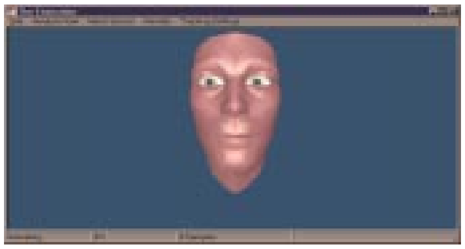
The Enunciator



Harmonic Reinforced Time Domain Frequency Tracking Technique to Perform Auditory Source Separation and Phoneme Detection for Frame Sequencing of a Tri-Dimensional Avatar….(whew)….

18-551 Final Report
Spring 2000
Group 4!

Khalid Goudeaux (ksg), Debbie Dressler (dd)
& Kevin Davies (kld)

INTRODUCTION:

The Problem:

Auditory scene analysis is a topic that has garnered a lot of attention because humans can do it so effortlessly. We can sit in a room and "tune" ourselves to listen to music, a person speaking, or a chirping bird outside. When listening to music, we can easily discern the tempo, pitch, and beat while separating and comprehending the foreground and background vocals.

Auditory source separation has been a challenging problem in DSP. It is difficult to develop an algorithm for source separation due to the significant frequency overlaps in the audio spectrum. Separation by means of an FFT analysis can be inconclusive. It is impossible to tell how much one source is contributing to a given frequency in a frequency spectrum if two different sources are both present at that frequency. The problem becomes even more difficult if a single channel audio signal is to be separated. This is commonly known as the 'cocktail effect' and there are many ways to go about solving this problem.

A useful source separation algorithm is very attractive because of its many applications. It can be used as a preprocessing step before performing speech/voice recognition to allow for more robust operation. It could be used to automatically transcribe sheet music for a live performance. The problem we seek to address is a simplified version of the Cocktail Party Effect in that we only wish to recover the lead vocal from a musical recording. This speech information will then be used to drive the lips of a 3D avatar and, in turn, make it lip sync with the music. This phoneme detection software already exists.

The Solution:

There were several possible approaches that we considered before tackling this problem.

Spatial Methods: This technique assumed a priori knowledge of the different spatial location of N different microphones in order to separate N different sound sources. This approach may assume that the voice is only present in one channel of a stereo audio signal in order to suppress the voice which is what last year's 551 group assumed for the karaoke application.

FFT based: Using methods such as the Fast Fourier Transform (FFT), Multiresolution Fourier Transform (MFT), Harmonic-Based Stream Segregation (HBSS), and correlation, these approaches attempt to decompose the signal. Components are grouped based on single frequencies or harmonics over time.

Marrian: These approaches use models of perceptual information processing. They utilize multiple techniques such as FM, onsets, AM, proximity, and harmonicity. The information is generally gathered in an AI-based algorithm and requires a learning stage [Wang,3 – 5].

We rejected these because of the unreliable results the Spatial and FFT produce. Spatial methods use too many assumptions about a signal and restrict analysis only to a small percent of the stored audio signals that actually exist. In order to have the frequency resolution needed to accurately determine frequencies present at a given time for the FFT method we would have to compute many FFT's over small numbers of samples and analyze the changes of the spectrums to change the filters. There is an extremely high number of computations involved in this and would not be practical on sampled data of 44100 samples per second to ever aim for a real time implementation. The Marrian neural network approach has the possibility of producing good results but would have been too complex to complete in a semester.

We decided on using a technique called *'Frequency Warped Signal Processing'* which was presented in Avery Wang's PhD thesis for Stanford. The technique uses instantaneous frequency and phase calculations on samples to calculate the changes in phase and frequency to help 'guess' the frequency of the next sample. The core piece of his algorithm is a frequency locked loop (FLL) that calculates instantaneous phase and frequency of the signal. This information is used to make an educated guess of the sample's frequency using current and previous samples' frequency information. The harmonic locked loop (HLL) is an array of frequency locked loops working together and comparing information to help make the most accurate guess of the frequency. This takes advantage of the harmonics that are present in a voice or instrument signal to produce better results in tracking with the noise of other sound signals present.

METHODOLOGY

Frequency Locked Loop (FLL):

Signal Representation:
The frequency locked loop algorithm looks at an audio signal at each instant in time as a bunch of amplitude modulated waveforms added together. This is similar to AM radio signals that exist in space, each occupying their own carrier frequency. The carrier frequency of an AM radio station is analogous to the fundamental frequency in a sound signal. The difference between AM signals and audio signals is that the carrier/fundamental frequency of an audio signal is not stationary in time. Below is an extremely simplified spectrum of 4 different voices with 4 different fundamental frequencies.

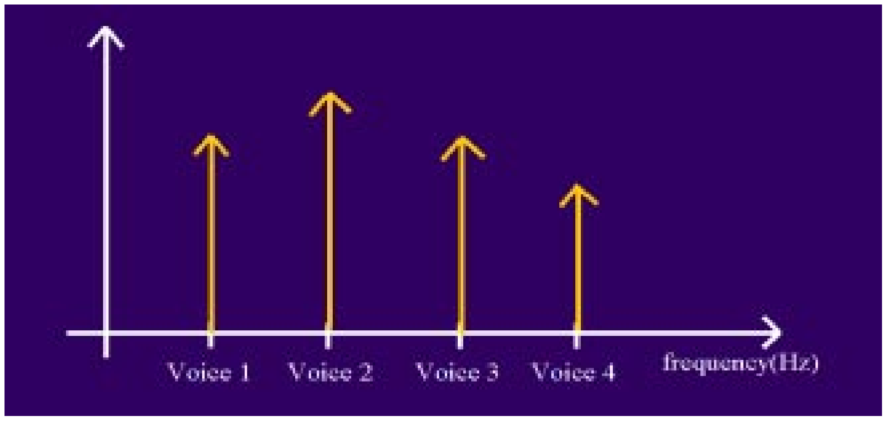


Fig 1: Simplified voice spectrum

In order to isolate ‘Voice 2’ we multiply the signal by a sinusoid with the frequency of ‘Voice 2’ to center it about the origin. After lowpass filtering, the resulting signal will only contain ‘Voice 2’ centered about the origin.

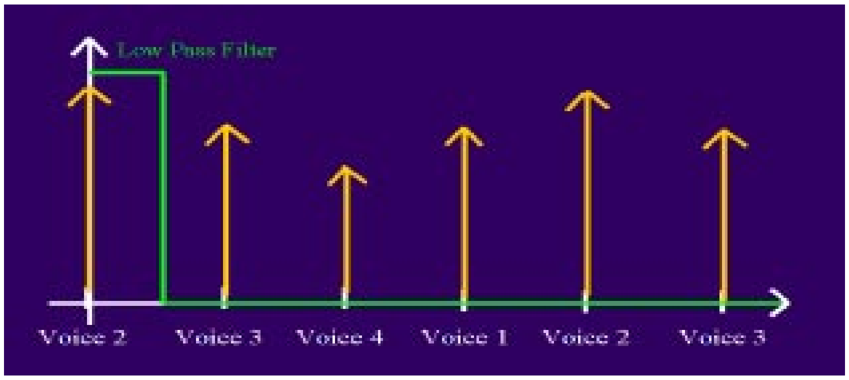


Fig 2: Isolated signal after FFT and LPF

Since the oscillator used to demodulate is controlled by the FLL’s guess of the frequency, the signal we want to track will generally not end up right at the origin.

Frequency Calculation:

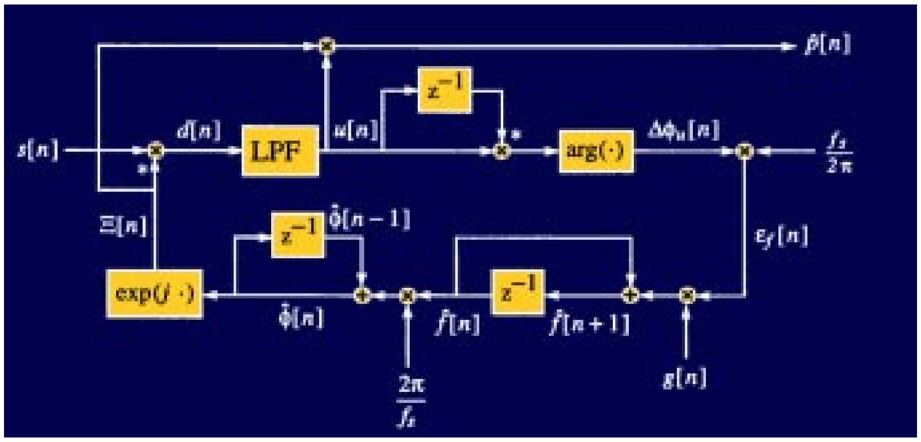


Fig. 3: Block diagram of FLL [Wang, 86]

Definition of Terms:

Inputs:

s[n]	Incoming audio signal
f[1]	Initial guess of frequency
g[n]	Feedback gain

Outputs:

p[n]	Reconstructed output
f[n]	Guessed instantaneous frequency

Other:

_[n]	Oscillator output
d[n]	Demodulated signal

The fact that there is very little time between samples allows us to use the approximation that the change in frequency is proportional to the change in phase. To calculate the change in phase we multiply the output of the lowpass filter (LPF) with the complex conjugate of the previous output of the LPF and calculate the phase of the complex result [Wang, Chp. 4].

$$u[n] * u[n-1]^{*} = \text{real} + j * \text{imag}$$
$$\Delta\phi_u[n] = \arctan(\frac{\text{imag}}{\text{real}})$$
$$\varepsilon_f[n] = \Delta\phi_u[n] * (\frac{f_s}{2\pi})$$

Where u[n] is the output of the LPF, $\Delta\phi_u[n]$ is the change in phase, and $\varepsilon_f[n]$ is the change in frequency of the input frequency you're tracking. To update the frequency of the oscillator, the change in frequency information multiplied by a gain term (g[n]) is added to the current estimated frequency and then converted back to a phase argument to be fed to the oscillator. The oscillator then makes new calculations for the demodulation of the next sample.

Reconstruction:

If the FLL is tracking correctly, to get our desired signal, we only need to remodulate the output of the lowpass filter to get the amplitude information and recenter the signal at its fundamental frequency.

Considerations:

* This implementation does require some a priori knowledge about the signal and needs a fairly accurate guess of the frequency to start off.
* The FLL does well at tracking a signal as long as another audio source doesn't find its way into the passband of the lowpass filter. Once another frequency comes within the bandwidth of the filter away from the signal being tracked, the FLL will track the one with the highest amplitude, which is not necessarily the desired signal.
* The gain term g[n] basically controls how fast the frequency is able to change in our oscillator. If the gain term is set too high, the frequency estimations jump around too much, and the signal is lost. If the gain term is set too low, the FLL will not be able to keep up with the changes in frequency and will lose the signal as well. We found that for most audio signals, setting g[n] between .01 and .02 worked well.

Harmonic Locked Loop (HLL)

Signal Representation:

The FLL is good for tracking single frequencies, however voice is a harmonic signal with multiples of the fundamental frequency present. Since the FLL becomes confused when frequency crossing occurs and in some cases starts tracking the wrong signal, the redundancy of the harmonics helps to predict the correct frequency in these cases.

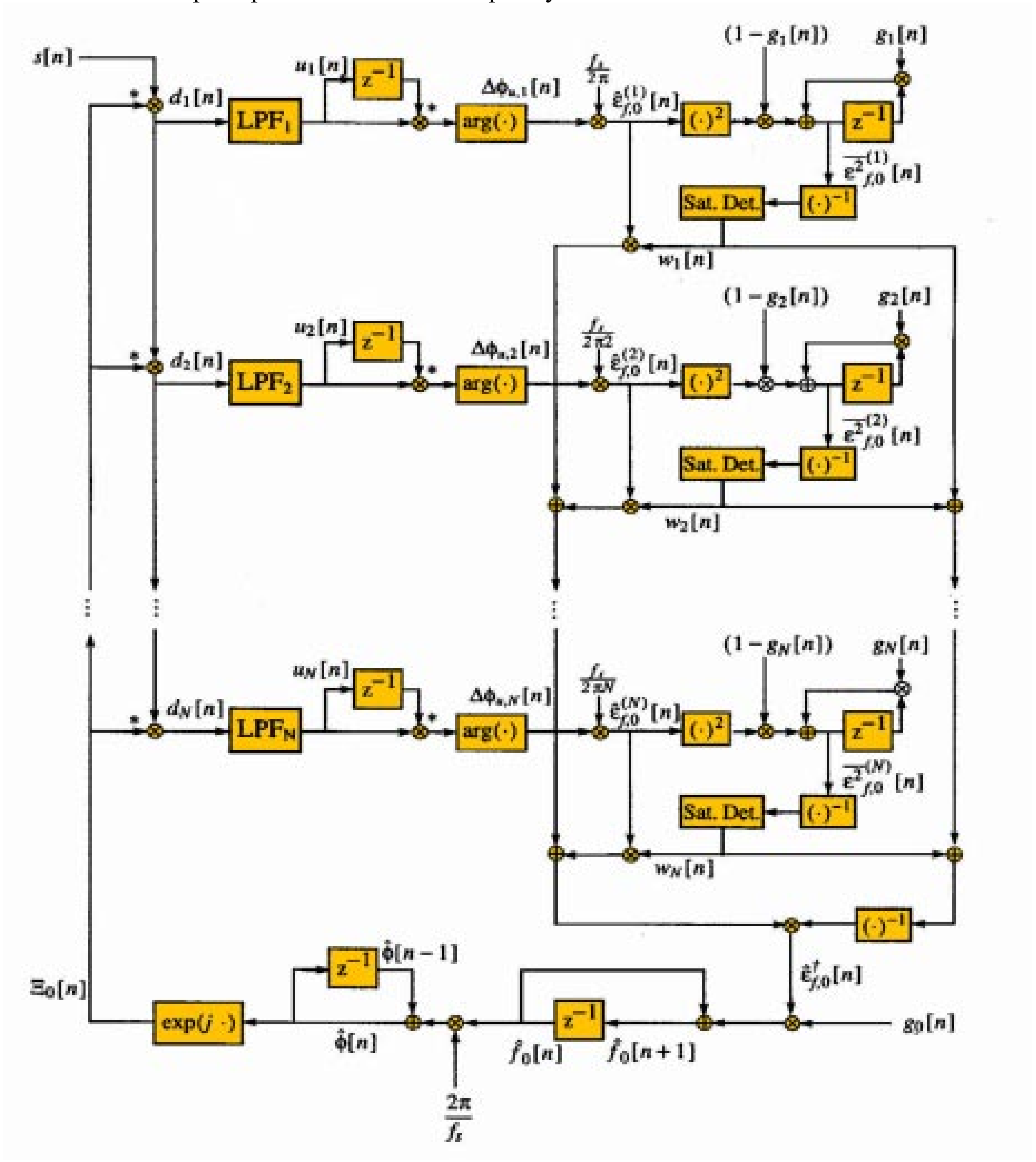


Fig. 4: Block diagram of HLL [Wang, 98]

Frequency Calculation:

The basic structure of the HLL is simply an array of FLL's with some modifications to calculate the next fundamental frequency. The redundancy of the demodulation helps condense the structure since only one oscillator is needed to repeatedly demodulate the incoming signal and shifting the harmonics down to the origin for lowpass filtering. The change in frequency of the fundamental is calculated repeatedly in each FLL for each harmonic. The only difference with the FLL in this step is that each harmonic is not calculating its own change in frequency but the fundamental's change in frequency so the change in phase when converted to change in frequency is divided by the number of the harmonic.

$$\varepsilon_{\mathrm{fi}}[\mathrm{n}] = \Delta\phi_{\mathrm{u}}[\mathrm{n}] * (\frac{\mathrm{f_s}}{2\pi * \mathrm{i}})$$

Where _$_{\mathrm{fi}}$[n] is the change in fundamental frequency calculated using the i$^{\mathrm{th}}$ harmonic. The variance of each change in frequency calculated in order to produce a weighting term for a weighted average.

$$\mathrm{var_i}[\mathrm{n}] = (\text{å}_{\mathrm{fi}}[\mathrm{n}]^2 * (1 - \mathrm{g}[\mathrm{n}])) + (\mathrm{var_i}[\mathrm{prev}] * \mathrm{g}[\mathrm{n}])$$

var$_{\mathrm{i}}$[n] is the variance of the n$^{\mathrm{th}}$ sample of the i$^{\mathrm{th}}$ harmonic. This is done to take into account frequency crossing that may occur at some of the harmonics so if a harmonic has a high variance it correspondingly gets weighted less in the weighted sum.

$$\mathrm{weight_i} = \frac{1}{\mathrm{var_i}[\mathrm{n}]}$$

$$\mathrm{f_{f0}}[\mathrm{next}] = \mathrm{f_{f0}}[\mathrm{n}] + \frac{\overset{\mathrm{numharm}-1}{\underset{\mathrm{i=0}}{}} \mathrm{weight_i} * \text{å}_{\mathrm{fi}}[\mathrm{n}]}{\overset{\mathrm{numharm}-1}{\underset{\mathrm{i=0}}{}} \mathrm{weight_i}} * \mathrm{g}[\mathrm{n}]$$

Where $f_{f0}[next]$ is the guess of the next fundamental frequency to be fed into the oscillator [Wang, Chp. 5].

Reconstruction:

To reconstruct the signal, the output of each LPF for each harmonic is re-modulated up to its correct frequency by an oscillator running at (i+1)* f$_{\mathrm{f0}}$ Hz. The output of the LPF contains its amplitude information and once that has been re-centered to the correct harmonic, the reconstructed signal is just the sum of all these re-modulated harmonics.

Considerations:

The harmonic locked loop provides a much more robust method of tracking a voice or instrumental signal however, the nature of this method poses some potential problems.

* Since the HLL is looking for a frequency to track at every sample, if the singer takes a breath or the instrument stops playing, the trackers will just wander and the output will not be predictable. When the voice or instrument comes back in the trackers may have wandered too far away in frequency and started tracking something else, so the desired signal will be lost.
* The LPF becomes more important to the accuracy of the HLL when more sources are in an audio file. Decreasing the bandwidth and increasing the order helps to isolate the tracked signal and prevent the HLL from getting confused by surrounding frequency signals.
* The implementation shown above works well for instruments that have all harmonics present in its spectrum. Voice also has all harmonics present in its spectrum so the HLL provides a good solution to our problem at hand. However, if this were to be applied to instruments with only odd harmonics present, the trackers would perform poorly.
* Parameters:
  - Time varying filters could be used to aid the tracking. If the frequencies' trackers have a lot of disagreement on what the next frequency should be or the first guess of the fundamental frequency is off a bit, a filter with a wider bandwidth could be used. Once the trackers latch on, a LPF with a much smaller bandwidth could be used to improve the sound quality of the output and help with more accurate tracking.
  - Time varying gains (g[n]) could also be used for the same sort of purpose. The gain could be set to a high value when starting the trackers to help them make quicker adjustments to latch on to the correct frequency. This could be set to a lower value to increase stability once the signal is found.
  - The number of FLLs in the array is a parameter specified by the user depending on the type of signal being tracked.

IMPLEMENTATION:

The HLL tracker was implemented on a TMS320C6701 Evaluation Module (EVM) board containing a Texas Instruments C6701 DSP (C67). Music samples were transferred to the board, tracked using the HLL algorithm, and the reconstructed signal was transferred back. The HPI direct memory access method was used to transfer the audio samples to and from the EVM board. This placed the bulk of the data transfer overhead on the host PC rather than the EVM. In addition, the EVM sends the current frequency at the end of every block.

The EVM code was set up to accept either 11025 or 44100 Hz sampled data, represented as single precision floating point numbers. We added this versatility in so that the EVM could process CD quality audio or the 11025 Hz sampled audio that the phoneme tracking software uses. The LPF size and tap values needed to be chosen based on the sampling rate. Otherwise, the cutoff points of the filter would be inappropriately scaled. Two sets of LPF coefficients were stored on the EVM, and the code choose the proper filter based on the sampling frequency sent from the PC (described below). The LPF size may be variable since a filter with the same cutoff points and quality will need more coefficients at 44100 Hz sampling than at 1025 Hz.

Three buffers were created on the EVM to minimize the idle time of the C67. At any given time after the initial setup, original audio was being transferred into one buffer, the EVM was processing the data in one buffer, and the processed data was being transferred back to the PC. In order to allow the LPF of the HLL tracker to occur easily in another separate circular buffer, the size of each of the three main data buffers needed to be a multiple of the LPF size. In the final implementation, the factor was 12. Due to the variable LPF size, the buffer size was also variable. The PC and the EVM needed to know both the buffer size, and it could be determined from the sampling frequency.

After the EVM sends the PC an initialization message, the PC sends a set of parameters as a group in FIFO fashion. The parameters include the starting fundamental frequency guess, the sampling frequency of the audio, the number of harmonics to be tracked within the HLL, and the gain of the frequency tracking ($g_n[n]$, assumed to be equal and constant throughout the musical piece). After these parameters are sent, the EVM gives the PC the address of the buffer in which to write the block of audio. The EVM gives the PC the address of the second buffer and immediately runs the HLL on the first block of samples. After this, the buffers are set up to begin a loop in which PC writes a block and receives a block while the EVM processes one. The transmission from the EVM of a memory address for new data is used as a message that the EVM has complemented a block. The PC sends an acknowledgement message after transferring the original and processed audio. The value of this acknowledgement changes after all data has been processed, and the EVM closes down. The flowchart in Figure 5 shows an overview of the communications and processing, and Table 1 details the communication and data transfer processes specific to the PC and EVM.

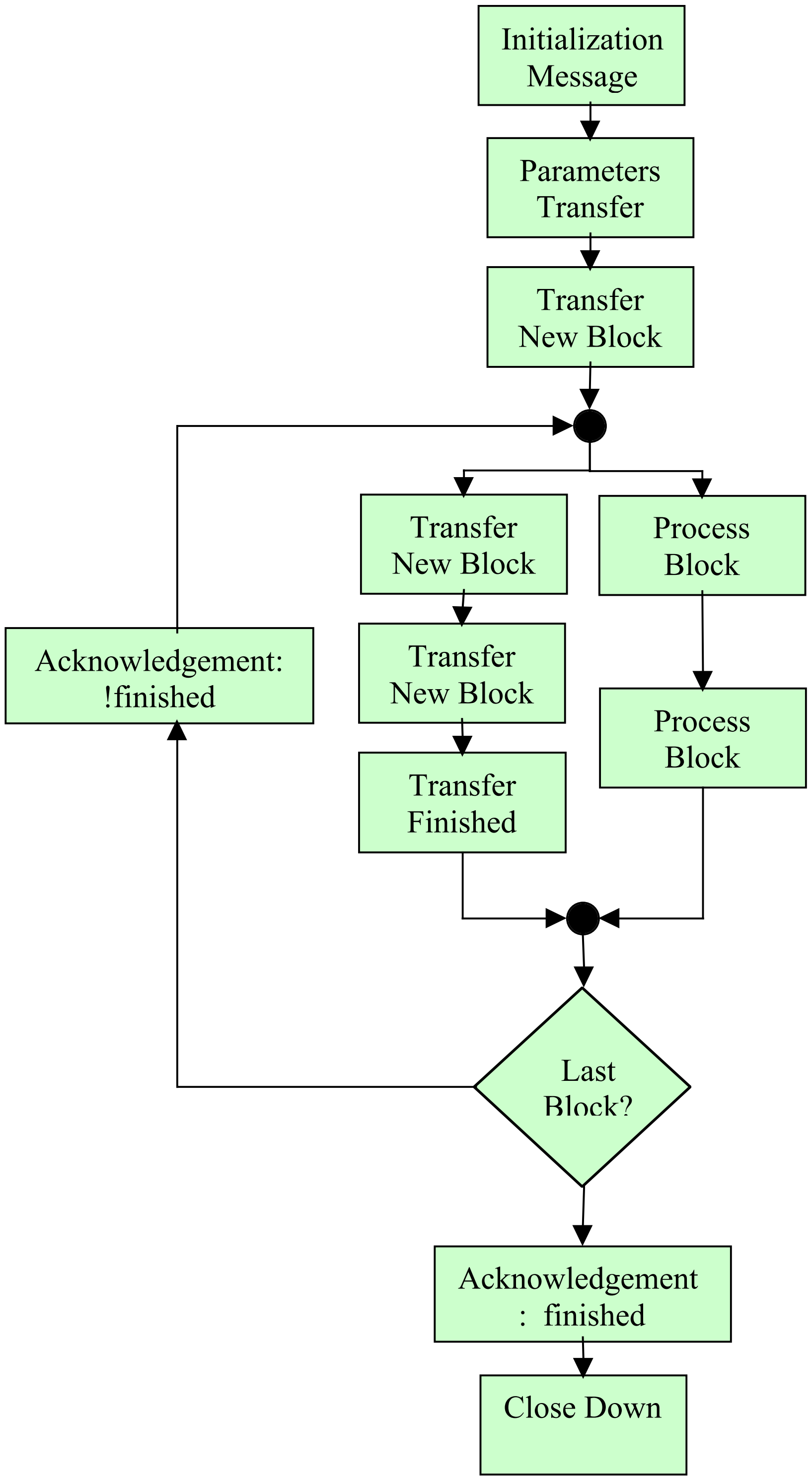


Fig. 5: Communication and tracking flowchart

	EVM	PC
	Send initialization message (sync send)	
		Receive initialization message (sync, using windows event to signal message)
		Send parameters (FIFO)
	Receive parameters (sync FIFO)	
	Send address of block for new data (sync send)	
epeat vice		Receive address of block for new data (sync, using windows event to signal message)
		Write block data (HPI)
		Send acknowledgement (message)
	Receive acknowledgement (sync retrieve)	
	Send address of finished block (mailbox 2)	
	Send current frequency (mailbox 3)	
	Send address of block for new data (sync send)	
Repeat until end		Receive address of block for new data (sync, using windows event to signal message)
		Receive address of finished block (mailbox 2)
		Receive frequency (mailbox 3)
		Write block of new data (HPI)
		Receive block of finished data (HPI)
		Send acknowledgement (message)
	Receive acknowledgement (sync retrieve)	

Table 1: Detail of communication between EVM and PC

Speed

Unfortunately, the HLL algorithm limits the use of parallelization to decrease processing time. In the HLL feedback loop, most of the computations are dependent on previous computations. In the multiplications that require real and imaginary parts, it should be possible to use both A and B side floating point multipliers of the EVM. We ran into difficulties with floating point however. There seems to be a bug within the Code Composer environment in which the 'Target Version' setting resets itself to 'Default'. With this setting, the floating-point unit of the C67 is not used, and computations take much longer. Based on our profiling, it seems that a floating-point multiplication takes 12 clock cycles when the C67 floating point multiplier is not used.

We must look at the most computationally intensive operations first. The sin, cos, atan calls in the HLL algorithm slow the C67 considerably. We consider three possible implementations of these trig functions: math.h, mathf.h, and a lookup table. Table 2 shows the amount of time for each operation. Again, the floating-point unit is not being used.

	<math.h> (cycles)	<mathf.h> (cycles)	Lookup table (cycles)
cos	185	185	130
sin	398	185	185
atan	300	147	84

Table 2: Cycles for single calls of trig operations

For each sample, there are (1 + <number of trackers>) cos operations and the same number of sin operations. The number of atan operations is equal to the number of trackers. Ideally, we would like to run 10 trackers, so the time consumed by these computations is very significant. The total time is shown in Table 3.

	<math.h> (cycles)	<mathf.h> (cycles)	Lookup table (cycles)
cos	2035	2035	1430
sin	4378	2035	2035
atan	3000	1470	840
total	9413	5540	4305

Table 3: Cycles for trig operations of one sample tracking 10 harmonics

Strangely, the use of a lookup table does not present a significant enough improvement over the mathf.h library to overcome the loss of accuracy it presents. For this reason, the final EVM implementation uses the mathf.h library for trig operations.

Floating-point operations are also important. We are able to halve the number of floating point multiplications by taking advantage of the symmetry of the FIR filter. The filter must multiply real and imaginary parts, though. There is a total of (1 + 18*<number of trackers> + <LPF size>) floating point multiplications. This represents about 6972 clock cycles with 10 harmonics and 400 LPF taps, since the compiler refused to use the floating-point unit (12 cycles per multiplication). Using the mathf.h library and tracking 10 harmonics with a LPF size of 400, there are a total of 12512 cycles dedicated to trig and floating-point multiplications.

It was important to place the most frequently used code and data in C67 onchip memory for speed considerations. By placing the ldev6x and ldrv6x library RTS code in off chip SBSRAM and limiting the heap size to 0x001500 (slightly above what is needed for to allocate memory for 10 harmonic trackers), we were able to place the signal buffers and code in onchip memory. The signal buffers represented 28800 bytes of memory.

DEMO:

At our demonstration, we showed separation of a soprano opera singer from a background string instruments. We then used the separated voice to drive the lips of a 3D avatar. We built a multi-threaded Win32 application that imported VRML files and rendered the scene in OpenGL. The face model was composed of a mesh of triangles rendered with smooth shading. The application spawns a new process that swaps the audio file down onto the EVM board for processing and stores the result. Using the Windows Messaging API, it communicates with its parent application to update it on its progress. The system is stable, but in accordance with Murphy's Law, did not perform completely as planned at the demo.

Given more time, we would have completed the addition of an animated violin to the system. We planned to bring the raw frequency information back from the EVM board and use it to drive the violin. The rate at which the bow would move back and forth across the strings would be controlled by the variance of the frequency. In addition, a hand would be placed in the scene and moved it up and down strings as a direct function of the frequency. Lastly we had planned to place an animated graph in the background that would plot the frequency in real time by using simple points in OpenGL.

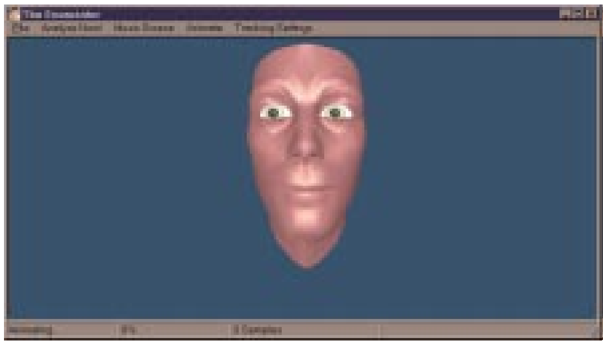


Fig. 6: Demo Application Rendering Face

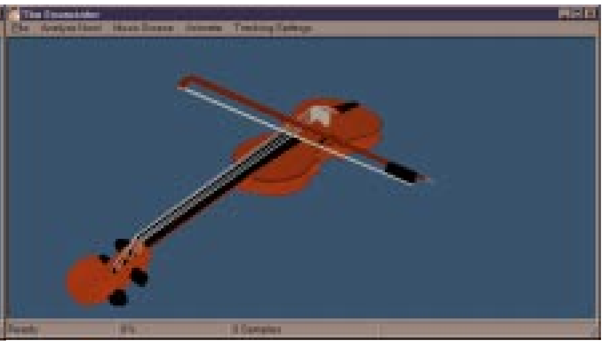


Fig. 7: Demo Application Rendering Violin

DISCUSSION:

After developing the code for the HLL and running it for several sound files we found that it performed poorly for sharp changes in frequency. It did well however, on opera type music where the singer sings a vowel sound most of the time and rarely articulates notes. The system was also very sensitive to the initial seed frequency given as input for the fundamental frequency. In general if our estimate was too low, the higher harmonic FLLs would be pulled down by the fundamental partial and the whole system would die. The fundamental would sometimes get pulled down by the higher harmonic tracking so far that it went negative. This could be avoided as in some cases as long as the initial guess was higher than the actual frequency.

The trackers falling in frequency could sometimes not be completely avoided just by changing the initial guess of the fundamental frequency. This is probably due to the fact that an FIR filter was needed to retain phase information so its size was limited because of processing speed. As we increased the order of the filter the quality of the output did increase and the trackers were less likely to go off track. We initially didn't realize exactly how important the LPF was to the system so we didn't spend as much time on it as we could have. Our filter never got good enough to completely block out surrounding signals. Its accuracy was restricted by the FIR's large size so the trackers would frequently die and fall in frequency if the bass was not eliminated well with the filter. Avery Wang had an implementation that used minimal computation to design an IIR filter with linear phase. Given more time we could have used this information to greatly reduce computation time of our filter and increase accuracy of the LPF for better tracking.

In some cases, it helped the accuracy to increase the number of harmonic trackers running on a signal but in some cases it didn't. Depending on the tone and quality of the voice being tracked, there were sometimes more significant harmonics in one signal than another. In the sound clips included we used 11 FLL trackers in our HLL.

Since we did not have an implementation for varying the gain term g[n] we just input it into the system as a constant somewhere between .01 and .02. This was varied depending on the nature of the signal we were trying to track.

Although, our method did not work on all types of music there is potential for it to be greatly improved. An advantage of this method is that after running the code you're left with a list of frequencies corresponding to the fundamental frequency at each sample which could be very useful especially since no other method can provide that information down to each sample but must group samples together over a period of time.

The output was sometimes distorted due to jumps in frequency. An addition to correct this would be a 'Kay's Smoother' [Wang, 84]. In order to smooth out the transitions in frequency from sample to sample, an exponentially weighted average of the previous samples is taken and the resulting frequency is used for reconstructing the signal.

It took about 37 seconds to process one second of 44100 Hz sampled audio using a LPF size of 200 and 10 trackers and 4.5 seconds to process one second of 11025 Hz sampled audio using a 88 tap LPF filter.

**The original and separated sound files along with the code are available at /afs/ece.cmu.edu/usr/dd/551/sounds.*

FUTURE WORK:

What we have presented has several places where tracking can be improved, great potential for future research and many applications.

One improvement is to provide different models for the locations of the harmonics. Some instruments only have odd harmonics while other instruments, like the piano, have harmonics located at greater than k times the fundamental frequency.

To enhance performance, other types of filters could be considered. It is possible to create a linear phase truncated IIR (TIIR) filter using the Additive Factorization and Magnitude-Squared Design methods outlined by Wang.

A logical addition to the system would be to automatically seed the harmonic tracker with the fundamental frequency. Locating onsets within the audio using a cepstral technique could do this. The ultimate goal of this type of work is to be able to parse an audio scene like a musical piece and separate all of its components. A map of the song could be created that contains all of the singers, percussion and instruments which could then be used for more abstract musical processing like determining tempo, crescendo and decrescendo, intro, verse, chorus and outro.

Such information would be useful to people who create visuals that are keyed with music. To be able to easily extract abstract properties of music would make timing video to the beat or feel of music much easier.

This full-featured auditory scene analyzer would also be extremely useful as a pre-analysis step for speech processing. If others are talking in the background while the target speaker is talking, the scene could be parsed and an intelligent method of identifying the target speaker could be applied to determine which separated source should be analyzed.

There is also speech research going on to build software that can understand what is being said in a meeting and make notes of meeting times, decisions and take basic minutes. The meeting must occur in a specific room with special sensors though. With full auditory scene analysis, the same could be accomplished for a meeting conducted in the park, which opens more avenues for such a speech system.

As you can see, there are many places where continued research in auditory scene analysis could lead. We have only touched the tip of the iceberg with our project in accomplishing separation of continuous audio. It’s an exciting field where there is still much to be learned.

REFERENCES:

Wang, Avery Li-Chun. "Instantaneous and Frequency-Warped Signal Processing Techniques for Auditory Source Separation". PhD Thesis, Dept. of EE, Stanford University, 1994.

RELATED INFORMATION:

Parsons, Thomas W. "Separation of Speech From Interfering Speech by Means of Harmonic Selection". Journal of Accoustical Society of America, 1976.

Seppanen, Jarno. "Finding the Metrical Grid in Musical Signals". Tampere University of Technologyl, 1998.

Wang, DeLiang. "Prinitive Auditory Segregation Based on Oscillatory Correlation". Ohio State University, 1996.

Y. Tougas and A.S. Bregman. "Auditory Streaming and the Continuity Ilussion". *Perception and Psychophysics*, vol. 47, no.2, pp.121-126, 1990.