The Sweet, Soothing Sounds of Pittsburgh

Final Report

Fall 2007, Group 7:

Umpei Kurokawa, Ryan Sherwin, Fabian Weissenberger, Tian Yang

Convolution Reverberation

Reverberation is defined as the acoustical effect of sound waves reflecting off the walls and other physical features of an environment until they fully decay. Reverberation is both linear and time invariant, thus making it an LTI system characterized by its impulse response. A reverberation effect, such as convolution reverberation, attempts to mimic this acoustical characteristic of a location. Convolution reverberation reproduces the aural qualities of an environment to produce an output that sounds as though the input signal was recorded in that particular environment. Our goal is to create a real-time convolution reverb effect on the C67 DSK.

Applications

Convolution reverberation is in high demand in both the audio and film industries. In the audio industry, convolution reverb is desired by both the musicians and recording engineers. With convolution reverb, musicians can be recorded in a studio, but sound as though they are in one of the most famous concert halls worldwide. Musicians and engineers both utilize this effect because it can fill in a sound that may be dull or empty in nature. Reverb is typically used in audio to create a full and well-rounded sound. Movie directors appreciate convolution reverb because if there is ever a location that is impossible or

near impossible to shoot on-site, all they have to do is record an impulse response and use convolution reverb to reproduce the acoustics of that location.

18-551: Past vs. Present

In past years, there has not been a single group that implemented real-time convolution reverberation. In fact, this is the first convolution reverb project in this course, with or without delay. This means that our entire project is new to 18-551. In addition to running in real-time, we also added a Pittsburgh touch by obtaining impulse responses from locations off campus. Not only is this project new to 18-551, but it is also uncommon in industry. Many software reverb plug-ins exist and are widely used today. However, convolution reverb hardware devices are not as popular. Even less popular are real-time convolution reverb devices. Our implementation would be one of a kind from this perspective. Another atypical feature is the use of Pittsburgh locations. Not many software reverb programs offer Heinz Hall or Soldiers and Sailors Memorial Hall.

Input Database

The input signal can be any form of audio to be played in through the codec. Any song, singing, or speech can be played through our system. What is nice about this is that the input can be live or recorded due to the real-time aspect of the project. This feature was displayed in the demo. We played a recorded song ("Yesterday" by The Beatles) and used the microphone provided in lab for live speech.

Impulse Response Database

First, we will address the database of impulse responses. We obtained impulse responses from Kresge Recital Hall, Alumni Concert Hall, and Chosky Theater on campus. The off campus locations consist of a standard living room, Soldiers and Sailors Memorial Hall, and Heinz Hall. With the help of Riccardo Schulz, the Associate Teaching Professor in Music Sound Recording, we were given access to these locations and the equipment necessary to record quality impulse responses. We also got random responses from the Internet, however, the quality of these was much lower than the ones that we actually recorded [5]. These were included not for the purpose of adding more to the list, but to show two alternate ideas. First, our hardware device will work for any response fed to it with audible distinctions between impulse responses. Second, after listening to the Internet responses, a new appreciation should be had for the acoustical excellence of the concert halls in Pittsburgh and the responses recorded from those locations.

Impulse Response Collection

The first aspect of the impulse response collection is the equipment. The software used in all recordings was Digidesign's ProTools. All impulse responses were exported in the correct format (.wav) using Audacity. For all locations, the loudspeakers were at center stage and the microphones were set up in the middle of the concert hall, unless otherwise noted. Alumni Concert Hall and Kresge Recital Hall used different equipment than the other recordings due to the installed equipment in these locations. These locations were recorded using Apple's Power Mac G4. Both concert halls offer the Digidesign Digi002 hardware interface for ProTools. A powered Mackie Loudspeaker and stereo pair of Neumann KM144

microphones and were used to emit and record the source, respectively. A PowerBook G4 was used to create the excitation source.

The other locations were recorded using an Apple PowerBook G4 and Digidesign's MBox2 interface. The microphones used were Schoeps' CMC 6 amplifier and MK4g capsule. An Apple iPod was used to create the excitation source since the PowerBook was being used for the recording. The iPod connects to a Mackie mixer. The mixer then sends the signal to the QSC Audio 3350 Amplifier. Finally, the signal is sent to the RAMSA loudspeakers for playback.

Excitation Source

The excitation source sent into the room was a linear sine sweep from 0 to 24,000 Hz at a rate of 12,000. This was determined to be the best method based on research and testing. John Edwards' "Acoustic Room Response Analysis" discusses the three popular methods. The criteria he sets out for deciding on the best source is that it must span 20 Hz to 20 kHz and must be of the proper length. A short pulse will give accurate information, but fade quickly, whereas a longer pulse will be less accurate, but contains more energy to excite the room characteristics. The first is the Maximum Length Sequence (white noise). This pseudo-random number generator outputs a signal that can harm a loudspeaker. He also addresses sending an actual impulse, for example, a firecracker. This method is too quick to meet energy requirements (nor would it be allowed by any reputable concert hall). The final method that he focuses on is the chirp signal. This sine sweep covers 0 to 24 kHz and is under three seconds due to the normal room response being under 3 seconds [2].

In testing, we compared the maximum length sequence, linear chirp and logarithmic chirp. Both chirps were tested at rates of 10,000, 12,000, 18,000, and 24000. From these, it was determined that the best

result came from a linear sine sweep with a rate of 12,000. The microphones were set up using the ORTF method. This method is standard in stereo recording and is taught here at Carnegie Mellon University. In this method the microphones are set up with 110 degrees between the two cardiod transducers such that the ends are 17 cm apart. This allows for the best representation of the stereo field. The microphones were placed in the middle of the room on a stand approximately 20 feet in the air.

Impulse Response Processing

The recordings are then listened to and the best impulse response recording from each location is exported from ProTools and imported into Audacity. Although ProTools can export to the .wav format, it was determined that these .wav files are not readable by MATLAB. Audacity, however, can export files into the proper .wav format for MATLAB to read in. Now we have a recording that includes both the sweep and the room response in a format that MATLAB will accept. The following text is visualized in the image below…. The next step is to read all of these .wav files into MATLAB for processing. The recordings are in stereo, however, since we are currently working in mono, only one channel of the recording was taken. Nonetheless, both channels are still available if we were to switch to stereo.

The next step is to correlate the recorded sample with the excitation source sent out. This is done to remove the sweep from the recording and leave only the room response. Now it is necessary to get the result of the correlation into a useful form. To do this, it is first scaled so that the data ranges from -1 to 1. This scaled version is then re-sampled down to 16 kHz (from 48 kHz) in order to run on the DSK. Next, we found the peak of the correlation and cut the impulse response from that point on to include 4,096 samples. For an explanation of why we chose 4,096 samples, please refer to the Block Decomposition section. The audible affect of not keeping all samples is that the reverberation is not as full. In the

demonstration we made this clear. When listening to the end of the speech, the length of the reverb tail is an obvious indication that the impulse response has been trimmed. During speech, however, using the full impulse response washed out the voice to the point of being incomprehensible. The short impulse response actually provided the reverb and maintained intelligibility of the speech. Also, trimming the impulse response did not affect the ability to distinguish between different locations. Heinz Hall, Alumni Concert Hall, and Soldiers and Sailors Memorial Hall still maintained their respective audio qualities.

When cutting the impulse response, the first and last values were not guaranteed to be zero or close to zero. This means that for using a 4,096 sample impulse response, some clicking will result at the beginning and end of the waveforms. In order to cope with this issue, we multiplied the impulse response with a triangle function to decay the shortened impulse response to zero. It ramped up quickly in the beginning and faded out slowly at to the end. Using this envelope or window is both good and bad. The negative aspect of it is that it slightly changes the impulse response. This means that we are not accurately representing the location to the precision that was first intended. The positive feature of this window is that it greatly reduced the clicking sound heard when using our system. For a full-length impulse response, this window would not be necessary.

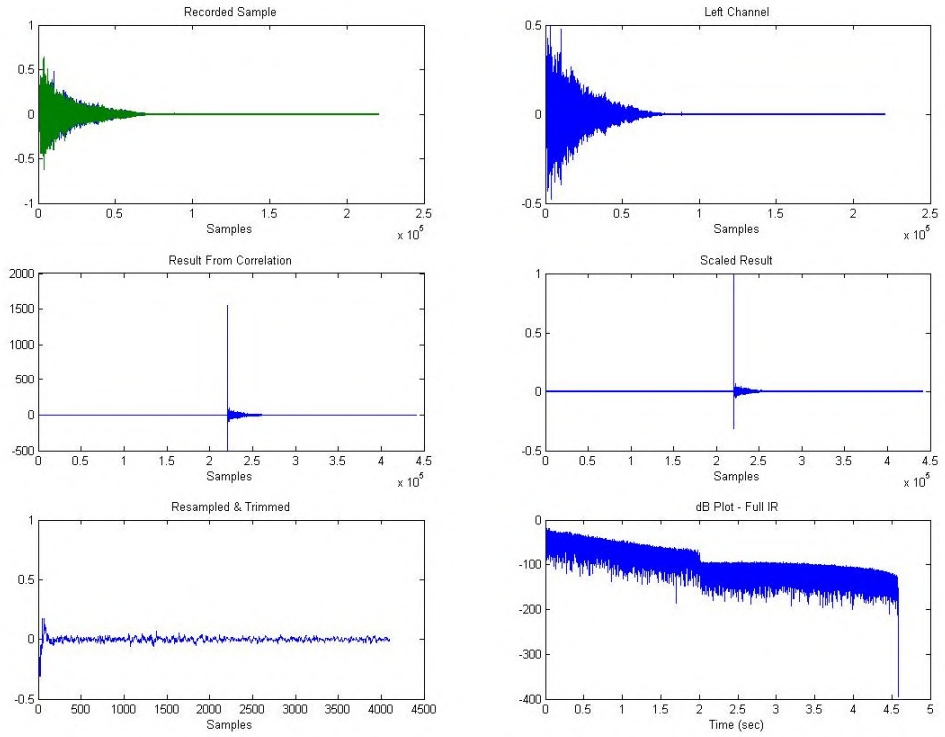

Figure 1. MATLAB Impulse Response Tracking

Impulse Response Analysis

Reverberation time is an identifying characteristic of a music hall. This is defined as the time required for the sound to decay to 60dB below the direct sound. Mr. Ray Clover, from the Pittsburgh Symphony, specially requested we give him the reverberation time of Heinz Hall in order to compare these values with those of the acousticians that have been through Heinz Hall. In the process, it made sense to measure it for the other places as well as an extremely interesting aside. This also is a statement for the quality of the Impulse Response recordings. Below is a table with the locations and reverb times. Of note, is that in doing research, it was found that the typical bedroom or living room reverberation time is 0.4 seconds. In our recording, we have found the reverberation time of a living room to be 0.4196

seconds. Also, it is interesting to see that the average time in Heinz Hall is 1.402 seconds and the time for the normal Orchestra seat in row K is 1.403. In order to experience the best representation of a concert hall, the optimal seats are those on the main level before the balcony is overhead.

Location	***Reverberation Time (s)***
Alumni Concert Hall	*1.963*
Kresge Concert Hall	*0.8938*
Chosky Theater	*0.6439*
Soldiers And Sailors Memorial Hall	*0.8242*
Heinz Hall – Orchestra Row K	*1.403 (seen above)*
Heinz Hall – Orchestra Row S	*1.345*
Heinz Hall – Grand Tier	*1.355*
Heinz Hall – Dress Circle (Balcony)	*1.505*
Heinz Hall – Average	*1.402*
Living Room	*0.4196*

Figure 2. Reverberation Time Table

William Gardner's Algorithm

Our project is based on an algorithm written by William Gardner in his 1994 Audio Engineering Society paper, "Efficient Convolution without Input/Output Delay" [1]. This paper came about from the fact that in order to use a two second impulse response with an FIR filter, it would require an 88,200 point filter at 44.1 kHz sampling rate. The proposed methods that are more realistic than this are a block fast Fourier transform (FFT) method using overlap-add or overlap-save. However, these methods cause serious delay as the processor must wait for an entire block of input samples to be ready, then perform the calculations, and finally have a block of output samples available. In order to get around this delay, there are two important steps to take.

First, the beginning block must be done with a direct form FIR filter in order to start without delay. Second, a method of increasing block size will not add delay as the results of each block can be computed and summed together by the proper time. The scheduling seen below is Gardner's solution (for a filter response of size 16N) that calls for constant demand on the processor. He defines N as the smallest block size for which block convolution is faster than the direct form FIR filter. The constant demand comes from the fact that each block of size B starts 2B samples into the filter response. For example, with the first value in the parenthesis representing the starting point of the convolution, it is seen that at 4N, the 2N blocks start.

The next important part of this algorithm is the scheduling of these computations. The blocks that need to be computed in the shortest amount of time must take the highest priority. Therefore, the N point blocks have the most priority, then the 2N blocks, and finally the lowest priority goes to the 4N blocks. All of these guidelines continue in this fashion for a larger impulse response.

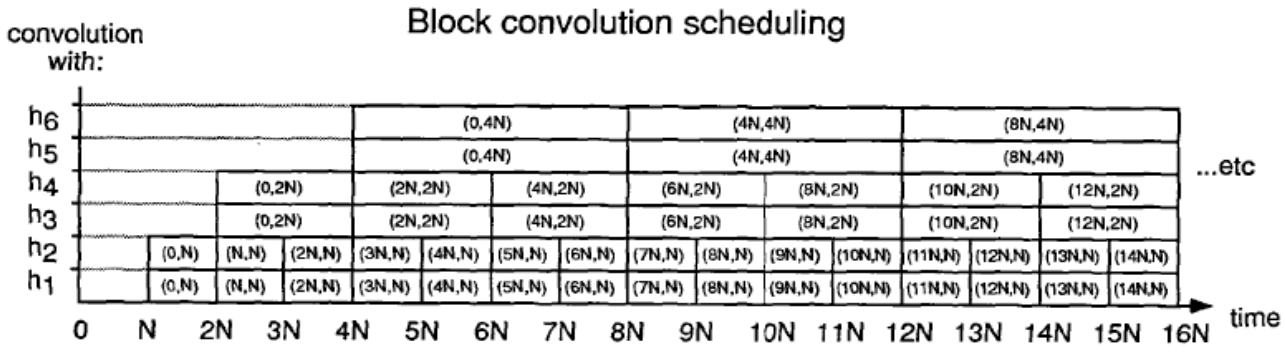

Figure 5. Practical decomposition of filter response (top) and corresponding block convolution scheduling (bottom).

Figure 3. Gardner's Proposed Block Schedule

Block Decomposition

The block decomposition was structured in the same way as recommended in Gardner's paper due to the constant processor demand that is involved. This meant an impulse response had to be divided into segments of 2N, N, N, 2N, 2N, 4N, 4N, etc. Two deciding factors paid a role in determining the constant N. Since the h0 block of size 2N was convolved with the current input sample, the timing of this function depends heavily on the size of N. As can be easily seen in the diagram below, the speed of the convolution is increasing linearly with increasing size of N.

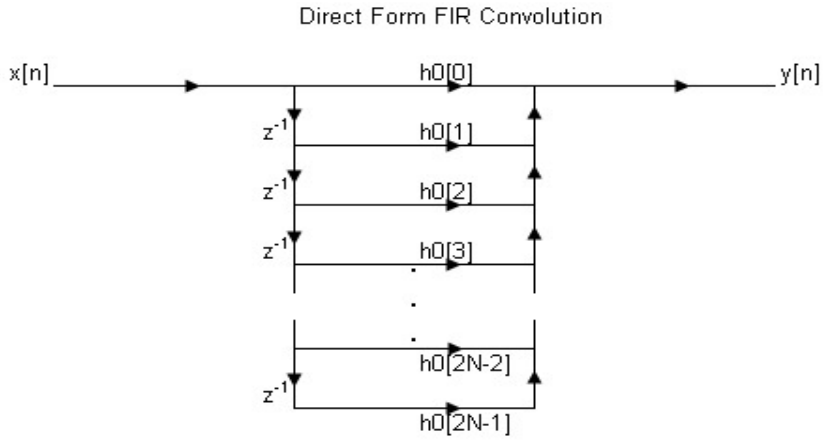

Figure 4. Convolution speed of input sample with h0 block depends on variable N

*The second factor was the limitations on the size of the FFTs. Due to memory constraints, the maximum FFT that we could perform in internal memory with the Texas Instruments DSPF_sp_fftSPxSP function was a 1,024 point FFT requiring 2,048 point input, output and twiddle factor arrays. However, using the custom made radix-2 function, a maximum FFT size of 2,048 could be achieved. This provided the additional constraint that the largest block size, i, would have to equal 8 as i*N = 2048 must be satisfied. From these two requirements, we successfully implemented the following block decomposition method. The sequence of blocks was 2N, N, N, 2N, 2N, 4N, 4N, 8N, 8N where N=128. This corresponds to a time domain impulse response of 4,096 points.*

Below is the scheduling for these 8 blocks, which will be used in the block convolution. The y-axis represents the IR blocks and the x-axis represents the input. For example, after 2N input samples have passed, comp_block() will be called for h1&h2 blocks and for the h3&h4 blocks. At 3N comp_block() is called for h1&h2 again as another N new input samples are available for block convolution.

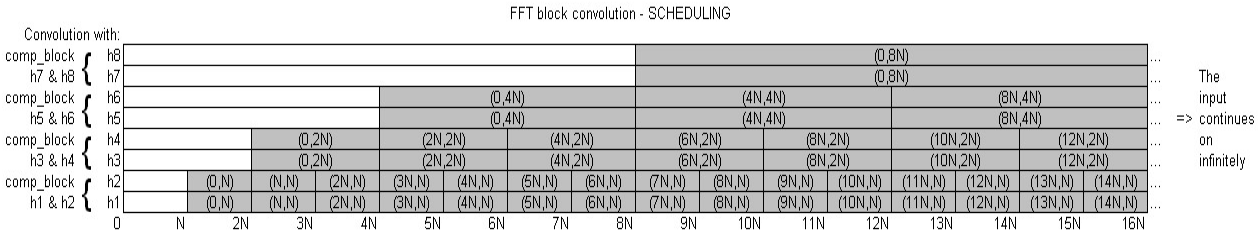

Figure 5. Scheduling as implemented

Constraints

The C67 DSP chip has a 4.4 ns instruction cycle. Using a sampling frequency of 48kHz, as initially proposed, would have required us to compute an output sample in 4,735 cycles for a zero delay implementation. By the update presentation we had already collected enough data on FFT timing to see

that this was not a realistic outcome, and subsequently proposed using a sampling frequency of 24 kHz. The chart below shows the theoretically available cycle times for sampling frequencies we considered.

Theoretical cycle times available

Processor cycle time (sec)	Sampling frequency (Hz)	Cycles available between interrupts
4.4E-09	48000	4735
	24000	9470
	16000	14205

Figure 6. Maximum available cycle times per output sample for a zero delay implementation.

The chart below shows the timing of all functions that were called during interrupts. These are average values, so although the total number of cycles needed is listed as approximately 7,550, running the algorithm at 24kHz, that is with a maximum number of 9,470 cycles available, produces some error as the number of cycles in between any two interrupts may well be slightly above 9470 causing the algorithm to fail.

	cycles per calculation	sampling period to finish calculation	cycles per sampling period
h0 compuation	750	1	750
xmit Interrupt call	382	1	382
rcv Interrupt call	378	1	378
h1h2 transfers	3744	128	29.25
h1h2 computation	235792	128	1842.125
h3h4 transfer	4032	256	15.75
h3h4 computation	245800	256	960.15625
h5h6 transfer	14976	512	29.25
h5h6 computation	1073118	512	2095.933594
h7h8 transfer	16128	1024	15.75
h7h8 computation	1081866	1024	1056.509766
		Total Cycles per Sampling Period	7554.724609

Figure 7. Cycles Count. Transfers refer to costly EDMA transfers that must happen before computation of the blocks. Computations refer to the actual block computations.

FFT: Early Efforts

A clear advantage in this project is that we used the DSK from very early on, only resorting to MATLAB to check results and run quick calculations. Most of the first few weeks were spent searching for a suitable FFT function and running timings for different sized FFTs. In particular, we were searching for a radix-2 FFT since the most efficient block size decomposition, according to Gardner, involves increasing every other block by a factor of 2. Furthermore, we were initially looking for a cache-optimized function. However, due to the large numbers of FFTs we calculate, this factor was only of secondary importance since the advantage of cache-optimized FFTs becomes negligible for large numbers of calculations. Initially we looked at the TI DSPF_sp_fftSPxSP, a mixed radix, cache-optimized FFT with complex input. Unfortunately, using this function as a radix-2 FFT did not work with the twiddle factor table that TI provided specifically for this function. Further disadvantages with this function were that the input was complex interleaved and we were only using real inputs causing our input block and IR block array to only use half the space. Since we were taking two N-point FFTs, multiplying them, and then taking the N-point IFFT, we could only fill half the input and IR blocks in the first place. This means that we could only use 25% of the input array's total size. This situation is shown in the diagram below.

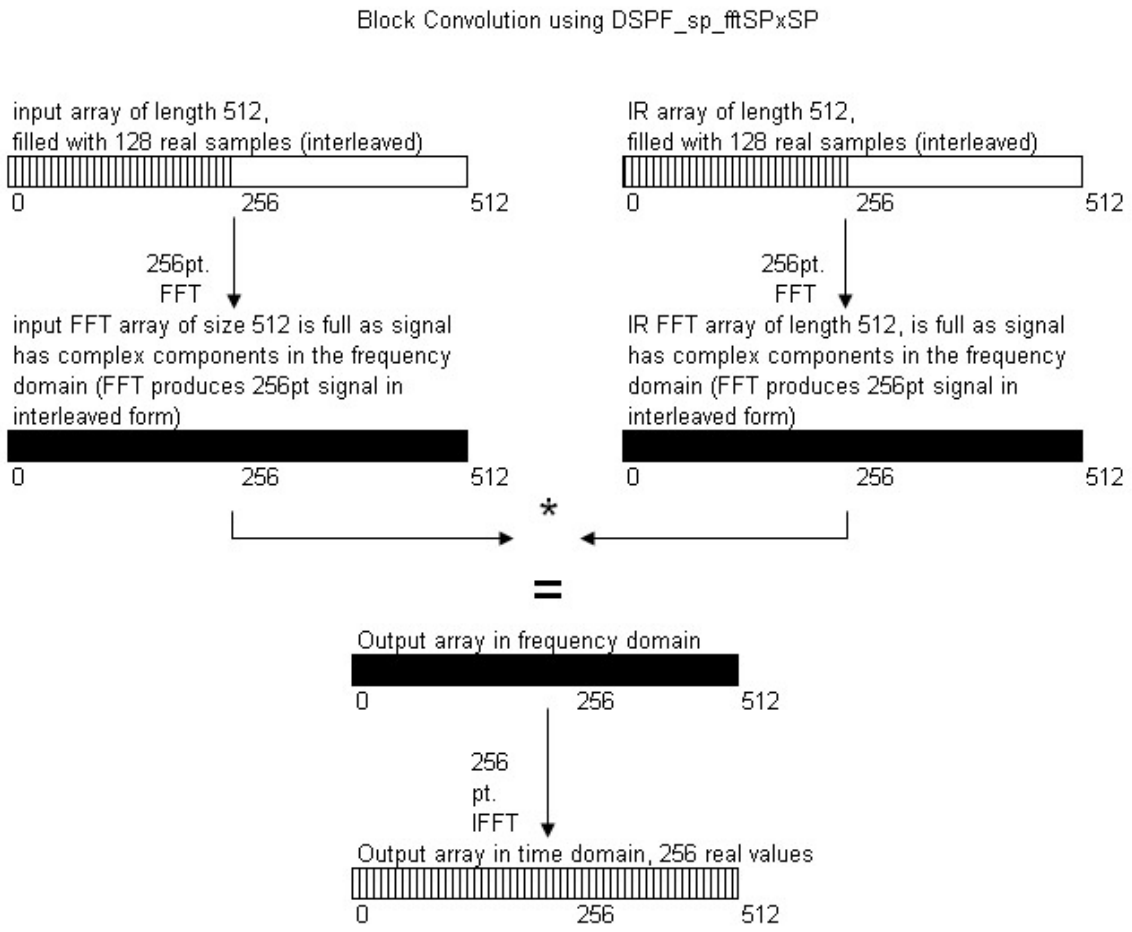

Figure 8. Diagram of memory inefficiencies of using DSPF_sp_fftSPxSP

For example, in order to convolve a 128 pt. input block with a 128 pt. IR block, we would need to use 256 point FFTs. This would mean the input and output arrays as well as the twiddle factor array must be of length 512 as they are complex interleaved. In order to produce 256 output samples, we use at least 7 arrays (twiddle table, input in time, input in frequency, IR in time, IR in frequency, output in frequency, output in time) of 512 floats each. A somewhat aggravating side effect was that this TI function would overwrite the input array (but not with any useful data). Unfortunately, there was no real FFT function in the library nor in any other C6000 libraries we browsed through. The non-cache optimized function DSPF_sp_cfftr4_dif was another function we looked at due to the favorable property that it would

overwrite the input buffer, therefore requiring 3 less FFTs. However, this function did not have an associated inverse FFT function and also didn't fit our initial criteria of a radix-2 FFT function.

In implementing these FFTs, we found out that we could not link the complete library, so we used C-callable ASM files taken from the source file instead. This worked fine for the initial testing, but raised concerns when calling these FFTs during interrupts, due to their limited interruptibility. Since the C equivalent code of the TI FFT functions did not suffer from interruptibility issues, we used these instead although they were slightly slower than their ASM counterparts. Furthermore, the code for the IFFT function contained a bug (outputting the data in the wrong order and not scaling) that we fixed to produce the correct output.

Real FFT

Implementing Fast Fourier Transform Algorithms of Real-Valued Sequence with the TMS320 DSP Platform *by Robert Matusiak describes a method of efficient DFT computation for real-valued numbers [3]. This method allows us to compute a 2N length real sequence in an N point FFT, and compute its IFFT using an N point IFFT. The documentation by Matusiak also included C code that implements the described methods but we did not use them as they were written for 16 bit integers and we are using 32 bit floats. Below describes the steps in computing the real FFT.*

Suppose g(n) is a purely real 2N point sequence of numbers. Define

x1(n) = g(2n)

x2(n)=g(2n+1)

for n=0 to N-1

the DFT of g(n) can be computed using

x(n) = x1(n) +x2(n)j

and G(k) = X(k)A(k) + X(N-k)B(k)*

for k=0 to N-1

A(k) = ½ (1 – jW $^{k}_{2N}$) B(k)= ½ (1 + jW $^{k}_{2N}$)

Similarly for an inverse FFT computed from a real sequence of length 2N, we can compute the answer using an N point IFFT.

G(k) is the input

X(k)= G(k)A(k) +G*(N-k)B(k)*

For k=0 to N-1 , so we omit the symmetric values in G(k)

If we take the IFFT of X(k), we obtain the IFFT of G(k), g(n)

Using this real FFT method, we are able to compute an FFT of 2N real samples with an N point FFT. This gives us tremendous computational savings as we can use a FFT that is half the length for each block. We are also able to save memory since we no longer have to store data in a redundant complex interleaved form.

EDMA Transfers

Some EDMA transfers are costly in that they must be finished before any computation can begin. Other transfers can be performed for free asynchronously, under the assumption that access to this data will

not be needed before the transfer finishes. For example, we can transfer the output buffer into internal memory while the FFT is being computed, since the FFT will take from 250,000 to a million cycles, which is far greater than the cycles that an EDMA transfer will take.

The below steps explain how the transfer works in order to compute the FFT convolution of two blocks of the same length (i.e., h1 and h2).

1)Load twiddle table, load precomputed FFT of impulse, load input samples from input buffer (These are the only costly loads, they must finish before we can compute the FFT!)

2)Load portions of the output buffer for the first block, compute FFT convolution for the first block.

3)Add computed FFT convolution to the loaded output buffer.

4)Send the added block to output buffer in external memory. Load new output buffer for the second block. Compute FFT convolution for the second block.

5)Add computed FFT convolution to the loaded output buffer.

6)Send the added block to external memory.

As can be seen, all transfers except those in step 1 can happen for free computationally.

Streamlining and Optimization

To reduce the number of calculations needed for the block convolution, the impulse response was sent in mixed form, meaning that the first block was represented in the time domain, but all subsequent blocks were already converted into the frequency domain. This was very useful, as it allowed us to perform one less FFT per block, and was also more efficient because the impulse responses were loaded into a .txt file ahead of time. Furthermore, since the FFT of a real-time signal is symmetric, we could exploit that property to save space by cutting all of the frequency domain representations down to half the size. Taking this into consideration, the impulse responses of 4,096 time samples were now represented in mixed form of size 10,496. The figure below illustrates specifically how the mixed form representation is concluded to be of 10,496 samples.

Block Decomposition

Block	Block Size	Time Domain Length	Mixed Representation Domain	FFT size (complex pts)	Mixed Representation Length (10496 samples)
h0	2N	256	Time	N/A	256 (purely real)
h1	N	128	Frequency	512	512 (complex interleaved)
h2	N	128	Frequency	512	512 (complex interleaved)
h3	2N	256	Frequency	512	512 (complex interleaved)
h4	2N	256	Frequency	512	512 (complex interleaved)
h5	4N	512	Frequency	2048	2048 (complex interleaved)
h6	4N	512	Frequency	2048	2048 (complex interleaved)
h7	8N	1024	Frequency	2048	2048 (complex interleaved)
h8	8N	1024	Frequency	2048	2048 (complex interleaved)

These values are half of the FFT size, because FTs of real samples in time are symmetric. By exploiting this property, we only have to use half the space.

Figure 9. Block decomposition of impulse responses

In order to achieve a working implementation of the algorithm under strict timing and memory constraints, we devoted a large amount of time to thinking about how to cut down cycle times of our functions. Of the functions called during interrupts, we were primarily concerned with fir_conv() and comp_block(), as well as comp_block()'s sub functions, do_fft_block1() and do_fft_block2(). These last two functions represent the convolution of the first and second block of any block pair (such as h1&h2, h3&h4, etc.).

The speed of fir_conv(), the function that performs direct form time domain convolution, was greatly increased when we got rid of the mod operators. On an optimization level of (–o1) and (–op3) getting rid of mod operators reduced the cycle time for fir_conv() from 15,922 cycles to 3,642 cycles. By placing this function in a separate C-file, and setting file specific optimization levels at (-o3) and (-op3), the cycle times for fir_conv() was reduced even further to 662 cycles.

The comp_block() function convolves two same sized blocks, for example h1 and h2. Since they both are being convolved with the same exact input segment, the FFT of this input segment only needs to be calculated once. Since the h1 and h2 blocks are already represented in the frequency domain, the only other FFTs needed are two IFFTs to convert the output of the two block convolutions into the time domain. This means that we will need a total of 3 FFTs per comp_block(), as opposed to the intuitive 6 FFTs (1 FFT for the input block, 1 FFT for the impulse response block, and 1 IFFT for the output block for each of the two block convolutions).

PC-DSK - client-server relationship

The initial setup had the DSK board being the client and the PC acting as the server. This was an intuitive choice as it allowed us to request data to be sent to the DSK, process it and subsequently send it back to

the PC. The clear advantage is that the DSK knows when data is ready to be sent, and it can be sent as soon as the requestTransfer() function is called on the DSK side. Meanwhile the PC side was idling in a while loop waiting for a message from the DSK. If the PC were the client, and therefore calling the requestTransfer() function, there is no way of telling whether or not the DSK has already completed all the processing. This could mean transfers wouldn't occur, as the timeout values of the send and receive functions are reached before the processing is finished. For this reason we started off with the DSK being the client (equivalent to setup in lab 3).

This proved to be useful for testing as we would load a .txt file containing the IR into the PC side code, call requestTransfer() on the DSK side, then perform a convolution, and finally send it back to the PC side, where it would be written to a .txt file that could easily be used to compare values with MATLAB. We stuck with this method until the algorithm first worked, at which point we thought of the details concerning the synchronization of the GUI with our code.

The issue we ran into when using the DSK as the client was that commands from the GUI should be sent to the DSK as messages from the PC side (which would lie integrated with the GUI code). This would require the PC side code to call requestTransfer(). For our application we needed the PC side code to let the user choose an IR before the algorithm would run. At any point after that we needed to be able to send messages to the DSK to alter specific settings such as volume. To accommodate this we would have to implement receiving data while within an interrupt.

Changing the code from the DSK being the client to the PC being the client presented us with relatively few difficulties as we were introduced to both setups in labs 2 and 3. One concern was the timeout values in the net_recv_ready() and net_send_ready() functions on the DSK side. We were constrained by how large the timeout values of these functions can be since they are being called from inside an interrupt. We had to make sure that the number of available cycles during an interrupt was large enough to

accommodate both the transfer of data and the subsequent changes, in addition to the rest of the algorithm. We ended up giving net_recv_transfer() on the DSK side a timeout value of only 5 cycles.

Future Improvements

The efficiency of the algorithm clearly depends on the efficiency of the FFT that is used. A realistic implementation of this algorithm would have to take advantage of real valued inputs, and must be generous with the amount of memory that it will use for computation (an in place algorithm would be optimal).

Many other FFT algorithms exists that take advantage of real valued inputs. For example, Sorensen's real FFT algorithm eliminates redundant calculations within the proposed butterfly structure. It is more efficient than the "packing algorithm" that we used since it require no extra operation to transform the output back to its expected form [4]. There are many other alternatives, including Brunn's FFT algorithm, the Fast Hartley Transform, and the Fast Cosine Transform. It is unclear which of these methods will perform best on the c67x architecture, since it requires more analysis than counting the number of multiplication and additions. Future groups could investigate the efficiency of each of these algorithms, and implement the best one.

We also implemented the algorithm in floating point, and it would have been computationally more efficient o implement it in fixed point. Although TI's fixed-point implementation is not quite trivial, it would give us a worthwhile speed increase.

Adding Delay

Gardner suggests that it is possible to trade off delay for decreased computational cost. He proposes to get rid of the FIR filter and replace it with an FFT convolution. This restricts us to having 256 samples of delay in our algorithm, so we propose an alternate solution that is more flexible.

*We can improve the efficiency of our code tremendously by introducing inaudible delay into our system. Various studies and papers suggest anything less than 10 ms of delay is undetectable by the human ear, which translates to one hundredth of the sampling frequency (160 sampling periods at 16 kHz). If we incorporate this amount of delay in our algorithm, for computations of each block, we will get an additional 160 sampling periods to finish the calculations. This buys us 160*14,205 = 2,272,800 more cycles per block calculation. Note that this does not change any calculations in our algorithm, just the time that they are due.*

We can do a lot with these extra cycles, by either increasing the sampling rate or increasing the block length (or both). With 2,272,800 cycles, we can add 5 extra blocks of size 1,024, increasing our total impulse length to 9,216 samples. The figure below shows how the new scheduling will work with this implementation.

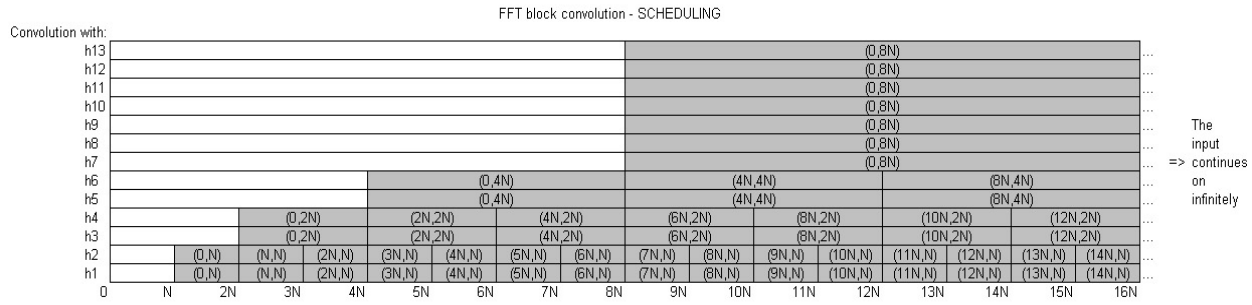

Figure 10. Scheduling proposed if delay is acceptable

We can also increase the sampling rate by adding delay. Our code includes a test within the interrupts to see if the algorithm is running in time. Basically, if the computation of a block does not finish before the scheduled time, it gives you a warning and exits. We modified this code so that you can now specify the delay and move back the scheduled time when the computation is due. Running some tests with this new code, we are able to run at 24 kHz with just two samples of delay. We were also able to run the code at 48 kHz with 500 samples of delay. Below shows how this is possible computationally.

		cycles per calculation	sampling periods to finish calculation	cycles per sampling period
h0 compuation		750	1	750
xmit Interrupt call		382	1	382
rcv Interrupt call		378	1	378
h1h2 transfers		3744	628	5.961783439
h1h2 computation		235792	628	375.4649682
h3h4 transfer		4032	756	5.333333333
h3h4 computation		245800	756	325.1322751
h5h6 transfer		14976	1012	14.79841897
h5h6 computation		1073118	1012	1060.393281
h7h8 transfer		16128	1524	10.58267717
h7h8 computation		1081866	1524	709.8858268
			Total Cycles per Sampling Period	4017.552564
			Total Cycles available	4735

Figure 11. Shows the computational cost of running the algorithm at 48kHz with 500 samples of delay. Notice that this is similar to Figure 7, except that there are 500 extra sampling periods to finish calculations for each FFT convolution block

GUI Design

Figure 11. Initial GUI view

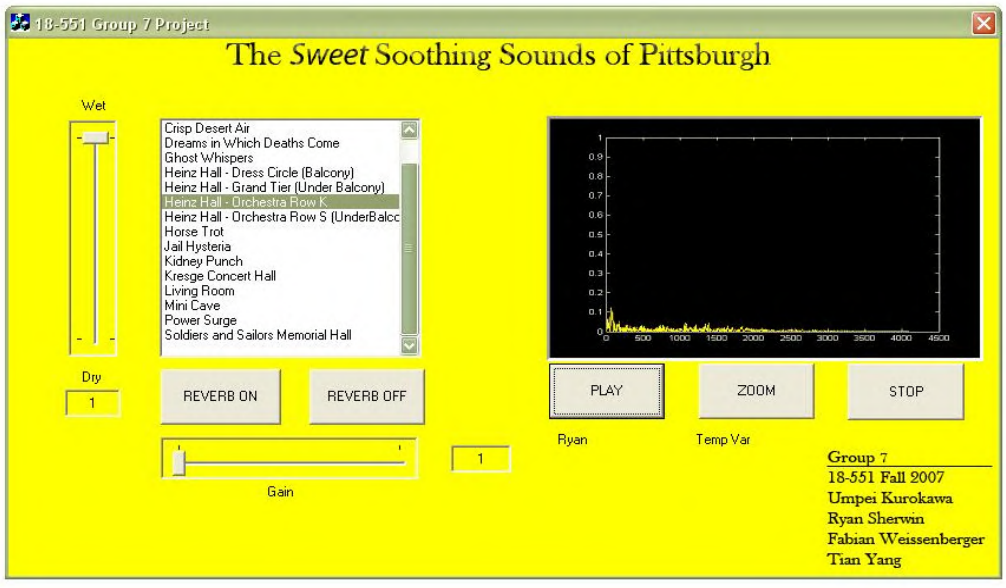

Figure 12. Loaded impulse response scaled from 0 to 1

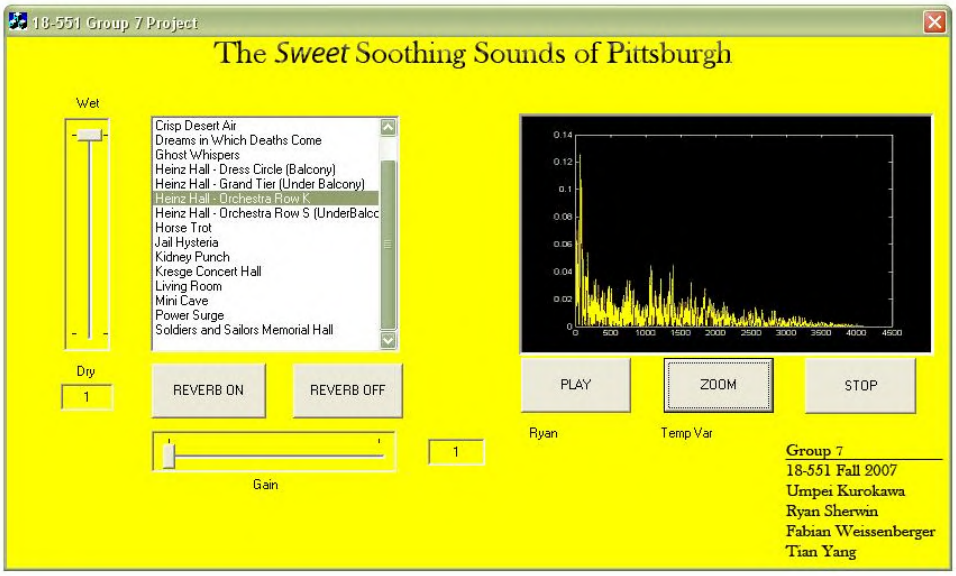

Figure 13. Zoomed impulse response

GUI Overview

To provide a front-end interface for the user, we created an MFC application using Visual C++. Several existing professional applications were used as guidance for the layout, including Sony SoundForge. Relevant features included selecting the different sound environment to be loaded, a wet/dry control, a volume control, and a graphic displaying the impulse response in time of the environment. Playback and stop controls are also assigned to buttons and sent to the DSK.

GUI Features

The wet/dry control is normalized from 1 to 100 and sent to the DSK. The value must be selected by dragging the slider to initialize the variable. A value of 1 equates to fully dry, which implies that the listener will only hear the original input. A value of 100 would be the input with full reverberation.

Therefore, a value of 50 is the middle point, which is 50% wet and 50% dry. The gain control operates in a similar fashion and uses a horizontal scroll to set the value of the gain. The names of the individual environments are displayed in a list box. They are then selected and loaded by the play button. There also is a control to display a zoomed view of the impulse response for better detail. The stop button commands the DSK to quit and allows the user to change the environment, the wet/dry, and the gain.

GUI Implementation

Our GUI was implemented with MFC classes as suggested by one of the TAs. When linking the GUI, we basically tied the PC side code into the C++ code that was written for the GUI. By connecting function calls to specific GUI buttons, we made sure that any given button would call the appropriate requestTransfer() option to send the message to the DSK. This program runs concurrently with Code Composer to execute commands on our DSK.

GUI Drawbacks

A few features could not be fully integrated, as originally designed. The scroll slider bar in MFC was unable to execute in real-time, thus affecting the functionality of the gain and wet/dry controls. In order to change the controls, the user must stop the reverb, change the controls to the desired values, and then start the reverb again. The user cannot go from one environment to another without pressing stop first since loading an impulse response is lengthy and thus not probable during an interrupt. Pressing play without stop beforehand will result in an endless loop and bug occurring.

Schedule

The general overview of the project is as follows. As aspected, there was much overlap in this project, but these are rough guidelines. We initially started with Fabian and Ryan researching the Texas Instruments FFTs and doing the necessary timings to determine which would be suitable for this project. Meanwhile, Umpei started implementing the algorithm in C code on the PC. Through much tribulation (adding libraries, no radix-2 FFT, etc.), it was determined that the Texas Instrument FFTs would not suffice as provided. The details of this have been discussed above. At this point, Umpei transferred to working on creating a real-valued FFT, Fabian concentrated on transfers and reading/writing data to file, and Ryan began recording the impulse responses. Umpei's work on combining the FFT and algorithm was intensive and took a few weeks. In this time, Ryan had recorded all of the impulse responses and did the necessary processing to create what would be used in the project and Fabian finished work on the above tasks and started the PC-DSK transfer code. By the time Ryan completed the impulse responses, Umpei and Fabian were nearly finished with their work. However, with time being of essence, Ryan helped both Umpei and Fabian complete their tasks by profiling functions, optimizing code, and attempting to aide in the PC-DSK transfer. By now, Tim had finished the front-end of the GUI. Combining everything, we all took part in linking the GUI.

References

[1] Gardner, William. "Efficient Convolution without Input/Output Delay". Audio Engineering Society: 1994, MIT.

- *Real-time algorithm implemented on C67*

[2] Edwards, John. "Acoustic Room Response Analysis". TechOnLine: June, 1997. <http://archived.techonline.com/community/ed_resource/feature_article/20119>.

- Impulse Response Collection Resource

[3] Matusiak, Robert. "Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS320 DSP Platform".

- Real FFT resource

[4] Toth, Laszlo. "Experience with Real Valued FFT Algorithms." Research Group on Artificial Intelligence. Hungarian Academy of Sciences.

- Real FFT and future improvements

[5] Spirit Canyon Audio. Sanitarium Impulse Response Database. 2005. <http://www.spiritcanyonaudio.com/sanitarium.php>.

- Extra Impulse Responses