constraints.

2. *HPS Harmonic Product Spectrum Algorithm* (presentation), CCRMA Stanford
Implementation of Analysis of Pitch Tracking Algorithms, Stefan Uppgard, Sweden
Efficient Pitch Detection Techniques for Interactive Music, Patricio de la Cuadra, Aaron Master, Craig Sapp, CCRMA Stanford
State-of-the-art in fundamental frequency tracking, Stephane Rossignol, Peter Desain and Henkjan Honing, University of Nijmegen, Netherlands

Learnt about various pitch tracking algorithms, and specifically the HPS algorithm, its strengths and weaknesses. Obtained MATLAB M-file for HPS from Connexions site.

3. *Multiple fundamental frequency estimation based on harmonicity and spectral smoothness*, A. P. Klapuri, IEEE Speech and Audio Processing

Obtained our inspiration for repeated pitch tracks/filtering to improve on fidelity of output track from this paper. Paper discusses how to perform auditory source separation by pitch tracking multiple instruments in a polyphonic track with the aid of oracle knowledge (number of instruments in track, music score).

4. Comb Filters, http://ccrma.stanford.edu/~jos/pasp/Comb_Filters.html
Adaptive Comb Filtering for Harmonic Signal Enhancement, Arye Nehorai and Boaz Porat

Learnt about details of comb filters, worked with Rajeev Ghandi to perfect our filter design. Also modified comb filter code from Professor Richard Stern to tailor for our use. Second paper was not specific to our application, but gave us useful insights into how comb filters work.

Code References

1. Bloodshed Dev C IDE (x86 GCC port), http://www.bloodshed.net
2. C Wave Library to access wave files,
   http://www.mcternan.me.uk/MCS/Downloads/WavLib.zip
3. MATLAB M-file for HPS,
   Pitch Detection Algorithms, http://cnx.rice.edu/content/m11714/latest, Connexions
4. Radix-2 (decimation-in-time) STFT C code, http://cnx.rice.edu/content/m12016/latest

It's the end...

No more cries for help from the poor lambs.

The lambs have been silenced!