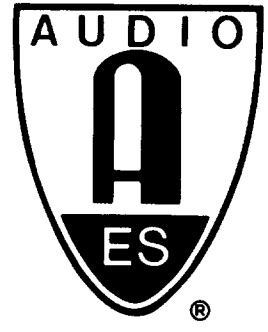
DIGITAL EQUALIZATION USING FOURIER TRANSFORM TECHNIQUES

2694 (B-2)

Barry D. Kulp
Zoran Corporation
Needham Heights, Massachusetts

Presented at the 85th Convention 1988 November 3-6 Los Angeles



AES

AN AUDIO ENGINEERING SOCIETY PREPRINT

DIGITAL EQUALIZATION USING FOURIER TRANSFORM TECHNIQUES

Barry D. Kulp
Zoran Corporation
Needham Heights, Massachusetts

Abstract

Equalization using time domain digital convolution becomes increasingly computationally intensive as impulse response length increases. Fourier transform techniques greatly reduce the computational load. The corresponding theory is reviewed, and various applications are detailed, including room, loudspeaker, instrument, and ambience equalization. A practical real-time implementation using an off-the-shelf digital signal processing integrated circuit is described. Theoretical and practical limitations of the applications and implementation are discussed.

INTRODUCTION

In this paper, we will explore both techniques and applications for performing digital equalization using Fourier transform theory. Some of the applications are ones that are not commonly thought of when the term "equalization" is used, which we normally think of as just manipulation of frequency spectrum characteristics, or filtering. For example, we will look at "ambience equalization", which we will take to mean manipulation of ambient, or reverberant, characteristics. This manipulation can include both the cancellation of a room's undesirable ambient characteristics and/or the creation of a desired ambient response.

All of the applications to be described, from simple filtering to reverb generation, can be achieved in the digital domain using time domain

convolution techniques. However, in many cases, the amount of computation required to perform the convolution directly in the time domain would be prohibitive, relative to practical constraints of processing time and/or hardware cost. Fortunately, frequency domain processing provides a solution that greatly reduces the computational load involved in performing these convolutions.

It will be assumed that the reader has some basic knowledge of digital signal processing: that signals can be sampled, that the samples can be combined using delay registers, multipliers and adders in some way to do filtering, and that Fourier transforms exist and that they transform a signal from its time domain representation (the samples) to the frequency domain (a sampled spectrum) in a manner similar to the Laplace transforms of analog signal processing. Building on this basic knowledge, we will briefly review digital convolution and the Finite Impulse Response (FIR) filter structure. From there we will explore the theory behind using Fourier transforms to perform "fast convolution", greatly reducing the computational load as promised.

After exploring the theory, we will then look at some of the many applications using these techniques. Next, we will look at a hardware solution enabling real-time implementation of the algorithms involved. Finally, we will examine some of the limitations and problems encountered in both the applications and the implementation, and discuss some ways to solve them.

CONVOLUTION THEORY AND FILTER STRUCTURES

Normally, when one thinks of a filter, one conceptualizes it as a change in the frequency domain characteristics (spectrum) of the signal. For example, a low-pass filter will allow the lower portions of the spectrum to pass through from input to output, but not the higher portions. However, due to the duality characteristics of the time and frequency domains, we know that multiplication in the frequency domain is equivalent to convolution in the time domain. In other words, to filter a signal with a filter having a certain frequency characteristic, the result being a signal with a spectrum that is the product of the spectra of the input signal and the filter, we must convolve (in the time domain) the

input signal with the impulse response of the filter. The impulse response can be thought of as a signal whose spectrum is the frequency response of the filter.

In the continuous time domain (the analog world), we don't actually set out to perform the convolution. Instead, we try to design a circuit that has the desired frequency response (spectrum), which just happens to have a certain impulse response that we can measure, which it convolves with any input signal, the result of which we can observe. So, while the circuit is actually doing the convolution, we don't normally think of the filter in that way. Instead, when we talk about the filter, we talk about its frequency-domain function, e.g. low-pass, high-pass, band-reject, etc. Further evidence of this mind-set is the fact that analog filter design programs are designed to convert back and forth between spectral characteristics and component values for resistors and capacitors in a particular circuit topography, with the concept of impulse response not always directly addressed (although some filter design packages will plot the impulse response and/or the step response of the circuit).

However, in the discrete time domain (the digital world), in addition to thinking in terms of the frequency-domain function, we also often think in terms of convolutions and impulse responses. This is evidenced by the fact that digital filters are often described as infinite-impulse-response (IIR) or finite-impulse-response (FIR). In fact, FIR filters are usually implemented in such a way as to directly perform the convolution on the input samples by doing a sum-of-products calculation. Also, FIR filter design programs convert back and forth between spectral characteristics and impulse response coefficient values.

Within the realm of digital filtering, there are various tradeoffs between the IIR and FIR filter structures. The advantages of FIR filters include: absolute guarantee of stability, well-behaved round-off error characteristics, ability to realize arbitrary frequency responses, linear phase, etc. The disadvantages of FIR filters are basically just delay time (particularly if using the linear phase structure) and computational load, which increases directly with impulse response length, which in turn grows as the "precision" of the filter increases. By "precision", I am referring to how closely the actual filter's frequency response varies from an "ideal" one, in such areas as ripple and transition band width.

Typically, FIR filters are implemented as a sum of products. This is easily conceptualized as follows: Because the filter is a linear system, we

may use superposition principles to manipulate our analysis of it. Consider the input sequence as a sum of many different sequences, each one consisting of all zeroes except for one unique point. In other words, the input to the system is a sum of many different impulses, each with its own amplitude and unique position in time. Therefore, the output of the filter will be simply a sum of many copies of the impulse response, each one shifted to a unique starting point in time, with a corresponding amplitude gain factor. If we look at any given output point, we note that it will have contributions from many of these various copies of the impulse response. In fact, if the length of the impulse response (the number of points between the first and last non-zero ones, inclusive) is N samples, then every input point will affect N output points, and conversely the output at any point in time will be affected by N input points (the current one and the N - 1 previous ones).

As a simple example, suppose the impulse response is only 3 samples long, and their values are 0.4, 0.3, 0.2. This means that each input point will contribute 0.4 times its own amplitude to the current output point, 0.3 times to the next one, and 0.2 times to the following one. Turn this around and we can see that each output point is the sum of 0.4 times the current input point, 0.3 times the previous input point, and 0.2 times the input point from 2 sample periods prior. The structure normally used to implement this is shown in Figure 1. It demonstrates both of these relationships: as each input point travels down the series of delay taps, it will contribute first 0.4, then 0.3, and finally 0.2 times its own amplitude to the output value before being "forgotten"; conversely, the output is a sum of the 3 most recent inputs, with the weighting factors correspondingly applied.

To repeat an important point for emphasis: the longer the impulse response, the more multiplications and additions must be done in the same amount of time, each sample period.

One thing to keep in mind about an FIR filter: it is not necessarily a "filter" in the normal sense of the term. It is, in fact, just a circuit or system that convolves an input signal with some finite impulse response. The impulse response, in turn, is just a series of delayed impulses, which create a series of delayed copies of the original input signal, multiplied by various weighting factors (coefficients, or you could even call them gain factors). Therefore, a sufficiently long FIR "filter" could perform functions normally thought of as "delay-line" functions, or be used to realize any arbitrary impulse response for purposes other than what would normally be called "filtering". Again, the only drawback is the

computational load normally involved in implementing long FIR filters.

All of this brings me to the main point of this paper: It is possible, using some mathematical trickery, to perform long FIR filter functions with much less computational load than would normally be required using a direct sum-of-products implementation. In the following sections, we will explore these tricks, and some of the various functions that can be performed with this structure.

FAST CONVOLUTION THEORY

In the preceeding section, we noted that the amount of computation that must be performed in order to implement an FIR filter structure grows in a direct linear fashion with the length of the filter's impulse response: one multiply and one add per filter tap per input point. Fortunately, FIR filters with long impulse response lengths may be performed with far less computational overhead using Fourier transform techniques. The basic idea at work here is the exploitation of the transform theorem, which states that "Convolution in the time domain is equivalent to multiplication in the frequency domain."

In order to perform multiplication in the frequency domain, we must be able to convert our signals back and forth between the time domain and the frequency domain. In order to do this, we will use the Fast Fourier Transform (FFT), the theory and development of which is beyond the scope of this paper. Once we have used the FFT to convert our input signal and our impulse response to their frequency domain representations, we will multiply them together to obtain the frequency domain representation of our output signal, and then perform an inverse FFT operation on that to obtain the time domain output signal.

This sounds simple enough, but in order to properly exploit this phenomenon, we must understand some additional concepts: the difference between linear convolution and circular convolution, and the two methods that may be used, overlap-and-add and overlap-and-discard. Also, it must be shown that this method, which certainly sounds more complicated than the direct time-domain sum-of-products implementation, is in fact more efficient in terms of computational load than that method.

Linear convolution is what we normally think of first when we think of convolution. It is what was described above in the "Convolution Theory and Filter Structures" section. Each input point, when passed through the filter and convolved with the impulse response, will contribute to N (where N is the number of taps, or length, of the impulse response) output points, starting with the one at the same time, and including N - 1 more following it. In this case, we can assign a second meaning to the "linear" in "linear convolution". We can say that it also means that the output sequence stretches out "linearly", i.e. in a straight line, in time following the end of the input sequence.

Let's look at an example. Suppose the length of our "filter" is 100. Therefore, each input point will affect 100 points, the "current" one and the 99 ones following it. Another way to think of this is to say that the impulse response starts at t0 and goes to t99. Now, suppose we want to "filter" an input signal that is 400 samples long (from T0 to T399). We know that the output sequence will start at T0 with a contribution from the input point at T0, but it will also stretch out timewise past T399. This is because the input point at T399 will affect 100 output points, starting at T399 and 99 more following it, out to T498. Overall, we can say that the 400 point input signal got "stretched" by 99 points by the "filter" to become a 499 point output signal. This is demonstrated in Figure 2.

Keeping that in mind, let's examine what we call "circular convolution". This is what happens when Fourier transform techniques are used to perform convolutions. The basic problem is that the transform operates on a constant number of points. That is, if we do a Fourier transform on 400 input points, we get a spectrum with 400 frequency "bins". If we then do an inverse Fourier transform on those 400 bins (even if we did the multiplication by the Fourier transform of the impulse response), we still end up with only 400 output points, not 499. What happened to the other 99 points? They ended up getting wrapped around back to the beginning of the 400 point buffer, as though it were a circular buffer with the end continuously spliced back to the beginning. These 99 points got added to the first 99 points of the output buffer, rendering both the beginning and the end of the signal invalid (there is no way to separate them back out, without just recalculating the whole thing a different way anyway).

To see how this circular convolution comes about, look at Figure 3. Keep in mind that a finite duration Fourier series carries with it the implicit assumption that it has sampled one period of a periodic waveform, as shown in Figure 3a. Figure 3b shows the results of a linear convolution on

each of the periods of this implied periodic waveform. Note that the period of the waveform is still 400 samples, but the length of each slice of it has grown to 499 points. Therefore, the end of one slice overlaps the beginning of the next slice. Figure 3c shows the actual resulting 400 point output sequence. The extra 99 output points generated by the convolution have extended over into the next implied period, and the first 99 output points have been corrupted by the extended output of the preceeding implied period. Since each implied period is identical, the net effect is that the extra points generated past the end of the sequence get wrapped around, circular buffer style, to overlap (and get added to) the beginning points of the output sequence.

The easy way to avoid the problems of circular convolution is to use Fourier transforms of a length that equals or exceeds the length of the expected resulting output sequence. In the case of our example, we expect an output of 499 points. Since we would like to maintain computational efficiency, we want to use the FFT to do our transforms (as opposed to just a direct form discrete Fourier transform). Since many FFT algorithms are based on a radix that is a power of two, a 512 point FFT would make a lot of sense. In order to use the longer length FFT, we simply need to append zeroes to the input signal to pad it to the proper length. Figure 4 shows this being done. Figure 4a shows the input signal, padded to length 512, and its FFT. Figure 4b shows the impulse response, padded to length 512, and its FFT. Figure 4c shows the result of multiplying the two FFTs in order to get the FFT of the output sequence, which is then derived by performing an inverse FFT operation.

This is all very convenient for operating on short, fintie duration input sequences, but what happens if we have a signal that is practically infinite (i.e. expected to go on for a very long time relative to the length of the impulse response) and we want to start getting results out now? Then we break up the input signal into shorter segments for immediate processing. This is where the techniques known as overlap-and-add and overlap-and-discard get used.

The overlap-and-add method is the easier one to use to explain this concept, so that's where we'll start. To use the example we have been using, suppose we have an impulse response with a length of 100 and we want to segment the input sequence into chunks of length 400 and use FFTs of length 512 (actually, we will segment the input sequence into chunks of length 413 without any problem). We simply take each segment of the input sequence, pad it out to length 512, and process it as before. Note that each chunk

produces an output that extends into the time of the next chunk. In this region of overlap, we must add the results of the two overlapping segments to get the actual values of the output sequence. What has happened is this: we have created a turn-on transient of invalid data (since it is missing the contribution from the final 99 points in the preceeding segment) and a turn-off transient of invalid data (since it is only the contribution from the preceeding points in this segment, and not the contribution from the "current" points in the following segment). In between we only have 314 points of valid data. In order to get 413 valid output points from the 413 valid input points we started with, we must add the turn-off transient from one segment to the turn-on transient of the following segment in order to get valid output points in the overlap region. This is shown in Figure 5.

The overlap-and-discard method is a little bit different from the overlap-and-add method. It allows us to do basically the same thing without having to do the extra addition steps in the overlap region. This is done by actually allowing circular convolution to occur. We still segment the input sequence every 413 samples, but now we take a whole 512 point sequence to do the processing on. As a result, we get 413 valid output points. The easy way to picture this is to realize that, by segmenting the input sequence in this way, we produce a sequence that would be 611 points long if it were linearly convolved: 99 points of turn-on transient, 413 points of valid output, and 99 points of turn-off transient. Since we are using a 512 point FFT, circular convolution will cause the turn-on and turn-off transients to overlap each other and become totally invalid. These 99 points get discarded, leaving us with our 413 valid output points, without having to do any addition in the overlap region (we did our overlapping in the input segmentation and circular convolution). This is shown in Figure 6.

To conclude our discussion of the fast convolution technique, let's examine the computational differences between this technique and the direct time-domain sum-of-products convolution technique. To continue with our example of a 413 point input sequence and a 100 point impulse response, it is easy to see that, since we must do a multiply for each "tap" in the filter for each output point we generate, we will do 51,200 multiplies, processing in the time-domain (if we are very smart about not processing input points that don't exist, we can get away with only 41,300 multiplies, or 100 multiplies per output point, the same result we would have if we were just continually processing an "infinite" input sequence).

To see how much processing is needed in the frequency domain, we first need to figure out how many multiplies are involved in doing a 512 point FFT.

If we implement it in radix-2 form, we do 9 passes, 256 butterflies per pass, and 4 multiplies (actually one complex multiply) per butterfly, for a total of 9216 multiplies. If we then say we have to do three FFTs, one on the input sequence, one on the impulse response, and the inverse one on the result of multiplying the other two, we get 3 X 9216 = 27,648 multiplies. If we also do 512 complex multiplies to multiply the FFTs, we add 2048 real multiplies for a grand total of 29,696 multiplies. This is not yet a very dramatic improvement.

Suppose, however, that we are processing a longer sequence (or a bunch of shorter ones) with the same filter. In this case, we can pre-compute the FFT of the impulse response just once and store it for use many times, just like we would store the impulse response for the time domain convolution, and not keep recalculating it from the filter design parameters for every sample. This reduces our number of multiplies to 20,480. Okay, we've gotten down to about 2 or 2.5 to 1 improvement. Still not great.

There is one final improvement we can make. Notice that we are convolving a real signal with a real signal and getting real results. Notice also that we are doing complex FFTs, which assume that the input sequence may be complex. Well, if we convolve a complex input signal with a real impulse response, we get a complex output signal whose real part is the convolution of the real part of the input sequence with the impulse response, and whose imaginary part is the convolution of the imaginary part of the input sequence with the impulse response. Note that the real and imaginary parts stay separate! (As long as the impulse response is real). We can exploit this by actually processing two input sequences at once (or two segments of a longer input sequence), by loading one in as the real part and the other as the imaginary part, doing complex FFTs and multiplies, and storing out the real part of the result as the first output sequence, and the imaginary part of the result as the second output sequence. Therefore, the computational load gets cut in half of what we just computed, for a total of 10,240 multiplies per 413 point input chunk. This is less than 25 multiplies per output point. Now we have an improvement of about 4 or 5 to 1!

The improvement gets even more dramatic as the filter length increases. As an example, suppose the impulse response is 16,385 samples long! Also assume that we will be continually processing an essentially infinite input sequence and the FFT of the impulse response will be pre-computed and stored. The easy part of figuring out how to compare the two methods is to note that the time-domain method takes 16,385 multiplies per output point. If we use 32K point FFTs, we can process 16,384 output points at once in the real part,

and another 16,384 output points in the imaginary part. To do so takes 2 FFTs (one forward and one inverse), each one 16,384 butterflies per pass, 15 passes, and 4 real multiplies per butterfly, for 983,040 per FFT. Twice that is 1,966,080. Add another 131,072 to do the 32K point complex multiply, and get a grand total of 2,097,152 multiplies per 32,768 output points, or only 64 multiplies per output point. This represents an improvement of over 256 to 1!

A similar analysis done for an impulse response length of 131,073 points and using 262,144 point FFTs yields a result of only 76 multiplies per output point, an improvement of 1724.6 to 1 over the time domain requirements.

Note that the computational load, which grows in a direct linear relationship with filter length in the time domain method, grows very slowly as the filter length increases when using the frequency domain method. This is a major advantage of frequency domain processing: it allows us to perform certain tasks that we could not previously do in the time domain under specific practical constraints of computation time and hardware cost (e.g. relatively inexpensive real-time audio processing).

APPLICATIONS

In the following sections, we will discuss how these convolution structures can be used to implement various functions, starting with ordinary filtering and including various functions other than what is normally thought of as filtering. For example, a pattern of delayed impulses, which can emulate the early reflections of a room's ambient response, can be created using convolution techniques. Ideally, this would provide no frequency coloration, and therefore this would not normally be thought of as a filter, but the FIR filter structure (implemented as a fast convolution using Fourier transform techniques) can be used to create it.

INSTRUMENT EQUALIZATION

The first application area we will look at is one that is simply a filtering application. This application, which I will call "instrument

equalization", is where frequency spectrum shaping is applied to a musical instrument (or voice or any other sound) during recording or sound reinforcement for the purpose of enhancing the sound in some (presumably) desirable way. Typically this is done by a graphic or parametric equalizer, or by built in tone controls in the instrument itself. Recently, digital equalizers have appeared on the market.

If this function were to be done utilizing the techniques presented here, we could obtain all of the advantages of the FIR filter structure, and avoid the disadvantage of greater computational load. The remaining disadvantage, processing delay, will be addressed in the "Limitations and Problems" section.

Calculating the impulse response, or filter coefficients, for a desired frequency response is not a trivial matter, but various techniques are well documented in standard digital signal processing textbooks [1,2] and are implemented in various filter design software packages available on the market.

One trap to avoid falling into is to think that one could implement the filter directly in the frequency domain (and not bothering to calculate the impulse response) by just taking the FFT of the signal, multiplying it by the desired frequency response, and taking the inverse FFT of the result to obtain the filtered signal. The reason why this will not work in general is because of circular convolution. Any arbitrary frequency response that one might apply in this manner will most likely transform (via inverse FFT) into an impulse response that would occupy the full length of the FFT window. In fact, a desired "ideal" frequency response would possibly have an infinitely long impulse response, which would wrap around on itself in a circular buffer manner as well. At any rate, circular convolution would occur, with two nasty side effects: the output would exhibit non-causal characteristics, in the sense that the circular convolution of an input event that occurs late in the FFT time window would cause a response to it to appear at an earlier time in that window; and each boundary point between FFT windows would have a splicing discontinuity, since the first point in a window would be affected by the points at the end of that window, and not by the points at the end of the previous window, at it should be for a proper linear convolution. Therefore, the best way to proceed is to use the existing filter design algorithms to create an impulse response of a certain length (and in the process making the frequency response non-ideal) and then proceed as described with an FFT window of longer length.

ROOM EQUALIZATION

Another application that we may use these techniques for is in the area of room equalization. Traditionally, room equalization for sound reinforcement has been done with either graphic equalizers (to generally "level" the frequency response of the room) and/or parametric equalizers (to notch out particularly troublesome resonances). Using long "filter" lengths, it is possible to almost exactly eliminate the early reflections and resonances, essentially "de-convolving" the early parts of the impulse response of the room. This results in a greater clarity of sound, while preserving the overall later reverberation characteristics. The increased clarity is a result of the delay between the original "direct" sound and the onset of the reverberant field, similar to the use of the "pre-delay" parameter found on some digital reverb units on the market.

The basic problem here is to come up with some finite impulse response that will perform the desired function. There are two ways to come up with a solution. Both ways involve first sampling the impulse response of the room itself. Then, determine what early characteristics are undesirable and window the impulse response to include only this early part of the impulse response. We now have a finite impulse response that we wish to cancel. We must create another finite impulse response that will cancel it, i.e. will create just a single impulse when convolved with the previously windowed impulse response of the room. Actually, this is impossible to do exactly, since this "inverse" impulse response that we are trying to determine will almost always be infinite in length. However, in practical terms, we can almost always determine some finite impulse response that will provide an acceptable attenuation of the undesirable characteristics of the reverberant field.

The first way to create this inverse impulse response is to calculate it by brute force. That is, take the windowed impulse response of the room and try to reduce it to just the original impulse by adding or subtracting time shifted and amplitude scaled copies of itself in an iterative fashion. The coefficients that are determined as the amplitudes (and signs) of the shifted copies become the inverse impulse response. Eventually, the error signal that remains uncorrected gets shifted out timewise and reduced in amplitude to an acceptable degree.

The second way to create the inverse impulse response is to take the Fourier transform of the windowed impulse response, take its inverse, and

perform an inverse Fourier transform to generate the inverse impulse response. In theory this works because multiplying the two transforms would give a transform with a constant value of 1, which corresponds to a single impulse when inverse transformed back to the time domain. In practice, this procedure has one serious pitfall: circular convolution! That's right, we are actually generating an inverse impulse response that works when CIRCULARLY convolved with the windowed impulse response. The only way to get this to work for a linear convolution on a real signal is to minimize the difference between the circular convolution and the linear convolution. To do this, we must first have a windowed impulse response whose inverse, while infinite in length, has some finite length beyond which there is some acceptably minimal amount of energy. The only way to determine this is probably trial and error. Once a length has been chosen, pad the windowed impulse response to that length with zeroes, then perform the transform, inversion, and inverse transform. The result will be a finite length impulse response that consists of the infinite inverse impulse response wrapped around on itself in a circular buffer fashion. Remember, though, that by picking a sufficiently long transform, the "tails" of this impulse response (the portions from distant times that got wrapped around) that cause error in the cancellation have been kept to a minimum level. Now, to perform the fast convolution on an actual signal (our normal operating condition after having determined the inverse response), we will of course have to use a transform length that is even longer still.

LOUDSPEAKER EQUALIZATION

A recent trend in the hi-fi audio industry is to make loudspeakers with increasing amounts of functionality integrated in, such as self-powered speakers or "digital" speakers that have a digital-to-analog converter (DAC), power amp and volume control integrated in. It should now be possible to build self-equalizing speakers as well, in order to provide the best reproduction possible from a given driver. The idea here is to use the same approach as described for "Room Equalization", in which the impulse response of the speaker is inverted for convolution with the actual input signal to be reproduced.

Two basic application approaches present themselves here. The first is to sample the impulse response of the speaker (drivers and enclosure) in an anechoic chamber, compute the inverse, and permanently store it in the

"brain" of the speaker when shipped from the factory, so that the speaker system itself is as good as it can be. The other approach is to let the consumer perform the "calibration" routine by placing a microphone at the preferred listening location, plugging it in to the speaker after installation in the preferred position, and pushing a button on the back of the speaker, which causes it to run its own calibration routine. This way, the combination of the speaker, its position, and the room acoustics is corrected for in the optimal way (again, limiting the length of the cancellation so that it only has to correct for resonances, early reflections, and frequency response variations, and not the entire reverberant field of the room until the sound is completely gone.

Note that this technique only corrects linear imperfections in the speaker (resonances, reflection and diffraction effects, frequency response variations) and not non-linear effects such as harmonic and inter-modulation distortions. Also note that it can only correct for a given listening position (in the room mode) or axis (in the anechoic chamber mode) and can not correct simultaneously for variations in room acoustics or axial response.

AMBIENCE EQUALIZATION

One more application area we will look at, which is the one that requires the longest "filter" length, is that of ambience, or reverberation, synthesis. In this case, the long "filter" impulse response would be the pattern of reflections that make up the reverberant field. In this case, the actual reverberation density, or number of reflections per unit time, could be made as heavy or light as desired. The exact placement of every single reflection could also be selected. Certain extra tricks could be played, such as creating binaural pairs of impulses that would impart a more natural directionality to each reflection, rather than just producing "stereo" reverb, with reflections occurring randomly on either the left or right side. It would even be possible to "sample" the reverberant field of an actual space for use. In the extreme, a binaural sample could be taken from the "best seat in the house" for each of many stage positions. Each instrument in a multi-track recording could be processed with a different reverb sample, corresponding to its desired stage position, resulting in an extremely realistic aural image.

To really live up to the name "ambience equalization", and not just ambience generation, we could add the concept of "room equalization" and come up with an impulse response that would first de-convolve a room's own lousy reverberant field, and then generate the desired ambient field. To do this, we could either first generate the room-equalizing de-convolving impulse response by either of the two methods previously described, and then convolve it with the desired ambient response; or we could directly generate it by the brute force method previously described, where we would shift and scale the room response in an iterative fashion to create the closest approximation to the desired response, rather than the closest approximation to a dry impulse as before.

REAL-TIME IMPLEMENTATION

As we saw in a preceeding section, using Fourier transform techniques to perform fast convolutions greatly reduces the computational load compared to performing the convolutions directly in the time domain. The question we wish to address now is: is this lower computational load one that is practical to implement? The answer is very emphatically yes!

Taking the last example given in the comparison of computational loads, suppose we do have an impulse response of length 131,073. At any typical digital audio sampling rate in the range of of 44.1 to 48 or even 50 KHz, this would represent a period of 2.6 to 2.97 seconds, enough for processing a respectable reverb sample.

Assuming a 50 KHz sampling rate, we would have 20 usec to perform the calculations necessary for each output point. In the time domain, each output point requires 131,073 multiplies (and adds). This would require hardware capable of performing a multiply (and accumulate) in 152 *pico*seconds! This is definitely not practical with today's technology.

However, in the frequency domain, we only require 76 multiplies per output point for this long impulse response. This only requires hardware capable of performing a multiply in 263 nsec. Today's single chip digital signal processing (DSP) microprocessors have typical multiplication cycle times on the order of 100 nsec or less. This is enough to easily handle the multiplication requirements, but the algorithm does require other processing.

For example, the radix-2 butterfly in the FFT requires 6 ALU operations (adds and subtracts) for each 4 multiply operations. Therefore, we need to find a DSP integrated circuit (IC) that can process FFTs efficiently, i.e. use its multiplier most of the time, not letting it sit idle while the IC is busy doing other things.

One such family of DSP microprocessors is the Zoran family of Vector Signal Processors. With a clock rate of 25 MHz, the multiplication throughput time is 80 nsec. Other parts of its architecture make it ideally suited for processing FFTs and vector multiplies. There is a dual ALU architecture embedded in the execution unit that allows the radix-2 butterfly to be performed in only 4 clock cycles, rather than 6. There is a separate bus interface unit that can perform bit-reversed addressing, and dual internal RAM sections that allow data I/O to be performed concurrently with FFT execution. Another feature of the architecture is that it only uses one external bus, for both instructions and data, and only uses it a fraction of the time while performing FFTs at full speed. This allows multiple processors to share a common bus and still run in parallel without fighting for bus time. In fact, FFT and vector multiply operations are both single instructions in the instruction set!

As an example, the ZR34325, a 32-bit floating point Vector Signal Processor, will perform a 1K complex point FFT (including all I/O and bit-reversed addressing for data re-ordering) in 1.732 msec, or about 21650 internal clock cycles (at 80 nsec). This FFT operation itself requires: (10 passes) X (512 butterflies/pass) X (4 multiplies/butterfly) = 20480 multiplies. This means that the '325 achieves almost 95% utilization of the multiplier bandwidth while performing an FFT. This results in an effective multiplier throughput rate that is still under 100 nsec, which is more than twice as fast as necessary for the example mentioned above.

The examples shown in Figures 7, 8, and 9 were performed on the ZR34161, a 16 bit block floating point/integer member of the Zoran Vector Processor family. Figure 7 shows a short "bump" (representative of a percussive sound) being convolved with a pattern of randomly scattered and weighted impulses (representative of an early reflection pattern in a reverb application). This example allows one to see very clearly the effect of the convolution in creating the many duplicate copies of the input signal. Figure 8 shows a rectangular pulse being convolved with a shorter triangular impulse response so that the turn-on and turn-off transients may be clearly seen. Figure 9 shows the same thing as Figure 8, except the rectangular pulse input occupies almost the entire length of the FFT. This allows us to see the results of

circular convolution very clearly, since the turn-on and turn-off transients now overlap each other at the beginning of the output buffer area. All three examples use FFTs of length 4096; only the real parts of the complex signals are plotted (except Figure 9b, where the zero imaginary part is also plotted to provide a reference baseline at 0, since the real signal never gets down that low, for easy comparison to Figure 8c), both the real and imaginary parts of the FFTs in Figure 7 are plotted.

LIMITATIONS AND PROBLEMS

As with any technique that appears to do great and wonderful things on the surface (and reducing computational requirements by a factor of over 1700 should certainly qualify as wonderful!), there are certain disadvantages involved here.

The first disadvantage is that of processing pipeline delay for performing the FFT based convolution. In the time domain, we can have immediate results--an output point is calculated from the current and N - 1 most recent input samples. However, in the frequency domain, since we are calculating blocks of output points by segmenting and overlapping the input data, we introduce delay. In our 100-tap filter example, we are taking input segments every 413 points. If we look at the first point in one of those segments, we see that we can't start the processing for it until 412 more input points have come in. Then, there is more delay while we do the processing--yes, we do very few multiplies per output point, but we have to process the whole block at once, so we have a lot of multiplies to do before we get the results that include that first output point that corresponds to that first input point we were talking about.

For certain specific applications, this delay would be objectionable, for instance in sound reinforcement for live music, if the impulse response used is long enough to create a noticeable delay. In other cases, a delay would not be a significant problem. Examples would include: a self-equalizing speaker for home use with a recorded or broadcast source, in which there is no live action to refer to; or any sound reinforcement application using a sufficiently short impulse response, and therefore negligible delay.

Another interesting application area that would not find the delay objectionable is the digital audio workstation. As an example, suppose you

were experimenting with various reverbs on a track, or even doing "ambience equalization" to fix a recording with bad source room acoustics. As long as the processing was done at a rate that is at least as fast as the playback rate, the pipeline delay would only add a few seconds to the "rewind time" between playbacks with different settings. Since the processing is done in digital scratchpad memory, any other tracks could easily be synced with the delayed processed track. The reverbed track could be stored in another area of scratchpad memory, so that subsequent playbacks of the same version would have no extra delay. Compare this with other reverb approaches: typical digital reverb techniques use several recirculating delays with cross-feedback to create many reflections, but while this is also easily done in real time with relatively cost-effective hardware, the reflection patterns are still synthetic and not real samples of actual spaces; time-domain convolution of a real reverb characteristic could also be done in software on whatever microprocessor happens to be in the workstation, but it would be excruciatingly slow when compiling a track with a new reverb characteristic of any appreciable length.

In those applications where the delay is a problem, there is one obvious solution that has its own other problem. If we simply reduce the number of input and output points that we are processing in one chunk, we do reduce the delay that comes from waiting for a whole lot more points. The problem is, we still have to do the same amount of processing (using an FFT that is still bigger than the length of the impulse response) for that smaller number of points. Therefore, the computational advantage of fewer multiplies per point is lessened.

In fact, it can be shown that the greatest computational advantage for a given FFT size occurs when the number of points processed and the number of filter taps are both approximately half the FFT size. For example, suppose we are using 256 point FFTs and we convolve 128 input points with 129 taps. The equivalent number of multiplies in the time domain is 128 X 129 = 16512. If instead we convolved 192 input points with 65 taps, the equivalent number of time domain multiplies (using the same number of actual multiplies to do the frequency domain processing) is only 192 X 65 = 12480. Similarly, increasing the number of taps for the same length FFT reduces the number of points that can be processed, also reducing the computational efficiency advantage.

The converse is not true, however. For a given impulse response length, we get greater computational advantage as the FFT size gets bigger. To continue the example, 2 FFTs and a complex vector multiply of length 128 take

4096 multiplies; with length 256 they take 9216 multiplies (an increase of 2.25X). However, with a 65 tap filter, we can either process 128 input points (64 in each half, real and imaginary) or 384 input points (192 in each half). This is an increase of 3X the number of points. Therefore, the number of multiplies per point goes down from 32 to 24! The problem with this observation is that increasing the FFT length increases the pipeline delay problem, as well as the noise problem, which will be discussed shortly.

The other way to reduce the pipeline delay is to segment the impulse response. In this method, we would take a short input segment and separately convolve it with each of the segments of the impulse response. The first result, or "current" segment, i.e. the result of the convolution with the first segment of the impulse response, would get played immediately; the remaining results ("future segments") would be saved for future use. While playing the "current" segment, we would also have to add to it the previously saved future output segments (which came from older input segments and later impulse response segments) that correspond to the same time slot. This reduces the pipeline delay by as much as desired, but does have the drawback of sacrificing some of the computational advantage. For instance, if we took our example of the 131,073 (128K + 1) tap filter that needed 76 multiplies per output point, and broke it up into 8 of the 16385 (16K + 1) tap filters at 64 multiplies per point, it would appear that we would need 8 X 64 = 512 multiplies per point (and an additional 7 adds per point). This is a lot more than 76, but still a whole lot less than 131,073.

Actually, that number can be improved substantially be removing some redundant calculation. First of all, when doing 8 convolutions with the same input points, we only need to do the forward FFT on the input segment once. Analyzing the requirements of the 16K convolution, we determine that, of the 64 multiplies per point required, 30 are for the forward FFT, 30 are for the inverse FFT, and 4 are for the multiplication of FFTs. By only doing the forward FFT once, we save 7 X 30 = 210 multiplies per output point.

Secondly, instead of saving the final convolution results of each segment, why not just save the multiplied FFTs? Then, when it comes time to play back a certain segment in time, we would first add together all the multiplied FFTs that correspond to that time segment, and then perform just one inverse FFT per output segment. This is permissible because of the principles of linearity and superposition. We will do more extra adds, since we will be adding complex vectors, not real ones, but we will save an additional 7 X 30 = 210 multiplies per output point, for a total of 512 - 210 - 210 = 92 multiplies per output point. This number could also be derived as

follows: 30 for one forward FFT per segment, 30 for one inverse FFT per segment, and 8 X 4 = 32 for the 8 multiplications of FFTs. 30 + 30 + 32 = 92, as expected. Not that much of a penalty after all!

Another technique to increase throughput is the use of multiple processors in parallel. One nice feature of the Zoran Vector Signal Processor family mentioned earlier is that multiple processors may be put on the same bus without sacrificing performance. Multiple bus architectures may also be used with multiple processors.

The other important practical limitation involved in frequency domain convolution is the same one encountered by any complex digital signal processing task, build-up of excessive computational noise. This is caused by rounding off multiplication results, especially when those rounded off results get passed through more multiplication stages, such as what happens in an FFT--each butterfly pass reuses the rounded off results of the previous pass. This is especially a problem when using 16-bit devices, such as the Zoran ZR34161 which was used to generate the examples shown previously. Figure 10 shows the computational noise that results in the "quiet space" between the bumps in the output of the example in Figure 7. Note that the amplitude scale has been greatly magnified, but the result would still be audible noise. The obvious answer to this problem is to go to a device that has greater precision, such as the new members of the Zoran Vector Signal Processor family: the ZR34322, which has a 32 bit integer/block floating point data format; and the ZR34325, which has a 32 bit single precision (IEEE-754 compatible) floating point data format (much greater dynamic range, but only 24 bits of precision). Determining which of those two data formats is better for this application is beyond the scope of this paper, but either one should yield a dramatic improvement in the computational noise performance, compared to a 16 bit integer machine.

SUMMARY

In summary, it has become feasible to attack an old set of problems (those requiring equalization in some form or other) with an old set of theoretical magic (discrete-time frequency domain transform theory), using a semi-old trick (Fast Fourier Transform algorithms) and new hardware (integrated circuits that can perform this stuff in real time).

In fact, the scope of the "problems requiring equalization" that can be attacked in this manner has been expanded by the efficiency of this approach. Specifically, the problems that are more often thought of as time domain problems, attacked with delay line solutions (like reverb generation), can now be solved in the frequency domain using a "filter" approach.

In this paper, we have explored the theory behind this approach, determined its computational advantages, shown how it can be used in a variety of applications using today's technology, and explored some of the problems that arise, with proposed solutions.

REFERENCES

[1] L.R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[2] Alan V. Oppenheim and Ronald W. Schafer, Digital Signal Processing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[3] Zoran; Digital Signal Processors Data Book, Zoran Corp. 3450 Central Expressway, Santa Clara, CA, 1987.

[4] Zoran; ZR34325 Vector Signal Processor Product Description, Zoran Corp. 3450 Central Expressway, Santa Clara, CA, 1988.

[5] Zoran; ZR34322 Vector Signal Processor Product Description, Zoran Corp. 3450 Central Expressway, Santa Clara, CA, 1988.

[6] Zoran Technical Note; Fast Convolution, TN92045, Zoran Corp. 3450 Central Expressway, Santa Clara, CA, 1987.

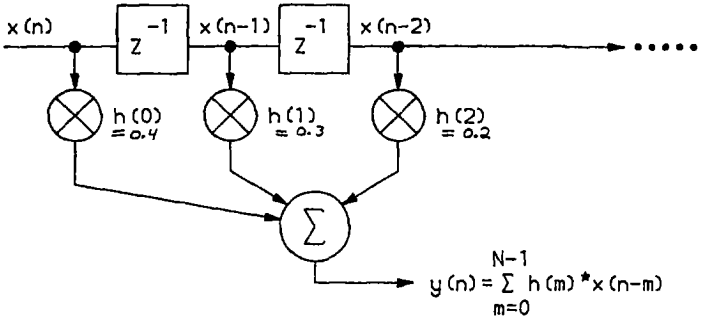


Figure 1. Typical FIR filter structure (3 taps).

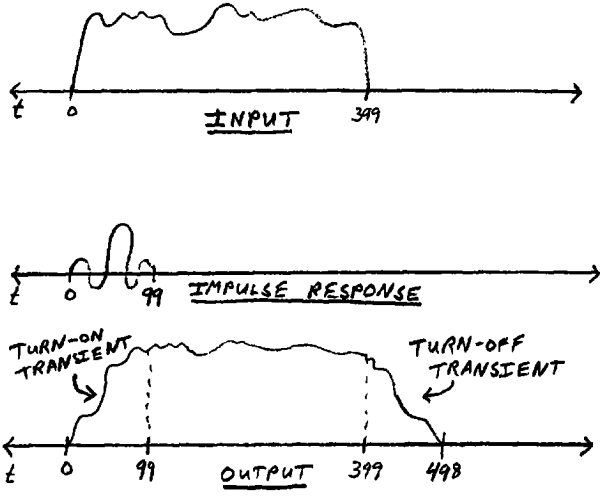


Figure 2. Linear Convolution, 400 Point Input, 100 Point Impulse Response.

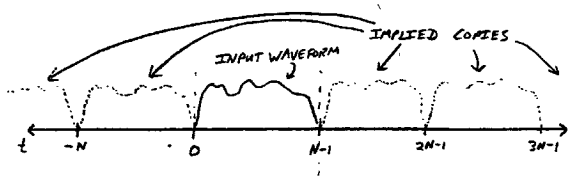


Figure 3a. Implied Periodic Waveform for Fourier Analysis

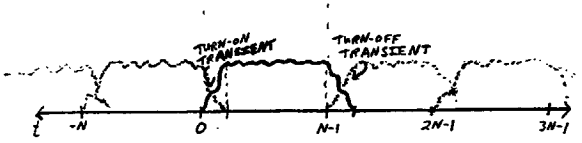


Figure 3b. Linear Convolution On Implied Periodic Waveform

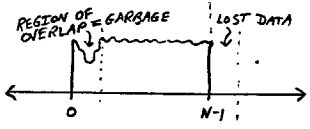


Figure 3c. Resulting Waveform

Figure 3. Circular Convolution

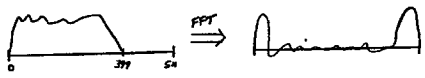


Figure 4a. Input Sequence Padded to 512 Points, and Resulting Spectrum.

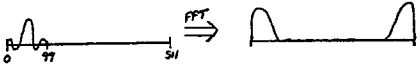


Figure 4b. Impulse Response Padded to 512 Points, and Resulting Spectrum.

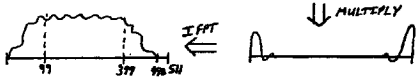


Figure 4c. Output Sequence Derived From Multiplying Above Spectra.

Figure 4. Linear Convolution Using FFTs On Padded Sequences

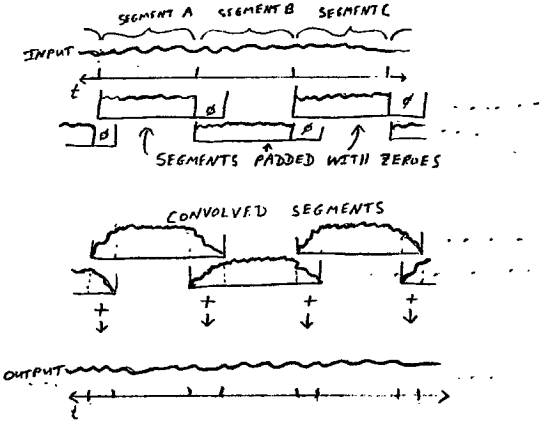


Figure 5. Overlap-And-Add Method for Segmenting Long Sequences

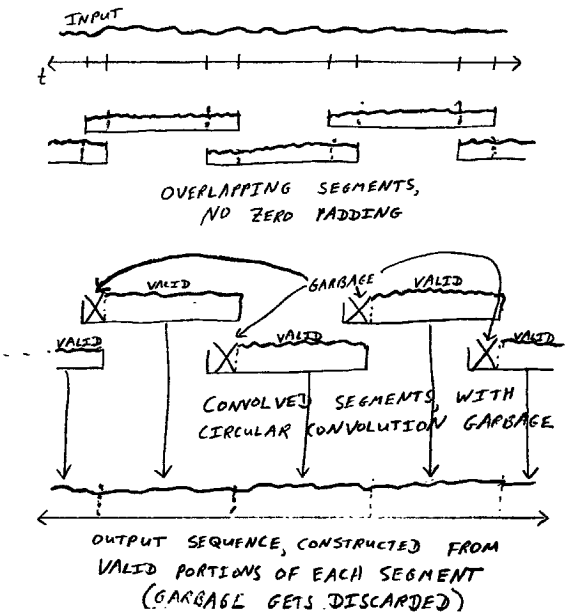


Figure 6. Overlap-And-Discard Method for Segmenting Long Sequences

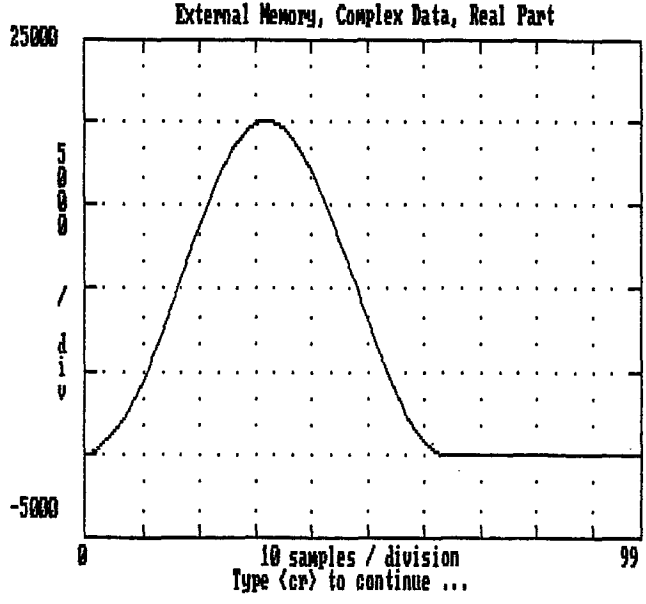


Figure 7a. 64 Point "Bump" Input (Time-scaled)

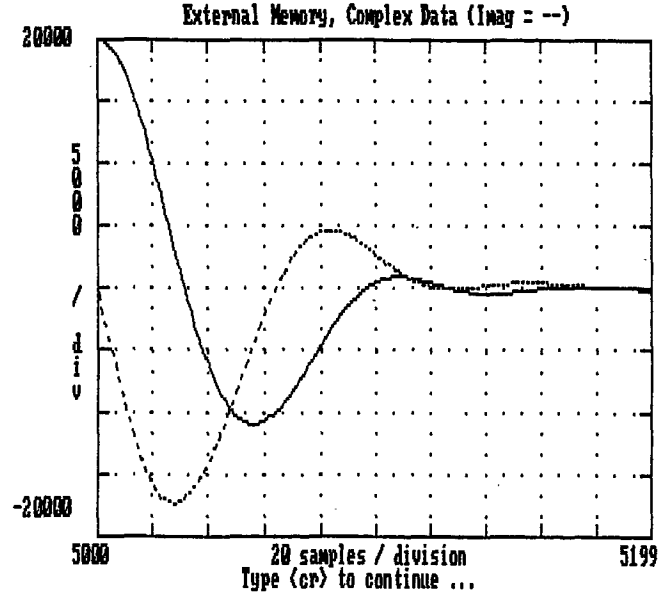


Figure 7b. Partial Display of Spectrum of Bump

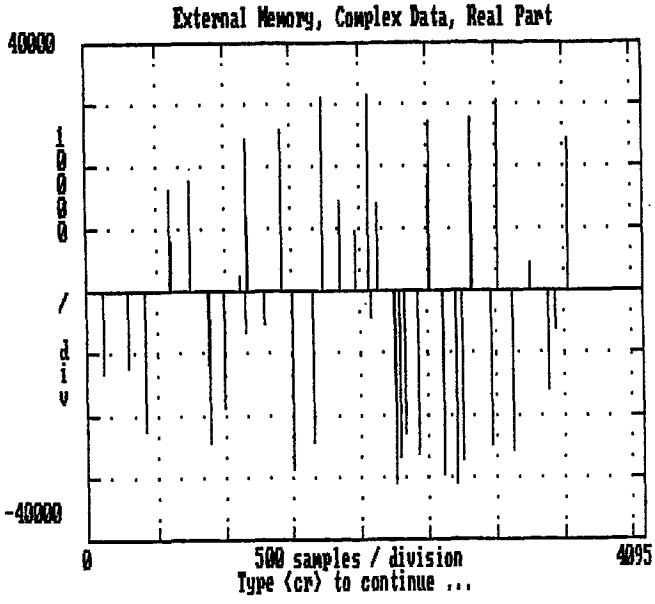


Figure 7c. Early Reflection Pattern

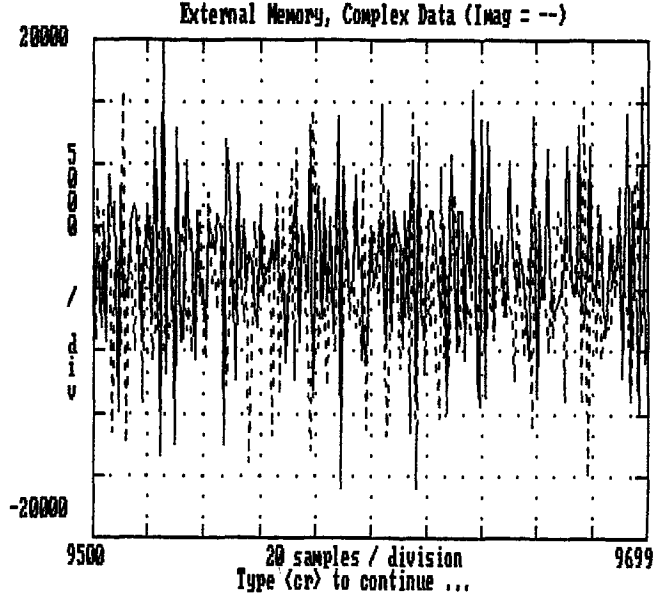


Figure 7d. Partial Display of Spectrum of Reflection Pattern

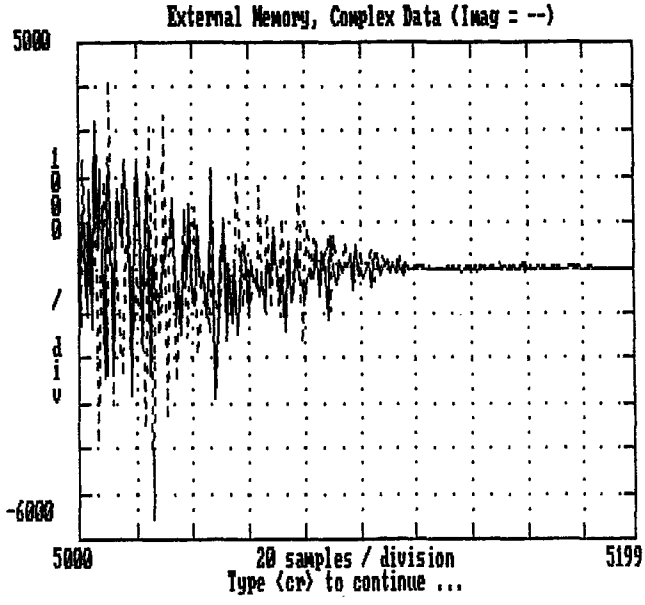


Figure 7e. Partial Display of Multiplied FFTs

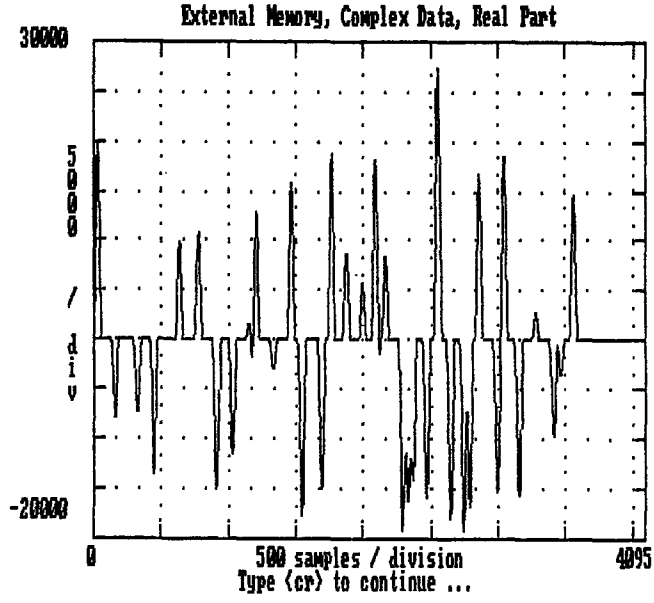


Figure 7f. Result of Convolution

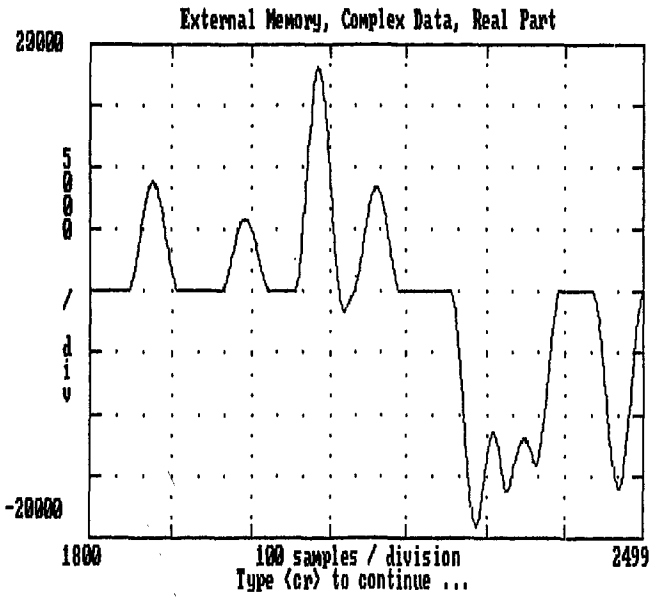


Figure 7g. Close-up Display of Portion of Result (Time-scaled)

Figure 7. Fast Convolution of "Bump" Signal with Reflection Pattern

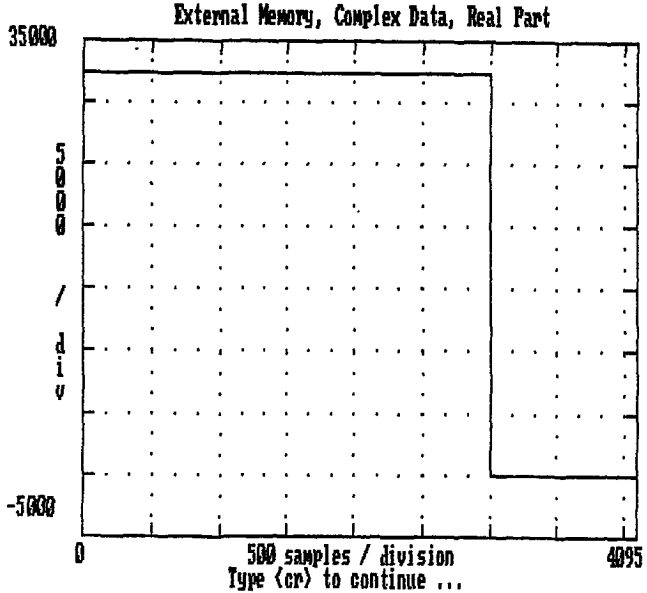


Figure 8a. Rectangular Pulse

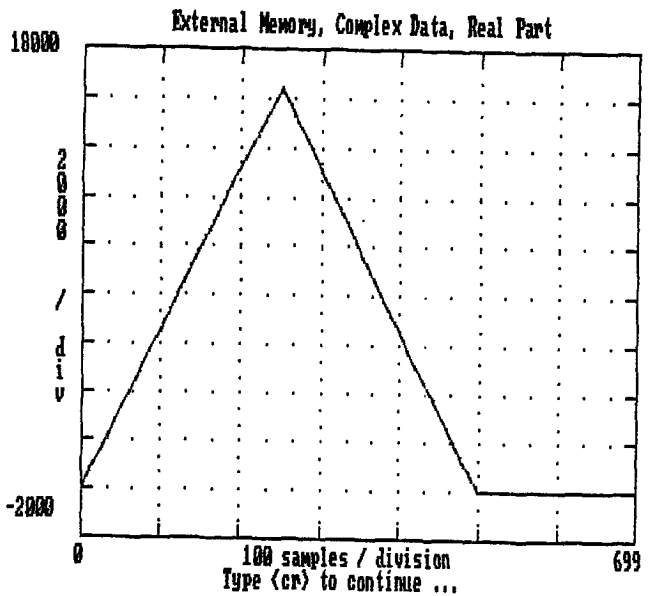


Figure 8b. Triangular Impulse Response (Time-scaled)

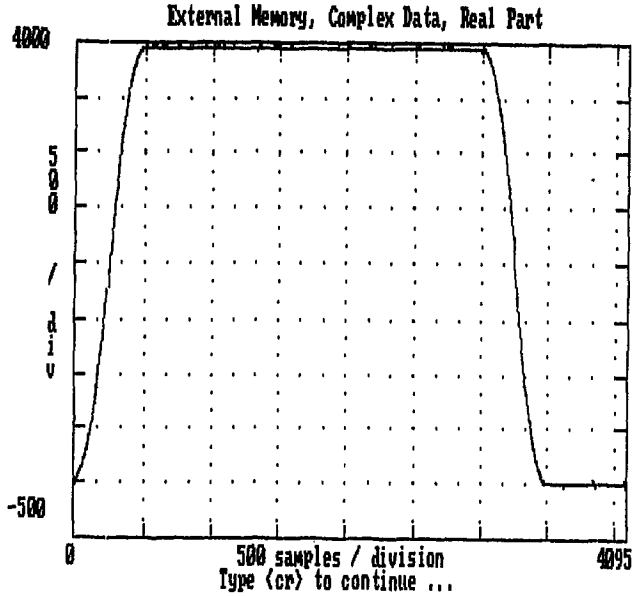


Figure 8c. Fast Convolution Result

Figure 8. Fast Convolution of Rectangular Pulse with Triangular Pulse

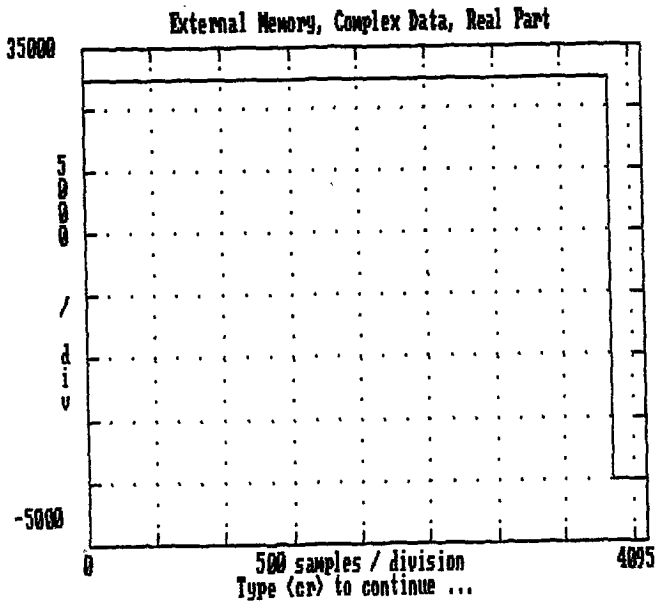


Figure 9a. Longer Rectangular Pulse

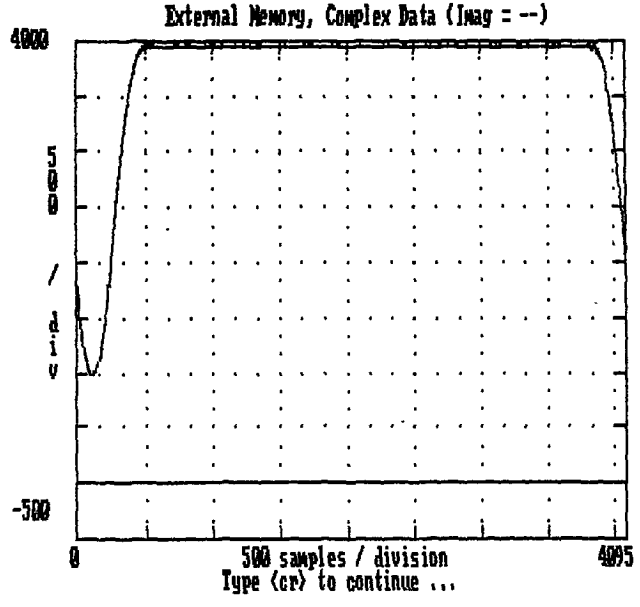


Figure 9b. Result of Circular Convolution with Triangular Pulse in Figure 8b.

Figure 9. Circular Convolution Example

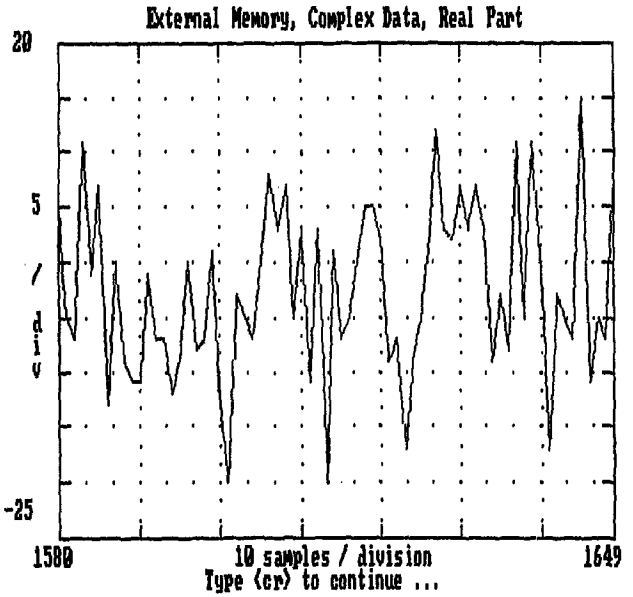


Figure 10. Computational Noise Resulting Between Bumps in Figure 7f.