UNIVERSITY OF MIAMI

REDUCING ARTIFICIAL REVERBERATION ALGORITHM REQUIREMENTS USING TIME-VARIANT FEEDBACK DELAY NETWORKS

By

Jasmin Frenette

A Research Project

Submitted to the Faculty of the University of Miami
in partial fulfillment of the requirements for the degree of
Master of Science in Music Engineering Technology

Coral Gables, Florida

December 2000

UNIVERSITY OF MIAMI

A research project submitted in partial fulfillment
of the requirements for the degree of Master of Science
in Music Engineering Technology

REDUCING ARTIFICIAL REVERBERATION ALGORITHM REQUIREMENTS USING TIME-VARIANT FEEDBACK DELAY NETWORKS

Jasmin Frenette

FRENETTE, JASMIN (M.Sc., Music Engineering Technology)
(December 2000)

Reducing artificial reverberation algorithm requirements
using time-variant feedback delay networks

Abstract of a Master's Research Project at the University of Miami.

Research project supervised by Professor William Pirkle.
No. of pages in text: 122.

Most of the recently published artificial reverberation algorithms rely on a time-invariant feedback delay network (FDN) to generate their late reverberation. To achieve a high-quality reverberation algorithm, the FDN order must be quite high, requiring a good amount of memory storage and processing power. However, in applications where memory and computational resources are limited such as hardware synthesizers or gaming platforms, it is desirable to achieve a good sounding reverberation. This thesis proposes the use of time-variant delay lengths to maintain the quality of the reverberation tail of an FDN, which reduces the algorithm's processing time and memory requirements. Several modulators are evaluated in combination with several interpolation types for fractional delay interpolation. Finally, the computation efficiency and memory usage of the time-variant reverberation algorithms are compared with the equivalent quality, higher order, time-invariant algorithms.

ACKNOWLEGMENT

I would like to thank professor William Pirkle, who guided me throughout this project, giving me invaluable recommendations and offering helpful insights into the audio industry. I am also very grateful to all the music engineering students who offered their time for the listening test. Furthermore, I would also wish to thank Robert Hartman and Amy Powers for proofing the draft version of this thesis. Finally, I would especially like to thank Jacinthe Granger-Piché for her help and endless encouragements in the achievement of this thesis.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLES

1 INTRODUCTION

Artificial reverberation algorithms are used in every commercial studio to add life to dry recordings (recordings that don't have any reverberation). Real reverberation consists of a large number of discrete echoes that would need a large amount of processing power to exactly recreate on a computer. Most artificial reverberation algorithms attempt to model real room reverberation by reproducing only the salient characteristics of those rooms. For example, they can use a model to simulate the individual early echoes of a room, and another model to simulate the late reverberation, perceived as being an exponentially decaying white noise.

This thesis will focus on the late reverberation synthesis. In order to create a density of echoes that approximate the decay of a real reverberation, most artificial algorithms use feedback loops and delay elements. The output of these algorithms (the reverberated signal) is produced by repeating the input signal (the signal to be reverberated) thousands of times per second to produce a density of echoes that is so high that it sounds like white noise. The frequency response of most of these reverberation algorithms contains discrete frequency peaks. When the echoes produced by the delay lines occur at a fixed rate, certain frequencies resonate more than others during the reverberation decay, which does not sound natural. One solution to help minimize this effect is to use more feedback loops. However, this solution requires more memory and computation.

This thesis proposes another alternative to produce a better reverberation. By changing the rate of the echoes in real time, we should be able to vary the location of the peaks in the reverberation frequency response in time. This avoids the build-up of resonances at specific frequency locations. It should also make the resulting reverberation sound much smoother. This solution would be especially attractive to applications that have limited memory access and computation power, such as game platforms or multimedia application. Game platforms typically have 1 or 2 MBytes of Random Access Memory (RAM) for both audio samples and effects such as reverb. Since the gaming market pushes for better sound quality, greater sound diversity, and more special effects, the memory needs are increased and any memory saving is highly desirable.

Chapter 2 will review the properties of room acoustics and introduce basic artificial reverberation algorithms, such as all-pass and comb filters. It will then summarize different approaches that have been used to create artificial reverberation. Chapter 3 will then focus on a specific reverberation algorithm called Feedback Delay Network (FDN) that is used in most of the recent artificial reverberation literature. It will describe how basic comb and all-pass filters can be assembled to form this general network. Chapter 4 will show how modulation can be used in an FDN to enhance its sound, and will review several modulation and interpolation types. It will also review previous attempts to use modulation in reverberation algorithms. Chapter 5 will describe the design of our modulated FDN algorithm, including the choice of modulation and interpolation methods. It will then detail the performance and memory consumption of the design, and present the result of a listening test comparing it to a non-modulated algorithm containing more delay lines.

2 ARTIFICIAL REVERBERATION

Reverberation is a natural acoustical effect. When a sound is emitted in a reverberant room, it is reinforced by a large number of closely spaced echoes. These echoes occur because the emitted sound bounces off the reflecting surfaces of the room. Artificial reverberation algorithms attempt to recreate these echoes using different techniques requiring varying computational requirements.

This chapter is an overview of artificial reverberation algorithms. Section 2.1 will introduce the two main approaches to artificial reverberation design, sections 2.2 through 2.4 will then review the acoustical properties of real rooms, and the remaining sections will finally present several types of artificial reverberation algorithms.

2.1 Physical vs. Perceptual Approach to Artificial Reverberation

Artificial reverberation can be achieved by using two different approaches. The first one, the physical approach, attempts to artificially recreate the exact reverberation of a real room. To achieve this level of detail, a reverberated signal is usually obtained by convolving the impulse response of a room with a dry source signal. The impulse response can be recorded directly from a real room, or it can be obtained from the geometric model of a virtual room. In this latter case, the geometric properties of the room (such as dimensions and wall materials) can be used to compute the coefficients of the impulse response.

Although this approach allows a precise rendering of the reverberation given the source and the listener's position, it is often not flexible enough and/or efficient enough for real-time virtual reality or gaming applications. For example, the time domain convolution of a three-second audio signal with a two-second room impulse response (sampled at 44.1 kHz) would require approximately 12 billion multiplications and 220,500 additions. The equivalent frequency domain convolution would require 220,500 complex multiplications plus the overhead due to the transform and inverse transform operations.

The second approach, called a perceptual approach, tries to generate artificial reverberation algorithms that will be *perceptively indistinguishable* from natural reverberation. The purpose of these algorithms is to reproduce only the salient parts of natural reverberation. This approach is generally much more efficient than the physical approach and ideally, the resulting algorithm could be completely parameterized. This paper focuses on this approach.

2.2 Perceptual Approach

The impulse response of the St-John Lutheran Church (Madison, WI) is shown in Fig. 2.1. As we can see, the first part of the waveform (from 0 to about 150 ms) is composed of discrete peaks, while the later part is more homogenous and decreases almost exponentially. We can model this impulse response by splitting it in three distinct parts, as shown in Fig. 2.2. According to this model, the impulse response consists of the direct signal followed by discrete echoes called *early reflections* (coming from walls, floor and ceiling) and the *late reverberation*.

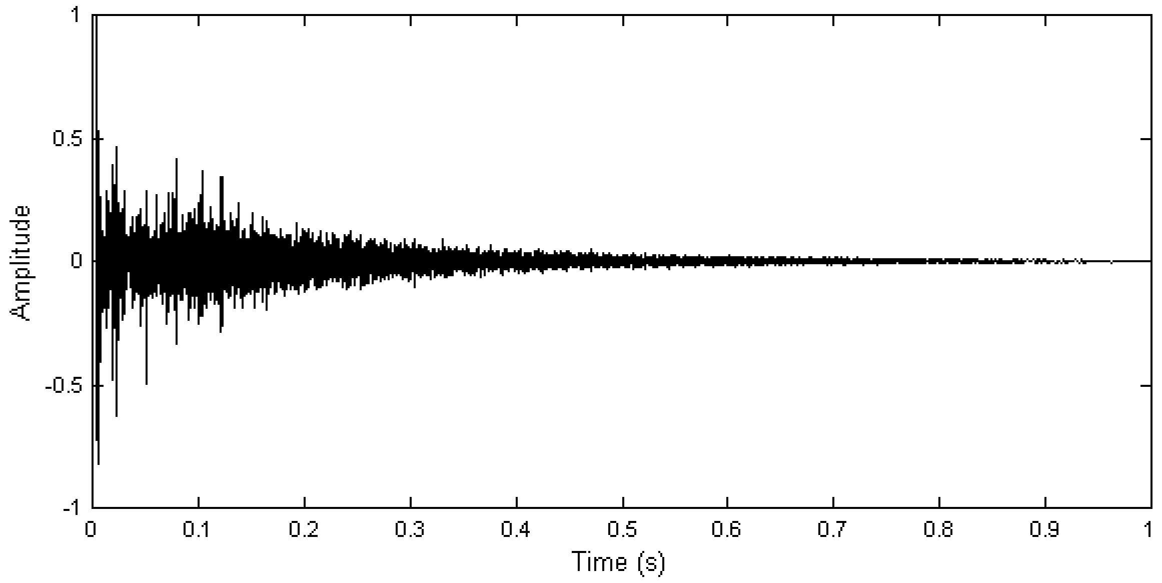


Fig. 2.1. Church impulse response (Sonic Foundry Inc, 1997).

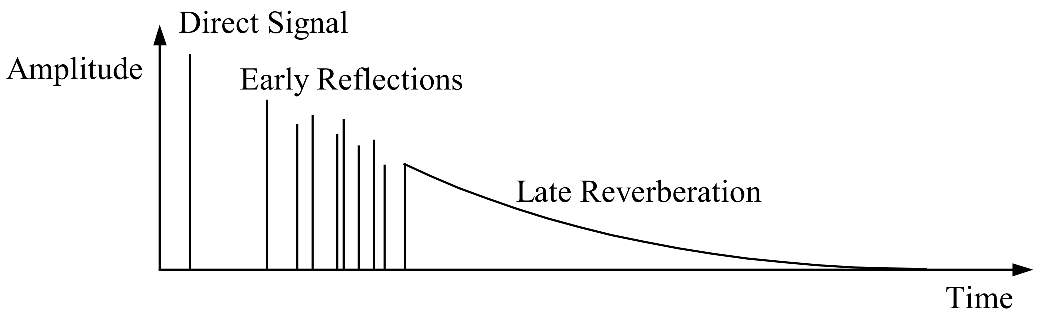


Fig. 2.2. Distinction between the direct signal, early reflections and late reverberation.

This suggests that an artificial reverberation algorithm could be divided in two parts. The first part would produce a finite number of discrete echoes that would coincide with the ones found in the real impulse response, and the second part would generate a high echo density, that would decrease exponentially.

The early reflections are generally computed using a geometric model of the room to be simulated. The most widely used methods are the *source-image* method and the *ray-tracing* method. These can both be combined with head-related transfer functions (HRTF). A discussion of these techniques is beyond the scope of this paper. However a good overview can be found in [10] and [51]. Sections 2.3 and 2.4 will now review the properties of late reverberation in real rooms. Modeling of late reverberation will be discussed in detail in sections 2.5 trough 2.7.

2.3 Reverberation Time and EDR

A room is often characterized by its *reverberation time* (RT), a concept first established by Sabine [33] in his pioneering work on room acoustics in 1900. The reverberation decay time is proportional to the volume of the room and inversely proportional to the amount of sound absorption of the walls, floor and ceiling of the room:

$$T_r = \frac{0.163 \cdot V}{A} \tag{2.1}$$

where T_r is the time (in seconds) for the reverberation to decay 60 dB, V is volume of the room (in m^3), and A is the overall absorption of the room. Since the absorption of a room is frequency dependent, the reverberation time of a room is also frequency dependent. For example, a room containing walls made of porous materials that absorb high frequencies will cause shorter RT as frequency increases.

To measure the RT, Schroeder [40] proposed to integrate the impulse response of the room to get the room's *energy decay curve* (EDC):

$$EDC(t) = \int_t^{\infty} h^2(\tau)d\tau \tag{2.2}$$

where *h(t)* is the impulse response of the room, and can be filtered to obtain the EDC of a particular frequency range.

Jot [17] and Griesinger [14] extended this concept to help visualize the frequency dependent nature of the reverberation. Jot proposed a variation of the EDC that he called the *energy decay relief* or EDR*(t,w)*. The EDR represents the reverberation decay as a function of time and frequency in a 3D plot. To compute it, we divide the impulse response into multiple frequency bands, compute Schroeder's integral for each band, and plot the result as a 3D surface. As an example, the EDR of a typical hall is shown in Fig. 2.3. We can see that the impulse response is decaying slowly at low frequencies. However, the reverberation time at high frequencies is much shorter because the walls absorb the high frequencies more than the low frequencies.

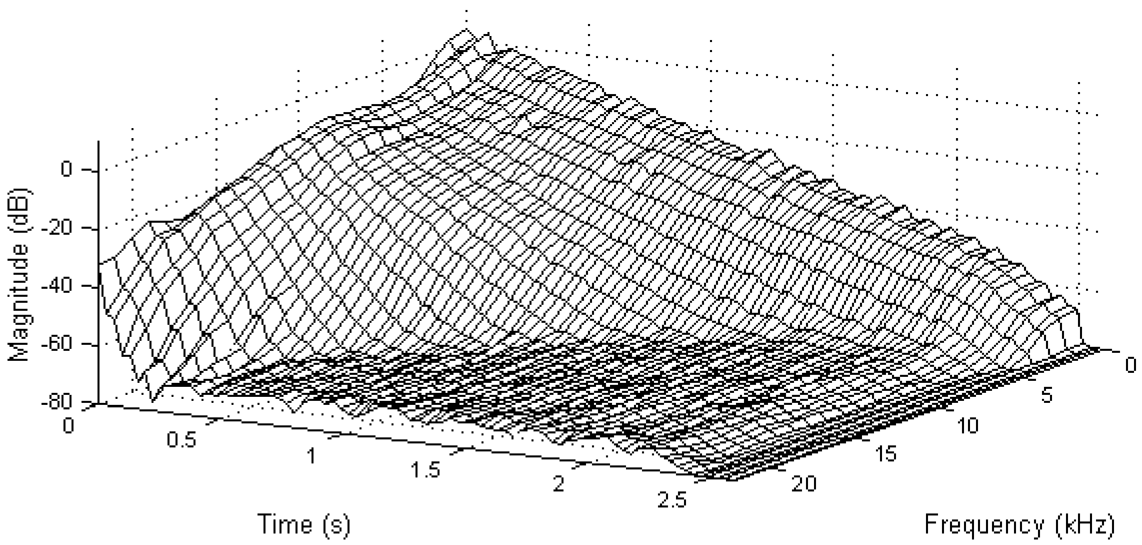


Fig. 2.3. Energy decay relief of a large hall.

2.4 Modal Density and Echo Density

A room can be characterized by its *normal modes* of vibration: the frequencies that are naturally amplified by the room. The number N_f of normal modes below frequency f is nearly independent of the room shape [24]. It is given by:

$$N_f = \frac{4\pi V}{c^3} f^3 + \frac{\pi S}{4c^2} f^2 + \frac{L}{8c} f \qquad (2.3)$$

where V is the volume of the room (in m^3), c is the speed of sound (in m/s), S is the area of all walls (in m^2), and L is the sum of all edge lengths of the room (in m). The modal density, defined as the number of modes per Hertz, is:

$$\frac{N_f}{df} \approx \frac{4\pi V}{c^3} f^2 \tag{2.4}$$

Thus, the modal density of a room grows proportionally to the square of the frequency. According to this formula, a medium sized hall (18,100 m^3 with a RT of 1.8 seconds) has a frequency density of 5800 modes per Hertz at a frequency of 1 kHz.

However, above a critical frequency, the modes start to overlap. Over this specific frequency, the modes are excited simultaneously and interfere with each other. This creates a frequency response that can be modeled statistically [24] [42]. According to this model, the frequency response of a room is characterized by frequency maxima whose mean spacing is:

$$\Delta f_{\max} \approx \frac{4}{T_r} \text{ Hz} \tag{2.5}$$

where T_r is the reverberation time. This statistical model is justified only above a critical frequency:

$$f_c \approx 2000\sqrt{\frac{T_r}{V}} \text{ Hz} \tag{2.6}$$

where T_r is the reverberation time and V is the volume of the room. According to this model, a medium sized hall (18100 m^3, with a RT of 1.8 seconds) would have a frequency response consisting of frequency peaks separated by an average of Δf_{max} = 2.2 Hz above a critical frequency f_c = 20 Hz.

Another major characteristic of a room is the density of echoes in the time domain. The echo density of a room is defined as the number of echoes reaching the listener per second. Kuttruff [24] has shown (using the source-image method with a sphere to model a room) that the echoes increase as the square of the time:

$$N_t = \frac{4\pi(ct)^3}{3V} \tag{2.7}$$

where N_t is the number of echoes, t is the time (in s), ct is the diameter of the sphere (in m), and V is the volume of the room (in m^3). Differentiating with respect to t, we obtain the density of echoes:

$$\frac{dN_t}{dt} = \frac{4\pi c^3}{V} t^2 \tag{2.8}$$

where N_t is the number of echoes that will occur before time t (in s), c is the speed of sound (in m/s), and V is the volume of the room (in m^3).

The time after which the echo response becomes a statistical clutter is dependent on the input signal width. For a pulse of width Δt, the critical time after which the echoes start to overlap is about [38]:

$$t_c = 5 \cdot 10^{-5} \sqrt{\frac{V}{\Delta t}} \tag{2.9}$$

For example, the echoes excited by an impulse of 1 ms in a room of 10,000 m^3 would start overlapping after 150 ms. After this time, we cannot perceive the individual echoes anymore.

Another important characteristic of large rooms is that every adjacent frequency mode is decaying almost at the same rate. In other words, even if the higher frequencies are decaying faster than the lower frequencies, all frequencies in a same region are decaying at the same rate. We should also note that in a good sounding room, there are no "flutter" echoes (periodic echoes caused when the sound moves back and forth between two parallel hard walls).

Now that we have reviewed the properties of reverberation in real rooms, we will focus on the different methods that have been developed to generate artificial reverberation.

2.5 Unit Comb and All-pass filters

Schroeder [38] was the pioneer who first attempted to make digital reverberation while he was working at Bell Laboratories. The first prototype he tried, called a comb filter (illustrated in Fig. 2.4), consisted of a single delay line of *m* samples with a feedback loop containing an attenuation gain *g*.

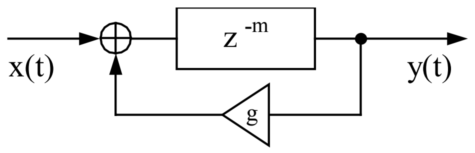


Fig. 2.4. Comb filter flow diagram.

Fig. 2.5 shows a comb filter similar to the one shown in Fig. 2.4. However, in this design, the attenuation gain is in the direct path and not in the feedback path. We will use this configuration during the remainder of this thesis because it will act as the building block of the larger designs. Note that the two designs will produce a similar impulse response, but one will be softer than the other by a factor *g*.

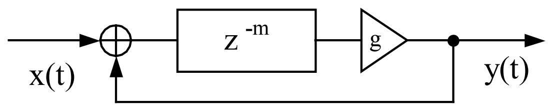


Fig. 2.5. Comb filter flow diagram.

The *z*-transform of the comb filter of Fig. 2.5 is given by:

$$H(z) = \frac{gz^{-m}}{1 - gz^{-m}} \qquad (2.10)$$

where *m* is the delay length in samples and *g* is the attenuation gain. Note that to achieve stability, *g* must be less than unity. For every trip around the feedback loop, the sound is attenuated $20 \cdot \log_{10}(g)$ dB. Thus, the reverberation time (defined by a decay of 60 dB) of the comb filter is given by:

$$T_R = \frac{60}{-20 \cdot \log_{10}(g)} \cdot mT \qquad (2.11)$$

where *g* is the attenuation gain, *m* is the delay length in samples, and *T* is the sampling period.

The properties of the comb filter are shown in Fig. 2.6. We see that the echo amplitude decreases exponentially as time increases. This is good because real rooms have a reverberation tail decaying somewhat exponentially. However, the echo density is really low, causing a “fluttering” sound on transient inputs. Also, the density of echoes does not increase with time as it does in real rooms.

The pole-zero map of the comb filter shows that a delay line of *m* samples creates a total of *m* poles equally spaced inside the unit circle. Half of the poles are located between 0 Hz and the Nyquist frequency $f = f_s/2$ Hz, where f_s is the sampling frequency. That is why the frequency response has *m* distinct frequency peaks giving a “metallic” sound to the reverberation tail. We perceive this sound as being metallic because we only hear the few decaying tones that correspond to the peaks in the frequency response.

Reducing the delay length *m* to increase the echo density will result in a weaker modal density because there will be less peaks in the frequency domain. Thus, increasing the echo density for the aim of producing a richer reverberation will result in a sound that resonates at specific frequencies. The last important thing to note about this filter is that increasing the feedback gain g to get a slower decay (and thus a longer reverberation time) gives even more pronounced peaks in the frequency domain since the frequency variations minima and maxima are:

$$\left|\frac{g}{(1 \pm g)}\right| \qquad (2.12)$$

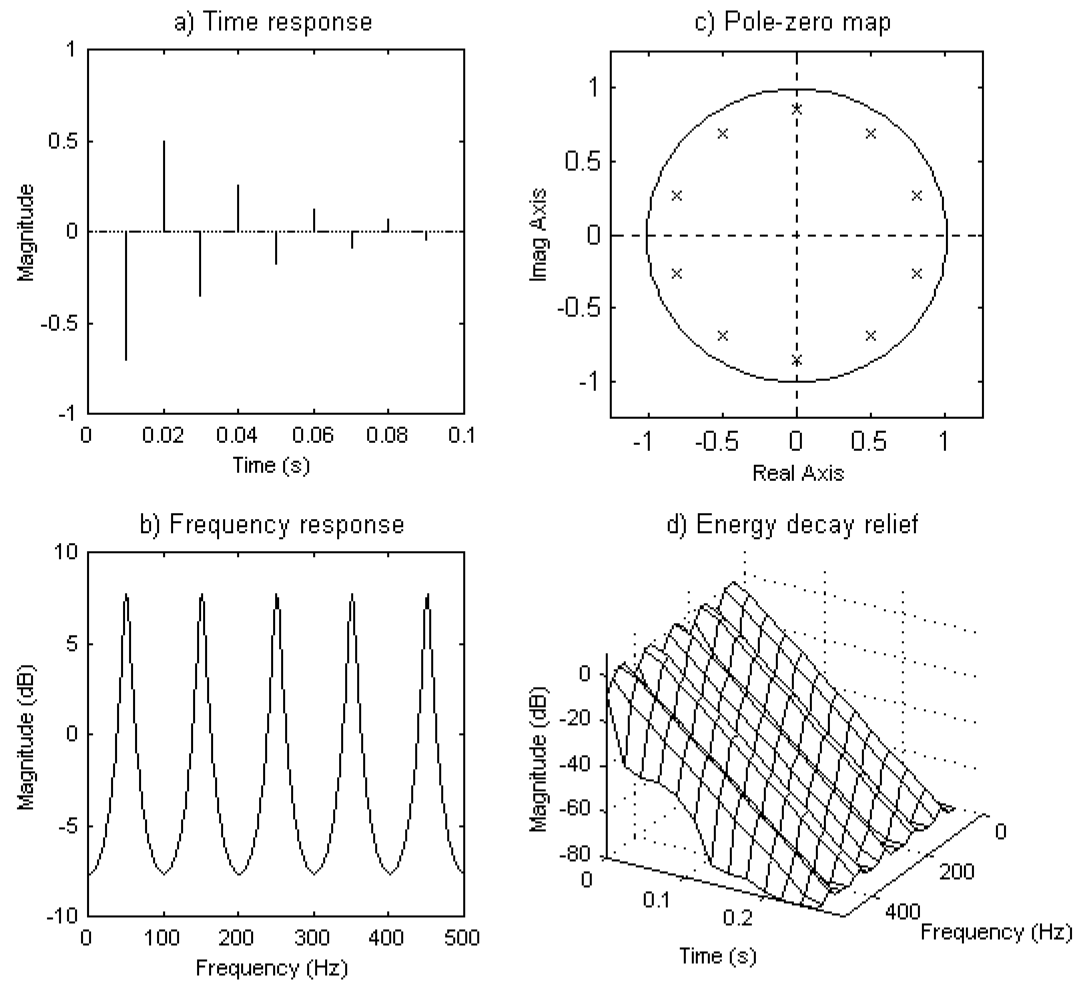


Fig. 2.6. Comb filter (with f_s = 1 kHz, m = 10, and $g = -\frac{1}{\sqrt{2}}$).

To solve the frequency problem of the previous design, Schroeder came up with what he called the all-pass unit shown in Fig. 2.7. Although the original flow diagram of the filter was different then the one shown here, the properties of both filters are equivalent.

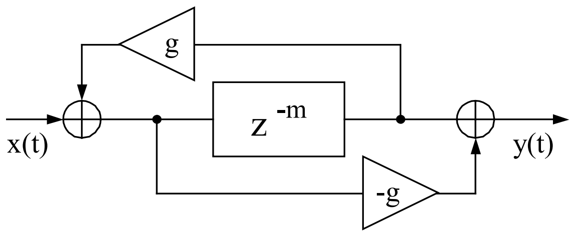


Fig. 2.7. All-pass filter flow diagram.

The *z*-transform of the all-pass filter is given by:

$$H(z) = \frac{z^{-m} - g}{1 - gz^{-m}} \qquad (2.13)$$

The poles of the all-pass filter are thus at the same location as the comb filter, but we now added some zeros at the conjugate reciprocal locations. The frequency response of this design is given by:

$$H(e^{j\omega}) = e^{j\omega m} \frac{1 - ge^{+j\omega m}}{1 - ge^{-j\omega m}} \qquad (2.14)$$

We can see that this frequency response is unity for all ω since $e^{j\omega n}$ has unit magnitude, and the quotient of complex conjugates also has equal magnitude. This leads to:

$$\left|H(e^{j\omega})\right| = 1 \qquad (2.15)$$

The properties of the all-pass filter are shown in Fig. 2.8. As we can see, by using a feed-forward path, Schroeder was able to obtain a reverberation unit that has a flat frequency response. Thus, a steady state signal comes out of the reverberator free of added coloration. However, for transient signals, the frequency density of the unit is not high enough and the comb filter's timbre can still be heard.

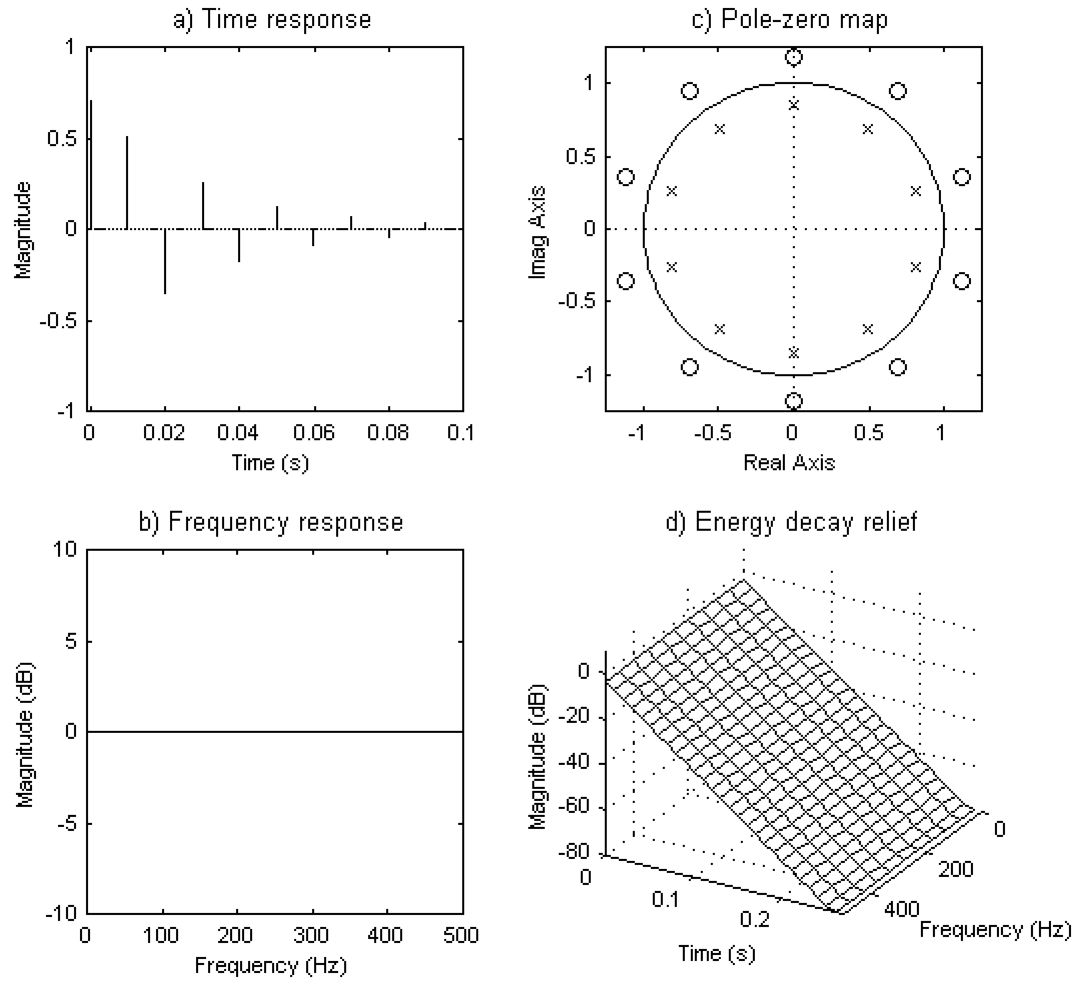


Fig. 2.8. All-pass filter (with f_s = 1 kHz, m = 10, and $g = -\frac{1}{\sqrt{2}}$).

It is interesting to notice that both the comb and the all-pass filters have the same impulse response (except the first pulse) for a gain $|g| = \frac{1}{\sqrt{2}}$. Even with other gains g, we find that both designs sound similar for short transient inputs. It is thus sometimes tricky to look at the frequency response plot of a filter, since its frequency response over a short period of time may be completely different from its overall frequency response.

That is why we often use impulses as an input signal to judge the quality of a reverberation algorithm – it gives us a good indication of the quality of the reverberator both in the time and in the frequency domain.

2.6 Filter Networks

By combining these two unit filters in different ways, we can create more complex structures that will hopefully provide a resulting reverberation with greater time density of echoes and smoother frequency response – even for inputs with sharp transients.

2.6.1 All-Pass Filter Networks

To increase the time density of the artificial reverberation, Schroeder cascaded multiple all-pass filters to achieve a resulting filter that would also have an all-pass frequency response. The result is shown in Fig. 2.9

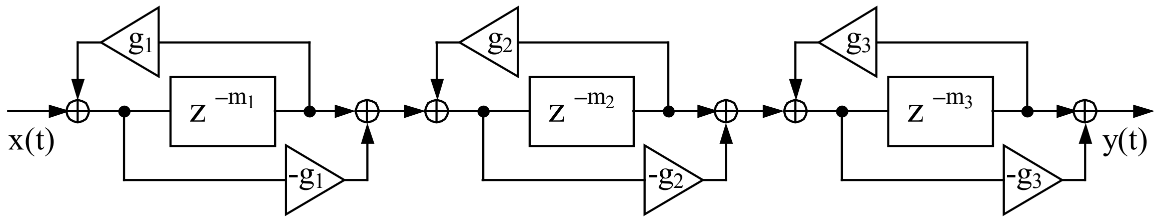


Fig. 2.9. Unit all-pass filters in series.

In this configuration, the echoes provided by the first all-pass unit are used to produce even more echoes at the second stage, and so on for the remaining stages. However, as Moorer [27][1] pointed out, the unnatural coloration of the reverberation still remains for sharp transient inputs.

2.6.2 Comb Filter Networks

Even if the unit comb filter has some frequency peaks, the combination of several unit comb filters in parallel can provide a filter having an overall frequency response that looks almost like a real room's frequency response. In this design, each unit comb filter has a different delay length, to avoid having more than one frequency peak at a given location. An example of parallel comb filter design in shown in Fig. 2.10

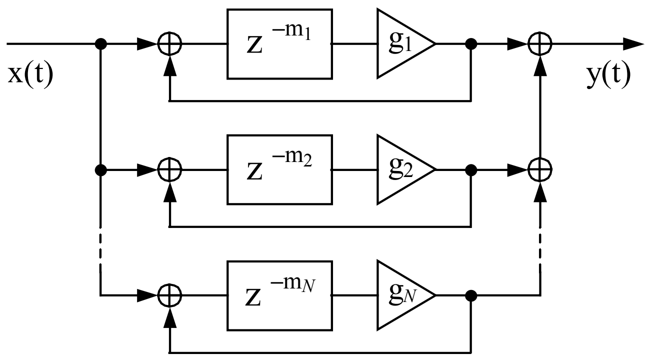


Fig. 2.10. Unit comb filters in parallel.

To achieve good results, each unit comb filter must be properly weighted. Also, the reverberation time of every filter must be the same. In order to do that, we can generalize (2.11) and select the gains g_p according to:

$$g_p = 10^{-3m_p T/T_r} \quad \text{for } p = 1, 2, \ldots, N \qquad (2.16)$$

where N is the number of comb filters.

Schroeder suggested choosing the delay lengths such that the ratio of the largest to the smallest is about 1.5. This parallel comb filter design leads to the *general feedback delay network* (FDN) design, which will be discussed in depth in chapter 3.

2.6.3 Combination of Comb and All-Pass Filter

To achieve a good echo density while minimizing the reverberation coloration, Schroeder put both comb filters (in parallel) and all-pass filters (in series) to give the reverberator shown in Fig. 2.11. He chose delay lengths ranging from 30 to 45 ms for the comb filters and two shorter delay lengths (5 and 1.7 ms) for the two all-pass filters. The attenuation gains of the comb filters were chosen according to (2.16) and the all-pass gains were both set to 0.7. In this design, the comb filters provide the long reverberation delay, and the all-pass filters multiply their echoes to provide a denser reverberation. However, this design sounds artificial because it still does not provide a high enough echo density. Audible resonating frequencies are also present in the reverberation tail.

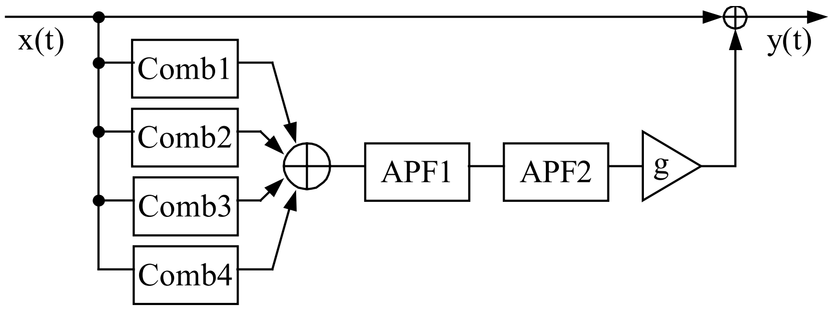


Fig. 2.11. Schroeder's reverberator made of comb and all-pass filters.

Piirilä [29] also suggested using a combination of comb and all-pass filters to produced non-exponentially decaying reverberation. He was able to produce several different reverberation envelopes to achieve interesting musical effects and to enhance reverberated speech.

2.7 Other types of artificial reverberation

This section discusses different types of reverberator designs that have not been implemented in this thesis. However, some of them could easily be combined together with a feedback delay network to produce an even more natural or efficient artificial reverberator.

2.7.1 <u>*Nested All-Pass Designs*</u>

To achieve a more natural-sounding reverberation network, it would be desirable to combine the unit filters to produce a buildup of echoes, as it would occur in real rooms. As suggested by Vercoe [49] and used later by Bill Gardner [5] and William G. Gardner [7], one solution is to use nested all-pass filters. In the diagram below, we substitute the delay line of a unit all-pass filter with a another all-pass filter having a transfer function G(*z*).

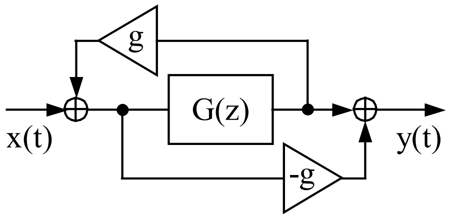


Fig. 2.12. Nested all-pass filters.

If G(*z*) has an all-pass frequency response, then the resulting filter will also have an all-pass response. That enables the use of any depth of nested all-pass filters while insuring the overall stability and frequency response of the system. However, Gardner found that this design still provided a somewhat colored response. He thus suggested the addition of a global feedback path (having a gain $|g| < 1$) to the system. With this new addition, the resulting reverberation is much smoother, probably because of the increased echo density provided by the feedback loop.

2.7.2 Dattoro's Plate Reverberator

Dattorro [3] presented the flow diagram of a plate reverberator (Fig. 2.13) inspired by the work of Griesinger.

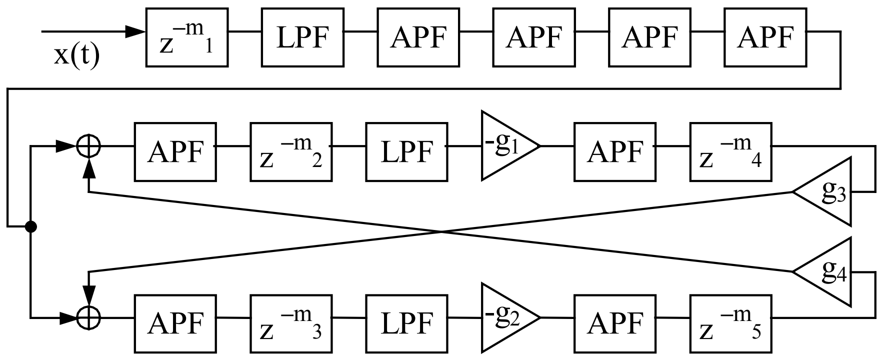


Fig. 2.13. Dattorro's plate reverberator.

The first part of the reverberator consists of a pre-delay unit, a low-pass filter, and four all-pass filters used as diffusers. The diffusers are used to decorrelate the incoming signal quickly, preparing it to enter the second part of the reverberator. This second section, which Dattorro calls the "tank," consists of two different paths that are fed back into each other. Each path is made of two all-pass filters, two delay lines, and a low-pass filter. We create the output of the reverberator by summing together (with different weights) the output taps from the tanks' delay lines and all-pass units. As we will discuss in section 4.3.1, the all-pass filters of the tank can be modulated to smooth the resulting reverberation.

2.7.3 Convolution-based Reverberation

As we mentioned in the beginning of this chapter, creating reverberation by convolving a signal with a real room's impulse response is not flexible or efficient enough for some applications. However, it can be done when high quality reverberation is needed. For example, commercial software such as Sonic Foundry's "Acoustic Modeler" DirectX plug-in lets the user convolve any audio material with real room impulse responses to produce a realistic reverberation. Since time-domain convolution requires a huge amount of processing power, several methods such as block-convolution [9] or multi-rate filtering [48] have been developed to decrease the processing requirement and input-output delay of the process.

It should be noted that convolution is not always done with a real room's impulse response. It is known that convolving the input with an exponentially decaying Gaussian white noise gives good results [26].

Granular reverberation is also a type of convolution-based reverberation. By convolving the input signal with sound "grains" generated by a technique called *asynchronous granular synthesis* (AGS), the virtual "reflections" contributed by each grain smears the input signal in time. The color of the resulting reverberation is determined by the spectrum of the grains, which depends on the grains duration, envelope, and waveform. More details on granular reverberation can be found in [30].

2.7.4 Multiple-stream Reverberation

To create a more natural sounding reverberation, we can also split the reverberation into multiple *streams*, where each stream is tuned to simulate a particular region of a room. Any of the filters mentioned earlier can be used for this purpose, and each filter can be assigned to a different output of the reverberator. They can also be mixed together by the use of other methods such as *head-related transfer function* (HRTF) models, to give each filter a virtual location in the stereo field.

For example, Stautner and Puckette [47] designed a reverberator made of four comb filters (corresponding to left, right, center and back speakers) that were networked together. With this specific design, a signal first heard in the left speaker would then be heard in the front and back speaker and would finally be heard in the right speaker.

2.7.5 Multi-rate Algorithms

Multi-rate algorithms split the signal into different frequency ranges. By combining a bank of all-pass filters with a bank of comb filters such that each comb filter processes a different frequency range, the reverberation time of each specific frequency band can be set by adjusting the corresponding comb filter's feedback gain. In this configuration, it is thus possible to use a different sampling rate for each band according to its bandwidth [50].

2.7.6 Waveguide Reverberation

Digital waveguides are widely used in the physical modeling of musical instruments. Julius O. Smith [45] suggested the use of lossless digital waveguide networks (DWN) to create artificial reverberation. A waveguide is a bi-directional delay line that propagates waves simultaneously in two opposite directions. When a wave reaches a waveguide termination, it is reflected back to the origin. When multiple waveguides are connected together in a closed lossless network, the reflections occurring in a room can be simulated, thus creating a reverberator. Also, with the addition of low-pass filters to the lossless network prototype, frequency dependent reverberation time can be obtained. More details on digital waveguide networks will be given in Chapter 3.

Waveguides can also be connected in a more structured way to form a 2D mesh [35] [36] or even a more complex 3D mesh [34]. A digital waveguide rectangular mesh is an array of digital waveguides arranged along each dimension, interconnected at their intersections. For example, the resulting mesh of a 3D space is a rectangular grid in which each node is connected to its six neighbors by unit delays. One of the problems of this configuration is that the dispersion of the waves is direction dependent. This effect can be reduced by using structures other than the rectangular mesh, such as triangular or tetrahedral meshes, or by using interpolation methods.

3 FDN REVERBERATORS

This chapter will present an analysis of feedback delay networks. Section 3.1 reviews the properties of the unit comb filter introduced in section 2.5. Section 3.2 covers the properties of the parallel comb filter network and explains how to achieve a frequency dependent reverberation time with this design. Section 3.3 derives a general feedback delay network based on the parallel comb filter network.

3.1 Unit Comb Filter Pole Study

The transfer function of the unit comb filter of Fig. 2.5 was given by (2.10) and can be rearranged as:

$$C(z) = \frac{g}{z^m - g} = \frac{1}{m} \cdot \sum_{k=0}^{m-1} \frac{z_k}{z - z_k} \qquad (3.1)$$

In the equation above, the poles z_k (where $0 \leq k \leq m$-1) are defined by $z_k = \gamma \cdot e^{j\omega_k}$, where $\gamma = g^{1/m}$ and $\omega_k = \frac{2k\pi}{m}$. As we already saw in Fig. 2.6c, the poles have equal magnitudes and are located around the unit circle. If the feedback gain is less than unity, this pole pattern corresponds to exponentially decaying sinusoids of equal weight, as illustrated in Fig. 2.6d. The inverse transform of (3.1) gives the following filter impulse response:

$$C(nT) = \frac{1}{m} \cdot \sum_{k=0}^{m-1} {z_k}^n \quad \text{(for n} \geq 0) \qquad (3.2)$$

3.2 Parallel Comb Filter Analysis

When combining N unit comb filters in parallel, the transfer function of the system becomes:

$$C(z)=\sum_{p=0}^{N-1}\frac{g_p}{z^{m_p}-g_p}=\sum_{p=0}^{N-1}\sum_{k=0}^{m_p-1}\left[\frac{1}{m_p}\cdot\frac{z_{k_p}}{z-z_{k_p}}\right] \tag{3.3}$$

By choosing "incommensurate" delay lengths m_p, all the eigenmodes (frequency peaks) of the unit comb filters are distinct (except at $\omega = 0$ and $\omega = \pi$). The total number of resonating frequency peaks is equal to the half the sum of all unit comb filters' delay lengths expressed in samples.

3.2.1 Equal Magnitude of the Poles

As Jot and Chaigne [16] pointed out, if the magnitudes of the poles are not the same, some resonating frequencies will stand out. This is because the decay time of each comb filter will be different. To make every comb filter decay at the same rate, the magnitude of all the poles must be equal. Therefore, we have to follow the rule:

$$\gamma = {g_p}^{1/m_p} \qquad \text{for any } p \tag{3.4}$$

This relates the length m_p of every comb filter p with its feedback gain g_p. Fig. 3.1 shows the smooth decay of a parallel comb filter's impulse response (with equal pole magnitude).

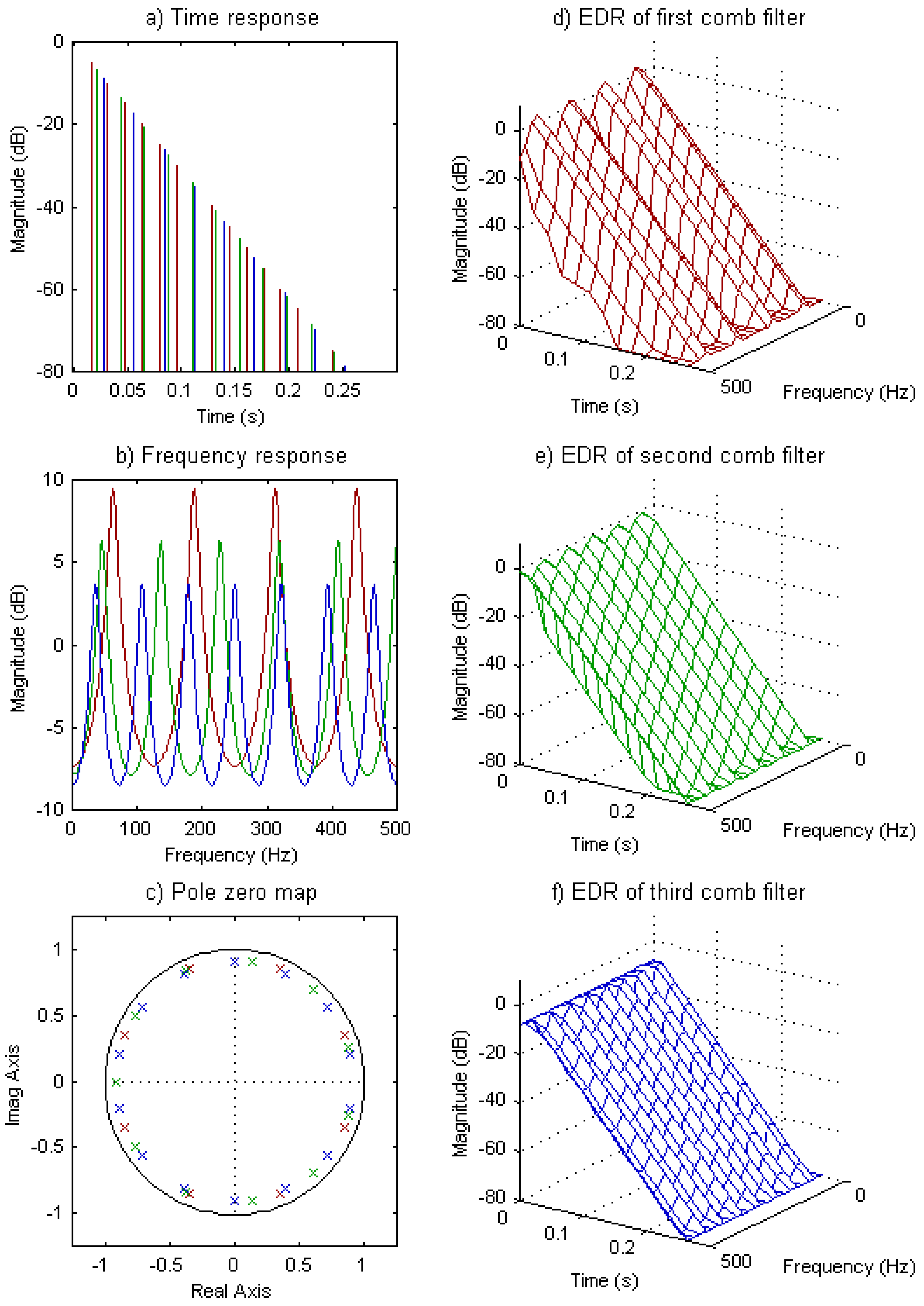


Fig. 3.1. Impulse response of a parallel comb filter with equal pole magnitude (with f_s = 1 kHz, m_1 = 8 in red, m_2 = 11 in green, and m_3 = 14 in blue).

3.2.2 *Frequency Density and Time Density*

Respecting condition (3.4) ensures that all the comb filters will have an equal decay time. However, comb filters that have longer delay lengths m_p will produce frequency peaks (also called *eigenmodes*) with weaker gains, as we can see in (3.3). To reduce this effect, we can try to keep the range from the shorter delay length to the longer delay length as small as possible. That is why Schroeder [39] suggested a maximum spread of 1:1.5 between the comb filter's delay lengths. The other way to get around this problem is to weight the gains of each filter proportionally to their delay lengths before summing their outputs. This ensures that every comb filter's response will be heard equally. The resulting frequency response is shown in Fig. 3.2.

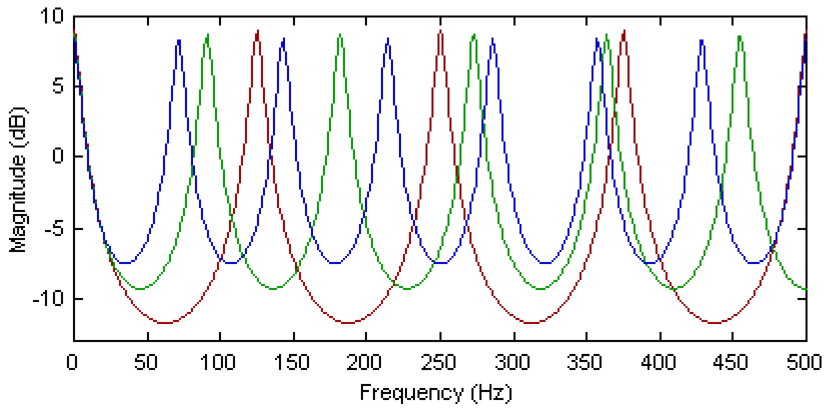


Fig. 3.2. Frequency response of the parallel comb filter of Fig. 3.1 with weighted gains.

The *frequency density* is defined as the number of frequency peaks *perceivable* by the listener per Hertz. A distinction should be made between this perceived *frequency density*, and the theoretical *modal density* discussed in section 2.4. For example, unless we use some weighting to correct the comb filters' outputs as suggested above, the frequency density may be lower than the modal density since the softer frequency peaks may not be perceived.

As discussed in the second chapter, the effect of an insufficient frequency density can be heard with both impulsive and quasi steady-state inputs. With impulsive sounds, the reverberation tail will produce "ringing" of particular modes, or beating of pairs of modes. With inputs that are almost steady state, some frequencies will be boosted while others will be attenuated. To determine the required frequency density for the simulation of a good sounding room, we can refer to the example used in section 2.4. We found that a medium sized hall (18100 m^3, with a RT of 1.8 seconds) would have a maximum frequency peak separation of 2.2 Hz above $f_c = 20$ Hz. This corresponds to a frequency density of 0.45 modes per Hertz.

The *time density* is defined as the number of echoes *perceived* by the listener per second. As with the difference between *frequency density* and *modal density*, we must note the difference between *time density* and *echo density*. The echo density that is present in real rooms is often greater than the perceived time density, because consecutive echoes occurring in a real room usually have completely different gains and louder echoes mask softer ones. In the case of a parallel comb filter design, each echo is heard and the echo density thus equals the time density.

If all the comb filters' delay lengths are spreading within a small range, we can approximate the frequency density and time density by:

$$\text{Frequency density:} \quad D_f = \sum_{p=0}^{P-1} \tau_p \approx P \cdot \tau \qquad (3.5)$$

$$\text{Time density:} \quad D_t = \sum_{p=0}^{P-1} \frac{1}{\tau_p} \approx \frac{P}{\tau} \qquad (3.6)$$

where τ is the average delay length of a comb (in seconds) and N is the number of comb filters. Thus, given a frequency density and a time density, we can find how many combs should be used and what their average delay lengths would be using the following equations:

$$P \approx \sqrt{D_f \cdot D_t} \qquad \text{and} \qquad \tau \approx \sqrt{\frac{D_f}{D_t}} \tag{3.7}$$

For example, to achieve a time density $D_t = 1000$ (as suggested by Schroeder [39]) and a frequency density $D_f = 0.45$, we would need 21 comb filters and an average delay of 21 ms. Griesinger suggested that high-bandwidth reverberation units should have a higher time density, that could even exceed 10,000 echoes per second. This would require more than 67 comb filters! Luckily, feedback delay networks will take care of this issue as we will see in section 3.3.

3.2.3 Reverberation Time

In chapter 2, the reverberation time was defined as the time for the decay to decrease 60 dB. For a parallel comb filter with equal magnitude, all comb units are decaying at the same rate. The following equation gives their decay rate in dB per sample period :

$$\Gamma = \frac{G_p}{m_p} \qquad \text{for any } p \tag{3.8}$$

This equation is equivalent to (3.4) but here, Γ and G_p represent γ and g_p in dB ($\Gamma = 20\log(\gamma)$ and $G_p = 20\log(g_p)$). Using Schroeder's definition of reverberation time and (3.8), we can find the reverberation time for each comb filter:

$$T_r = \frac{-60T}{\Gamma} = \frac{-60\tau_p}{G_p} \tag{3.9}$$

where T_r is the reverberation time, T is the sampling period and $\tau_p = m_p \cdot T$. Modifying either the comb filters' delay lengths or their feedback gains can change their reverberation time. The physical analogy to an increase of the delay lengths would be an increase in the virtual room dimensions. On the other hand, the feedback gain attenuation could be thought of as the amount of absorption occurring during sound propagation in the air. However, we must be careful when changing these gains since they also account for a change in the amount of wall absorption (that should not change when attempting to modify only the room size). Multiplying the delay lengths by α or dividing the feedback gains (in dB) by α would cause the reverberation to be increased by α. It should be noted that in both cases, the time and frequency density of the system would be modified.

3.2.4 Achieving Frequency Dependent Reverberation Time

As suggested by Schroeder [39] and implemented later by Moorer [27], inserting a low-pass filter in the feedback loop of a comb filter introduces some frequency dependent decay rates. This can be used to simulate wall absorption, since a wall tends to absorb more high frequencies than low frequencies.

By replacing the feedback gain g_p with a filter having a transfer function $h_p(z)$, we can achieve a frequency dependent reverberation time. Care must be taken when inserting these absorbent filters to ensure that no unnatural coloration is introduced in the reverberation. We would like every comb filter to decay with the same relative frequency absorption. This can be achieved by respecting a condition that Jot and Chaigne [16] called *continuity of the pole locus* in the *z*-plane:

> […] in any sufficiently narrow frequency band (where the reverberation time can be considered constant) all eigenmodes must have the same

decay time. Equivalently, system poles corresponding to neighboring eigenfrequencies must have the same magnitude.

All comb filters must respect this condition to avoid unnatural reverberation. If the transfer function *$h_p(z)$* is simply a frequency-dependent gain $g_p = | h_p(e^{j\omega}) |$, then the continuity of the pole locus can be ensured by (3.10) as derived from equation (3.4):

$$\gamma(w) = \left| h_p(e^{j\omega}) \right|^{\frac{1}{m_p}} \qquad \text{for any } p \tag{3.10}$$

The reverberation time is also frequency dependent and is given by:

$$T_r(\omega) = \frac{-60T}{\Gamma(\omega)} \qquad 0 \le \omega \le \pi \tag{3.11}$$

Fig. 3.3 shows a parallel comb filter with frequency dependent decay time ($T_r(0) = 3$ seconds and $T_r(\pi) = 0.15$ s) respecting the *continuity of the pole locus* condition.

Finally, we must note that changing the conjugate pole locations of some, or all, of the comb filters in the unit circle will cause some of the eigenmodes to have more energy than others. This changes the general frequency response of the output, as we can see in Fig. 3.3b. However, with the addition of a simple tone correction filter (in series with the parallel comb filter) having a transfer function *$t(e^{j\omega})$* as:

$$\frac{\left| t(e^{j\omega T}) \right|}{\left| t(e^{j\omega_o T}) \right|} = \frac{T_r(\omega_o)}{T_r(\omega)} \tag{3.12}$$

an overall frequency response that is independent of the frequency-specific reverberation time can be achieved, as shown in Fig. 3.4.

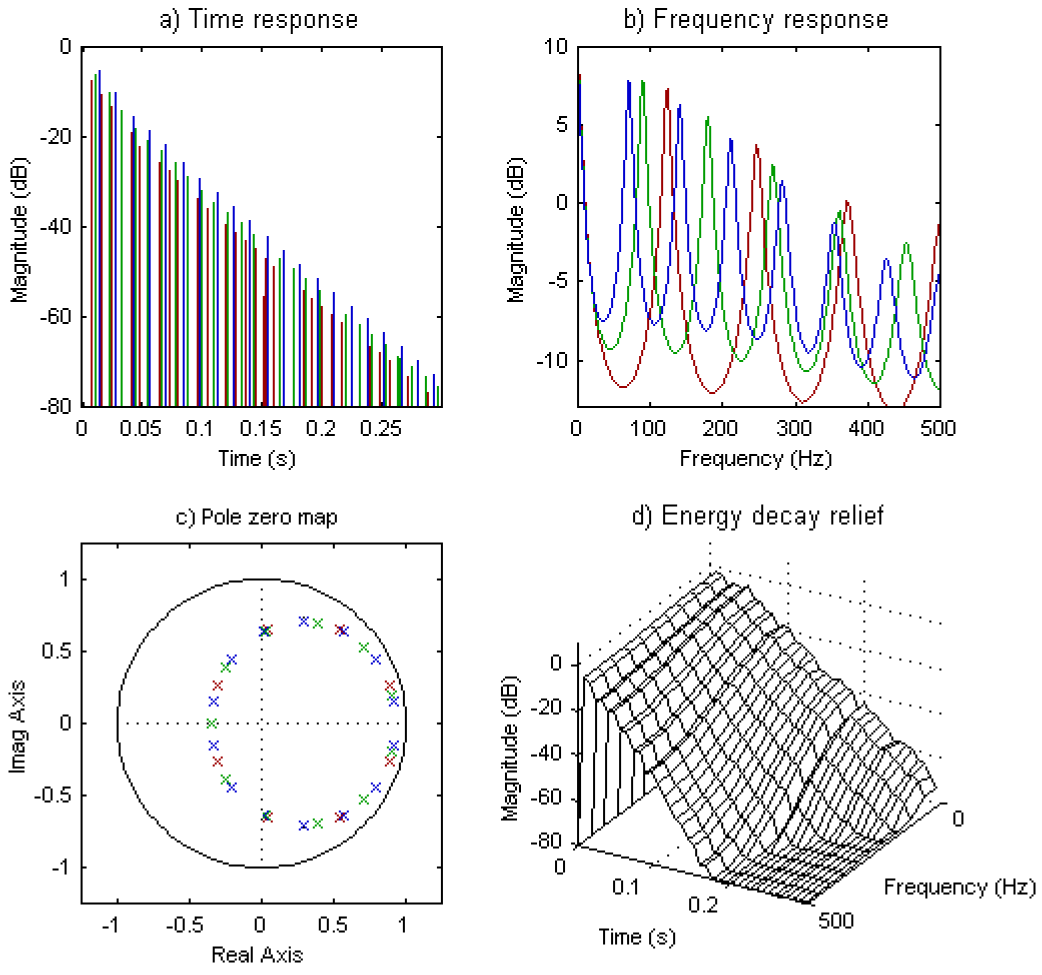


Fig. 3.3. Parallel comb filter with frequency dependent decay time (with $f_s = 1$ kHz, $m_1 = 8$ in red, $m_2 = 11$ in green, and $m_3 = 14$ in blue).

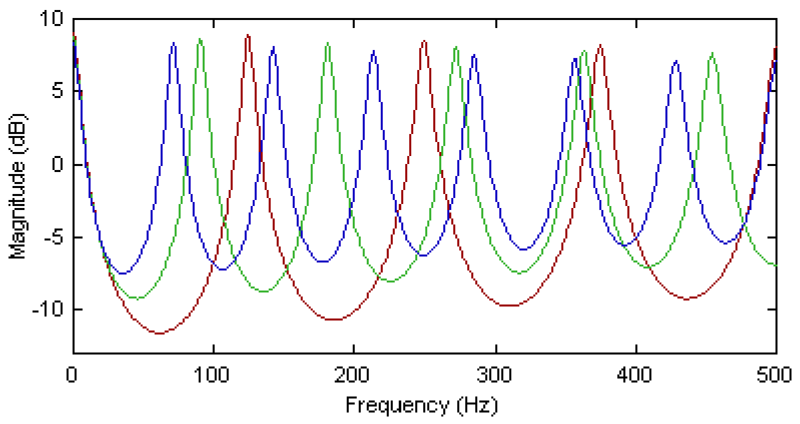


Fig. 3.4. Frequency response of the filter network of Fig. 3.3 with the addition of a tone corrector.

Jot suggested the use of first order Infinite Impulse Response (IIR) filters to handle the absorption, combined with a first order Finite Impulse Response (FIR) filter for tone control. The absorbent filters that will be used in our design have the following transfer function:

$$h_p(z) = k_p \cdot \delta k_p(z) \quad \text{where} \quad \delta k_p(z) = \frac{1 - b_p}{1 - b_p z^{-1}} \tag{3.13}$$

where $b_p = K_p \cdot \frac{\ln(10)}{80} \cdot \left[1 - \frac{1}{\alpha^2}\right]$ and $\alpha = \frac{T_r(\pi)}{T_r(0)}$. The corresponding tone corrector he used was an FIR filter with the following transfer function:

$$t(z) = \frac{1 - \beta \cdot z^{-1}}{1 - \beta} \tag{3.14}$$

where $\beta \approx \frac{1 - \sqrt{\alpha}}{1 + \sqrt{\alpha}}$. The resulting signal flow using both filters is shown in Fig. 3.5.

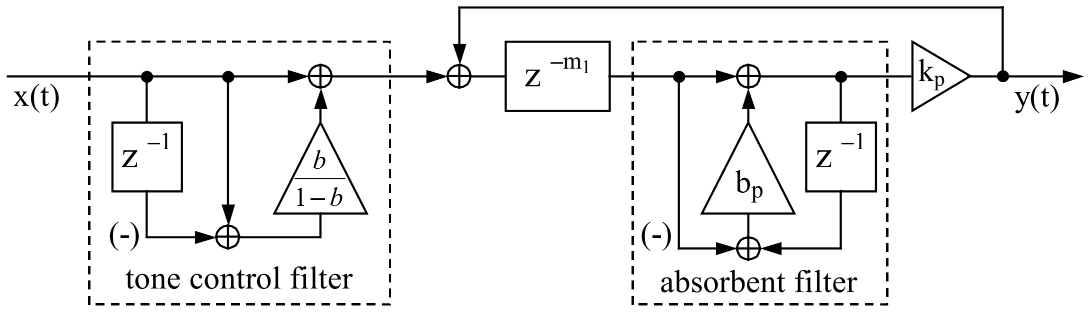
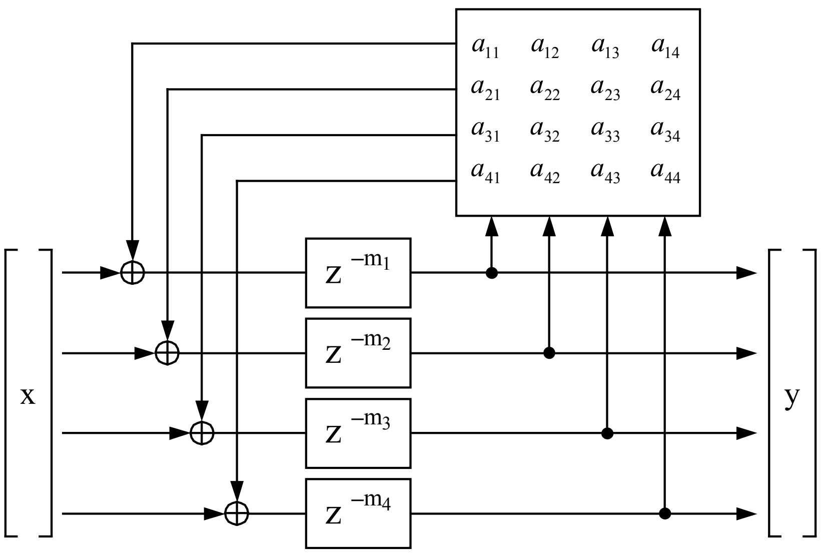


Fig. 3.5. Flow diagram using first order IIR absorbent filters and one first order FIR tone correction filter.

3.3 General Feedback Delay Network

As we saw in the previous section, a considerable number of comb filters are required to achieve a good time density in a parallel comb filter design, given a reasonable frequency density. Some other filter designs achieving a greater time density have been mentioned in the second chapter. Some of those designs, such as the series all-pass filter, produce a build-up of echo density with time as can be observed in real rooms. Ideally, we would like to find a single network that would be general enough to take advantage of the properties of both parallel comb and series all-pass filters. We would also like to achieve the same frequency dependent characteristics we were able to model with the parallel comb filter. This section describes the steps that were taken to achieve this goal.

Gerzon [11] was the first to generalize the notion of a unitary multi-channel network, which is basically an N-dimensional equivalent of the unit all-pass filter. Stautner and Puckette [47] then proposed a general network based on N delay lines and a feedback matrix $\mathbf{A}$, as shown in Fig. 3.6. In this matrix, each coefficient a_{mn} corresponds to the amount of signal coming out of delay line n sent to the input of delay line m. The stability of this system relies on the feedback matrix $\mathbf{A}$. The authors found that stability is ensured if $\mathbf{A}$ is the product of a unitary matrix and a gain coefficient g, where $|g| < 1$. Another property of this system is that the output will be mutually incoherent and thus can be used without additional processing in multi-channel systems.



a_{11} a_{12} a_{13} a_{14}
a_{21} a_{22} a_{23} a_{24}
a_{31} a_{32} a_{33} a_{34}
a_{41} a_{42} a_{43} a_{44}
z^{-m_1}
z^{-m_2}
z^{-m_3}
z^{-m_4}
x
y

Fig. 3.6. Stautner and Puckette's four channel feedback delay network.

3.3.1 Jot's FDN Reverberator

Jot and Chaigne further generalized this design by using the system shown in Fig. 3.7. Using the vector notation and the *z*-transform, the equations corresponding to this system are:

$$y(z) = \mathbf{c}^T s(z) + dx(z) \tag{3.15}$$

$$s(z) = \mathbf{D}(z)\left[\mathbf{A}s(z) + \mathbf{b}x(z)\right] \tag{3.16}$$

where:

$$s(z) = \begin{bmatrix} s_1(z) \\ \vdots \\ s_N(z) \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_N \end{bmatrix} \tag{3.17}$$

$$\mathbf{D}(z) = \begin{bmatrix} z^{-m_1} & & 0 \\ & \ddots & \\ 0 & & z^{-m_N} \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{bmatrix} \tag{3.18}$$

For multiple-input and multiple-output systems, the vectors **b** and **c** becomes matrices. Eliminating $s(z)$ in equation (3.17) and (3.18) gives the following system transfer function:

$$H(z) = \frac{y(z)}{x(z)} = \mathbf{c}^T \left[\mathbf{D}(z^{-1}) - \mathbf{A}\right]^{-1} \mathbf{b} + d \qquad (3.19)$$

The system's zeros are given by:

$$\det\left[\mathbf{A} - \frac{\mathbf{bc}^T}{d} - \mathbf{D}(z^{-1})\right] = 0 \qquad (3.20)$$

Thus, the poles of this system are the solution of the characteristic equation:

$$\det\left[\mathbf{A} - \mathbf{D}(z^{-1})\right] = 0 \qquad (3.21)$$

The analytical solution of the equation is not trivial. However, for specific feedback matrices, this equation is easy to solve.

Jot and Chaigne pointed out that we could represent any combination of unit filters (either comb or all-pass) by using the appropriate matrix. For example, if **A** is a diagonal matrix, the system represents the parallel comb filter described in section 3.2. Triangular matrices have also been studied because in that case, equation (3.21) reduces to:

$$\prod_{i=1}^{N} (a_{pp} - z^{m_p}) = 0 \qquad (3.22)$$

The series all-pass filter is itself a network with a triangular feedback matrix (having diagonal elements equal to the feedback gains g_p). Thus, a series all-pass filter that has the same delay lengths and feedback gains as a parallel comb filter also has the same eigenmodes (resonant frequencies and decay rates) as the parallel comb filter.

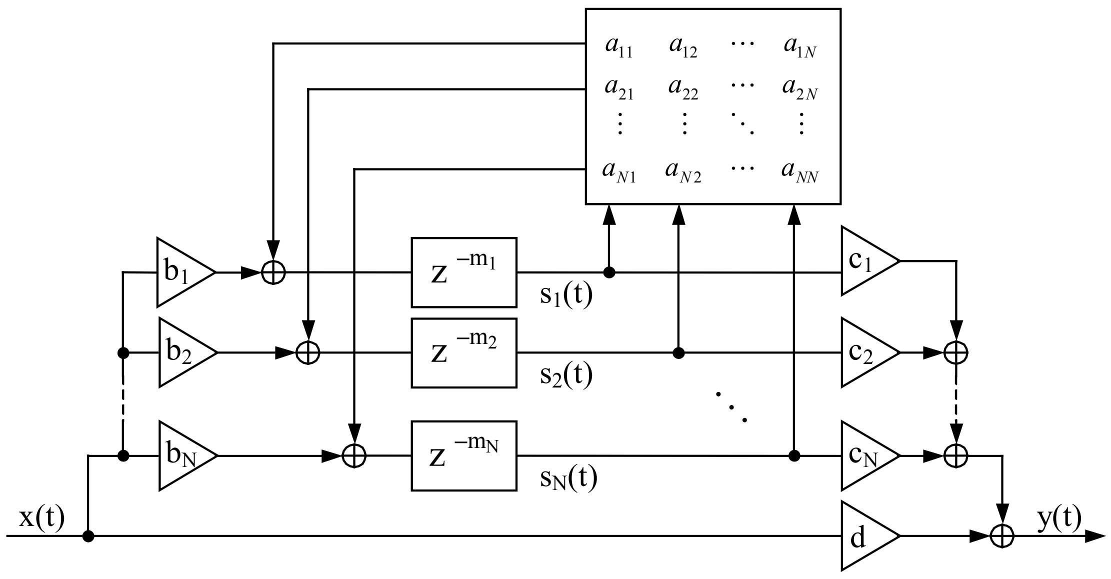

a_{11} a_{12} $\cdots$ a_{1N}
a_{21} a_{22} $\cdots$ a_{2N}
a_{N1} a_{N2} $\cdots$ a_{NN}
b_1
b_2
b_N
z^{-m_1}
z^{-m_2}
z^{-m_N}
$s_1(t)$
$s_2(t)$
$s_N(t)$
c_1
c_2
c_N
d
x(t)
y(t)

Fig. 3.7. Jot's general feedback delay network.

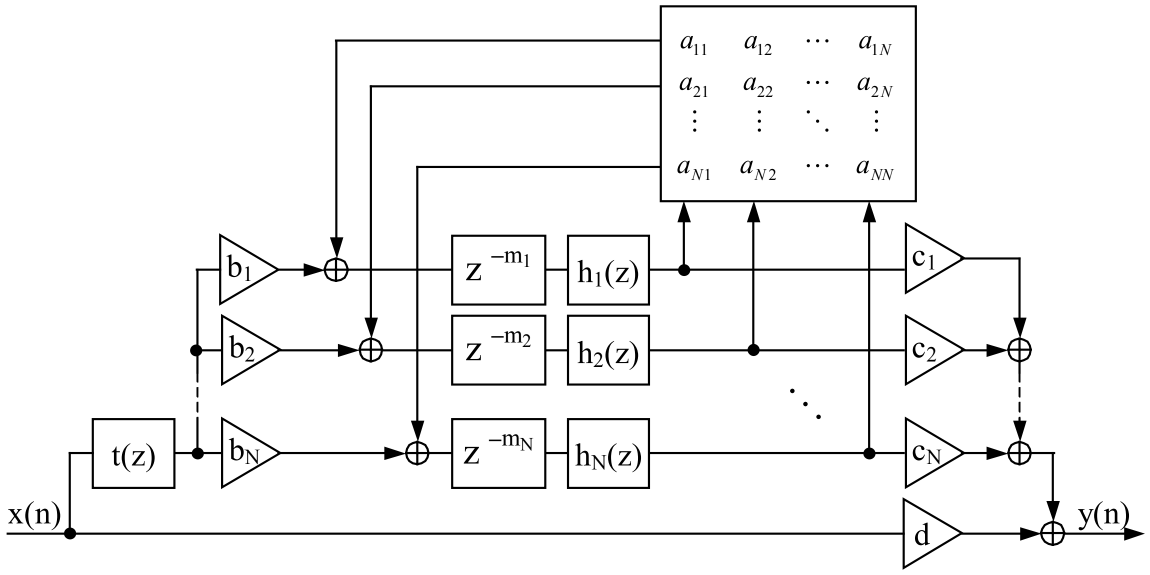
Unitary feedback matrices have been the most used feedback matrix because they ensure the system's stability (because the poles of a unitary feedback loop are all on the unit circle). These matrices preserve the energy of the input signal by endlessly circulating it through the network. Unitary feedback matrices thus always make the FDN prototype *lossless*, although other kind of feedback matrices could also result in a lossless system.

3.3.2 Achieving Frequency Dependent Reverberation Time

To control the decay time of a network, Jot and Chaigne proposed the use of a lossless feedback matrix combined with absorption filters right after each delay line. To make all the comb filters decay at the same rate and to respect the *continuity of the pole locus*, the following equation has to be respected:

$$20\log_{10}\left|h_p(e^{j\omega})\right| = \frac{-60T}{T_r(\omega)} \qquad (3.23)$$

This will move the poles from the unit circle to a curve defined by the reverberation time T_r. Again, this will introduce an overall high frequency attenuation that can be rectified by the use of a general tone correction filter described in section 3.2.4. The resulting FDN including absorption and tone correction filter is shown in Fig. 3.8.



a_{11} a_{12} $\cdots$ a_{1N}
a_{21} a_{22} $\cdots$ a_{2N}
a_{N1} a_{N2} $\cdots$ a_{NN}
b_1
b_2
b_N
z^{-m_1}
z^{-m_2}
z^{-m_N}
$h_1(z)$
$h_2(z)$
$h_N(z)$
c_1
c_2
c_N
t(z)
x(n)
d
y(n)

Fig. 3.8. Jot's general feedback delay network with absorption and tone correction filters.

3.3.3 Selecting a Lossless Prototype

The choice of the feedback matrix is critical to the overall sound and computational requirement of the reverberator. As we discussed in section 3.3.1, many matrices will create a lossless prototype, such as the diagonal matrix equivalent to the parallel comb filter network. For example, Stautner and Puckette [47] used the matrix:

$$\mathbf{A} = g\frac{1}{\sqrt{2}}\begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & -1 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \qquad (3.24)$$

where **A** is lossless when $|g| = 1$, because of its interesting properties when used in a multi-channel output system. However, significant improvement can be obtained by using a unitary matrix with no null coefficients. This will produce a maximum density while requiring the same amount of delay lines.

One such matrix is taken from the class of Householder matrices [17] and can be implemented efficiently as we will see:

$$\mathbf{A}_N = \mathbf{J}_N - \frac{2}{N}\mathbf{u}_N\mathbf{u}_N^T \qquad (3.25)$$

where $\mathbf{J}_N$ is an N x N permutation matrix, and $\mathbf{u}_N$ is an N x 1 column vector of ones. The resulting matrix contains two different non-zero values, thus maximizing the echo density. Also, $\mathbf{A}_N \cdot x(z)$ can be computed efficiently by permuting the elements of $x(z)$ according to $\mathbf{J}_N$, and adding to these the sum of the elements of the input times the factor $-\frac{2}{N}$. This requires around $2N$ operations, compared to the N^2 operations usually required for an N x N matrix multiplication. For the special case where $\mathbf{J}_N$ is the identity matrix $\mathbf{I}_N$, the resulting system behaves like a parallel comb filter with a maximum echo density as shown in Fig. 3.9. Jot noted that this system produces periodic clicks at a time interval equivalent to the sum of all delay lengths. However, these clicks can be avoided in the output of the system by inverting the sign of every other coefficient $\mathbf{c}_N$.

Choosing $\mathbf{J}_N$ to be a circular permutation matrix is another interesting possibility, because in this configuration, the delay lines feed each other serially. This simplifies memory management in the final implementation of the system.

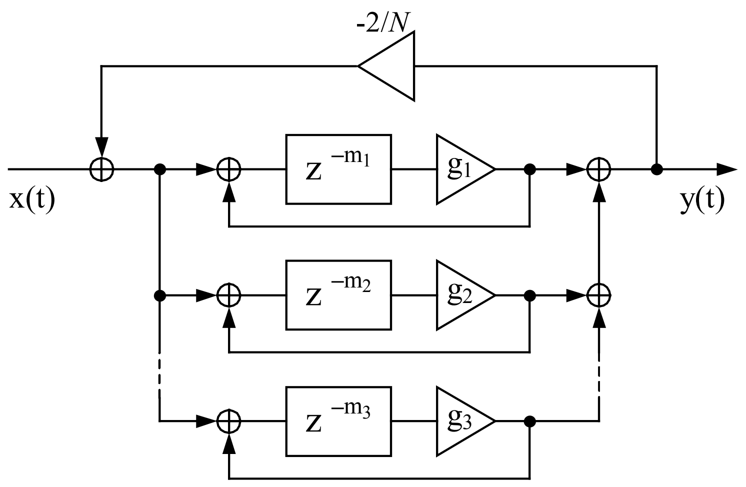


Fig. 3.9. Parallel comb filter modified to maximize the echo density.

Rocchesso and Smith [31] also suggested using unitary circulant feedback matrices having the form:

$$\mathbf{A} = \begin{bmatrix} a(0) & a(1) & \cdots & a(N-1) \\ a(N-1) & a(0) & \cdots & a(N-2) \\ \vdots & \vdots & \ddots & \vdots \\ a(1) & a(2) & \cdots & a(0) \end{bmatrix} \tag{3.26}$$

We can use the Discrete Fourier Transform (DFT) matrix **T** to diagonalize the circulant matrices:

$$\mathbf{A} = \mathbf{T}^{-1}\mathbf{D}\mathbf{T} \tag{3.27}$$

This implies that the eigenvalues of **A** can be computed by finding the DFT of the first row:

$$\{\lambda(\mathbf{A})\} = DFT\left(\left[a(0) \quad \cdots \quad a(N-1)\right]^T\right) \tag{3.28}$$

where {λ(**A**)} are the eigenvalues of **A** (the diagonal elements of **D**). The circulant matrix will be lossless if these eigenvalues have unit modulus. There are two main advantages in the use of circulant feedback matrices: a) specific eigenvalues can be specified during the design of the system, and b) the matrix multiplication only takes about $N \log(N)$ operations to compute since the Fast Fourier Transform can be used.

3.4 Comparing FDN with Waveguide Networks

Fig. 3.10 represents an N-branch Digital Waveguide Network (DWN) with a structure equivalent to an N^{th} order FDN. Each branch of the structure represents an FDN delay line and the length of each waveguide is half the length of the equivalent delay line. This is because a traveling wave must travel from the center junction to the termination of the branch and then back to the center junction. Odd delay line lengths can be converted to an even length waveguide with a reflecting termination made of a single delay element.

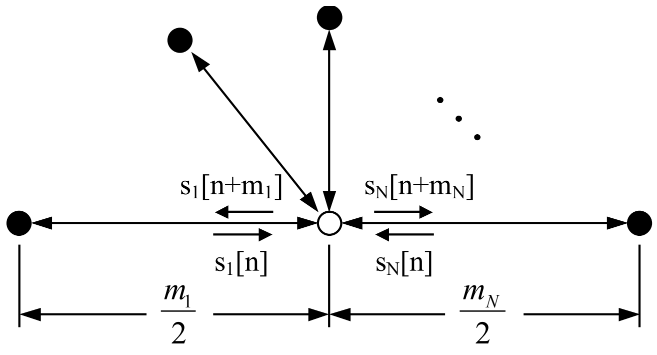


Fig. 3.10. Waveguide network consisting of a single scattering junction to which N branches are connected. Each branch is terminated with a perfect non-inverting reflection, indicated by a black dot [46].

By defining $p_i^+ = s_i(n)$ and $p_i^- = s_i(n+m_i)$, the usual DWN notation is obtained:

$$\mathbf{p}^- = \mathbf{A}\mathbf{p}^+ \qquad (3.29)$$

where $\mathbf{p}^+$ is the vector of the incoming traveling wave samples arriving at the junction (the delay-line outputs in an FDN) at time n, $\mathbf{p}^-$ is the vector of outgoing traveling wave samples (the delay-line inputs in an FDN) leaving the junction at time n, and $\mathbf{A}$ is the *scattering matrix* (the feedback matrix in an FDN) associated with the waveguide junction.

Rocchesso and Smith [31] have shown that the scattering matrix is said to be lossless if the total complex power at the junction is scattering invariant, i.e.,

$$\mathbf{p}^{+*}\mathbf{\Gamma}\mathbf{p}^{+} = \mathbf{p}^{-*}\mathbf{\Gamma}\mathbf{p}^{-} \tag{3.30}$$

$$\mathbf{A}^{*}\mathbf{\Gamma}\mathbf{A} = \mathbf{\Gamma} \tag{3.31}$$

where $\mathbf{\Gamma}$ is a Hermitian, positive-definite matrix which can be interpreted as a generalized junction admittance. In the case of Fig. 3.10, each waveguide has a characteristic admittance $\mathbf{\Gamma}_i$ and the matrix $\mathbf{\Gamma} = \mathrm{diag}(\mathbf{\Gamma}_1, \mathbf{\Gamma}_2, \ldots, \mathbf{\Gamma}_N)$. In the case where $\mathbf{A}$ is unitary, $\mathbf{\Gamma} = \mathbf{I}$. Thus, a DWN where each characteristic admittance $\mathbf{\Gamma}_i$ is unity is equivalent to an FDN with a unitary feedback matrix. We can see that the DWN leads to a more general class of lossless scattering matrices, because each waveguide may have a characteristic admittance different than unity. However, not all lossless scattering matrices can be interpreted as a physical junction of N waveguides (for example, a permutation matrix).

4 TIME-VARYING FDN REVERBERATORS

Several types of time varying modifications can be made to a general FDN algorithm to enhance the sound of its reverberation or to add life to an otherwise static reverberation. For example, changing the feedback matrix of an FDN in real-time would create a reverberator whose timbre changes with time. One could also change the output matrix (coefficients $\mathbf{c}_N$ in Fig. 3.7). This thesis investigates the use of modulation to change the FDN delay lengths in real time. Changing these delay lengths causes the resonant frequencies to change constantly, thus achieving a reverberation with a flatter perceived frequency response.

Fig. 4.1 and Fig. 4.2 show a typical modulated delay line where the output position of the delay line moves constantly around a center position. The output of the delay line is given by:

$$y(n) = x(n - D) \qquad (4.1)$$

$$\text{where } D = \text{NOMINAL\_DELAY} + \text{MOD\_DEPTH} \cdot y_{\text{mod}}(n) \qquad (4.2)$$

Here, NOMINAL_DELAY is the center position of the modulation, MOD_DEPTH is the excursion of the modulation on each side of the center position, $y_{\text{mod}}(n)$ is the output of the waveform generator, and D is a *positive real number* that can be split into an integer and a fractional part:

$$D = Int(D) + d \qquad (4.3)$$

When $d \neq 0$, the value of $x(n - D)$ falls between two samples and thus must be interpolated.

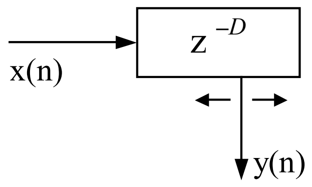


Fig. 4.1. A modulated delay line.

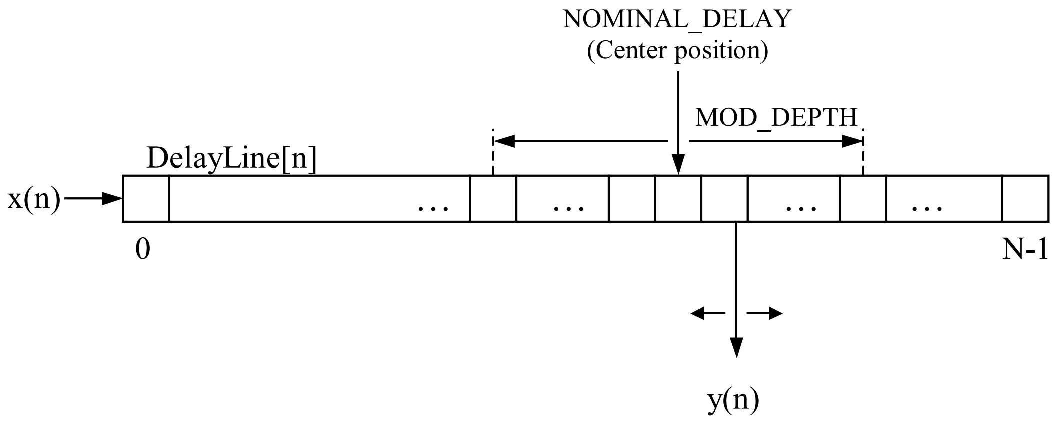


Fig. 4.2. Detailed version of the delay line of Fig. 4.1.

Different waveform types can be used as modulation sources, and different techniques can be used to interpolate the output value of the delay line. The most interesting modulation types and interpolation types will be presented respectively in sections 4.1 and 4.2.

4.1 Modulation Types

4.1.1 Sinusoid Oscillator

The most common modulation source is a simple sinusoid oscillator. There are many ways to implement such oscillators. One of them uses look-up tables where coefficients are already computed and stored in memory. However, since the purpose of this thesis is to produce an algorithm that requires low memory requirements (in addition to being efficient), we chose another way to do it requiring only a few words of memory. The implementation is based on the *z*-transform of the sin function:

$$Y(w) = F\{\sin(w_0)\}X(w) = \frac{\sin(w_0)z^{-1}}{1 - 2\cos(w_0)z^{-1} + z^{-2}} X(w) \tag{4.4}$$

<=>

$$\left(1 - 2\cos(w_0)z^{-1} + z^{-2}\right)Y(w) = \sin(w_0)z^{-1}X(w) \tag{4.5}$$

where $w_0 = \left(\frac{2\pi f_{osc}}{f_s}\right)$, f_{osc} is the desired oscillator frequency, and f_s is the sampling rate. $X(w)$ is an impulse signal used to start the oscillator, defined in the time domain by:

$$\begin{aligned} x(n) &= 1, \ \text{for } n = 0 \\ x(n) &= 0, \ \text{for } n \neq 0 \end{aligned} \tag{4.6}$$

Taking the inverse *z*-transform of (4.5), the following time domain equations are obtained:

$$\begin{aligned} y(n) - 2\cos(w_0)y(n-1) + y(n-2) \quad &= \sin(w_0)x(n-1) \\ &= 0 \quad \text{for n} > 1 \end{aligned} \tag{4.7}$$

<=>

$$y(n) = 2\cos(w_0)y(n-1) - y(n-2) \quad \text{for n} > 1 \tag{4.8}$$

where the initial conditions are given by:

$$y(0) = \sin(-w_0) = \sin\left(-\frac{2\pi f_{osc}}{f_s}\right) \tag{4.9}$$

$$y(1) = \sin(0) = 0 \tag{4.10}$$

where f_{osc} is the desired oscillator frequency and f_s is the sampling rate. It is important to note that this algorithm can be unstable for some frequencies, due to the finite word length of the implementation platform. More details on the implementation of this algorithm will be presented in section 5.4.1.

4.1.2 Filtered Pseudo-Random Number Sequence

We can also use a low-pass filtered pseudo-random number sequence as a modulation source. In practice, a simple uniform pseudo-random number generator (RNG) is sufficient and can be implemented efficiently. One class of RNG proposed by Knuth [23] is based on the linear congruence method, which uses the following equation:

$$y(n+1) = (ay(n)+c) \bmod m \qquad (4.11)$$

The initial value or *seed* is generally not important, unless multiple RNG are used at the same time. In this case, it is desirable to choose a unique seed for each of them. The choice of parameters a and c is crucial since a good generator should generate m values before repeating the output sequence, and every number between 0 and m-1 should be output once in every period of the generator. However, if multiple generators have to be used at the same time, a different seed should be chosen for each RNG so that all the sequences appear to be different from each other. The modulus m is usually chosen according to the word length of the implementation platform, to execute the mod function "transparently." For example, we chose a modulus of 32 because our implementation platform uses a 32-bit word length processor. More implementation details will be given in section 5.4.2. The other parameters we used (a = 1664525 and c = 32767) were chosen according to the rules of Knuth.

Once we get the raw pseudo-random number sequence, we must process it to obtain smooth oscillations. We can achieve slow oscillations using one pseudo-random number for every f_s/N sample, where f_s is the sampling rate and N is a constant to be determined experimentally. We could then obtain the intermediate values using linear interpolation. However, the abrupt changes in oscillation slopes would create sudden pitch changes in the reverberation output making this method unsuitable for our application. A better sounding (although more computationally intensive) approach is to pad zeros in the intermediate points and to low-pass filter the resulting sequence. Zero padding the original sequence makes the entire process more efficient since we compute one random number every N samples (we found that $N = 100$ gave good results). We tried several low-pass filters and found that a fourth order Chebyshev filter (obtained using the bilinear transform), with a cut-off frequency of 2 Hz and a pass-band ripple of 0.5 dB was required to obtain a smooth output with no specific prominent oscillation frequency. The resulting signal with a seed of 2 is shown in Fig. 4.3.

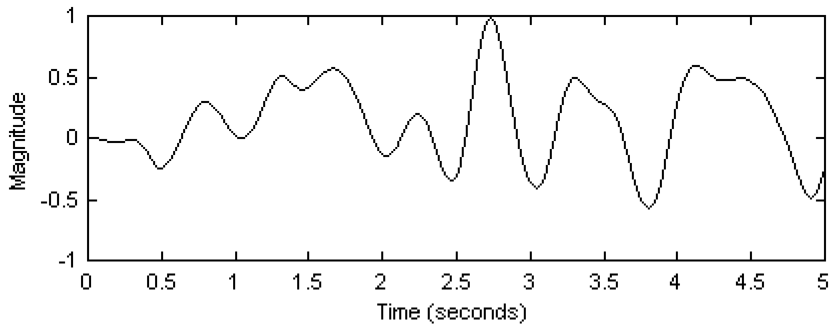


Fig. 4.3. Filtered pseudo-random number sequence.

4.1.3 Other Common Waveform Generators

Triangular waveform generators do not give good results when used as modulators, as they will produce reverberation tails consisting of two distinct pitches every period. The first pitch will correspond to the rising ramp and the second pitch will correspond to the falling ramp. To avoid this sudden change in pitch, we could low-pass filter the waveform. However, this would require an additional amount of computations.

The resulting waveform would then be similar to a sinusoidal waveform, but would require more computations. Since our goal was to reduce computations to a minimum, we did not use filtered triangular waveforms in our implementation.

Another common waveform is the sawtooth. Even if sawtooth modulators produce a reverberation tail having a constant pitch, the tail will suffer from clicks every generator period caused by the abrupt transition in the waveform. Again, appropriate low-pass filtering could remove these clicks, but it would result in an unacceptable increase in computation time.

4.2 Interpolation

In a time-varying implementation, a modulator is constantly changing the length of a specific delay line. For example, from a minimum delay of 500 to a maximum delay of 508 samples. Thus most of the intermediate delay values will not be exact integers (for example 501.25), so a delay of a fraction of a sample will be requested. Since the modulator rate will be slow (no more than a few hertz) and only a few samples wide, the delay length can not simply be rounded to the nearest output sample in the delay line. It is important to use interpolation, as rounding will result in audible noise or clicks in the reverberation tail.

4.2.1 General Concepts

The *z*-domain transfer function of the ideal delay of Fig. 4.1 is:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{z^{-D}X(z)}{X(z)} = z^{-D}X(z) \qquad (4.12)$$

The frequency response of this ideal system is given by:

$$H_{id}(e^{j\omega}) = e^{-j\omega D} \qquad (4.13)$$

where $\omega = 2\pi ft$ is the normalized frequency and T is the sampling period. Thus the magnitude and phase response of $H(e^{j\omega})$ are respectively:

$$\left|H_{id}(e^{j\omega})\right| \equiv 1 \text{ for all } \omega \qquad (4.14)$$

$$\arg\{H_{id}(e^{j\omega})\} = \Theta(\omega) = -D\omega \qquad (4.15)$$

Finally, we find the group delay by taking the derivative of the phase response and inverting its sign:

$$\tau_{g,id}(\omega) = -\frac{\partial\Theta(\omega)}{\partial\omega} = D \qquad (4.16)$$

The implementation of a fractional delay system to interpolate values from a band-limited signal can be considered as an approximation of the ideal linear-phase all-pass filter with a unity magnitude and a constant group delay D.

The impulse response of such a filter is computed by taking the inverse Fourier transform:

$$h_{id}(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H_{id}(e^{j\omega})e^{j\omega n}d\omega \text{ for all } n \qquad (4.17)$$

Using $H_{id}(e^{j\omega})$ found in (4.13), the ideal impulse response of the all-pass filter can be found:

$$h_{id}(n) = \frac{\sin[\pi(n-D)]}{\pi(n-D)} = \text{sinc}(n-D) \quad \text{for all } n \qquad (4.18)$$

For integer values of D, the impulse response is a single impulse centered at $n = D$. For all other values of D, the impulse response corresponds to a shifted equivalent of the sinc function centered at *D*. Such a filter is impossible to realize in a real-time system since it is infinitely long and also non-causal.

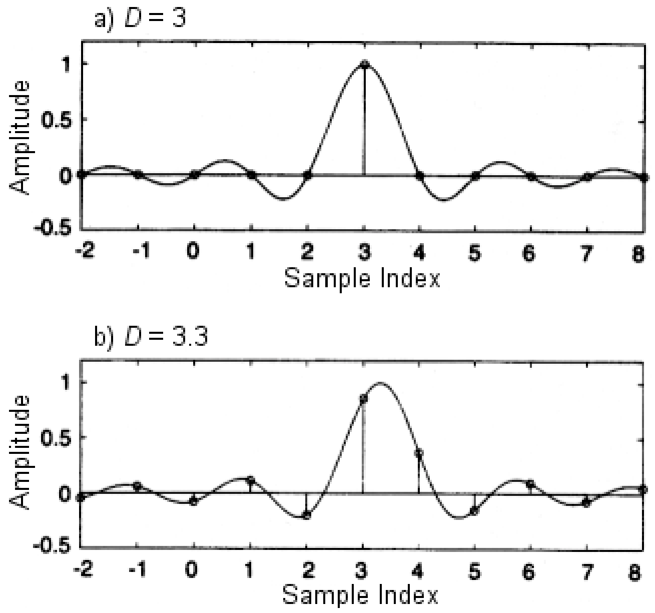


Fig. 4.4. Impulse response of an ideal all-pass filter with delay a) $D = 3$ and b) $D = 3.3$ (from [25]).

However, different methods (both FIR and IIR) have been devised to approximate this ideal all-pass filter function. Laasko [25] provides an extensive summary of these interpolation types and detailes their properties. In this project, two interpolation types have been selected (one FIR and one IIR), based on their efficiency and computation requirements. They are presented in the following sections.

4.2.2 <u>Lagrange Interpolation (FIR)</u>

The FIR methods all have the same *z*-domain transfer function:

$$H(z) = \sum_{n=0}^{N} h(n) z^{-n} \tag{4.19}$$

Coefficients are determined such as the norm of the frequency-domain error function

$$E(e^{j\omega}) = H(e^{j\omega}) - H_{id}(e^{j\omega}) \qquad (4.20)$$

is minimized.

The idea behind Lagrange interpolation is to make this error function maximally flat at a certain frequency, typically at $\omega_o = 0$, so that the approximation is at its best close to this frequency. This can be achieved by making the derivatives of the frequency-domain error function equal to zero at this point:

$$\left. \frac{d^n E(e^{j\omega})}{d\omega^n} \right|_{\omega=\omega_o} = 0 \quad \text{for } n = 0, 1, 2, \ldots N \qquad (4.21)$$

Differentiating and substituting $\omega_o = 0$ in equation (4.21), a set of $L = N + 1$ linear equations are obtained. They can be expressed as an impulse response of the form:

$$H(z) = \sum_{n=0}^{N} k^n h(k) = D^n \quad \text{for } n = 0, 1, 2, \ldots N \qquad (4.22)$$

or, in the matrix notation form:

$$\mathbf{Vh} = \mathbf{v} \qquad (4.23)$$

where **h** is a vector containing the coefficients, **V** is an L x L *Vandermonde* matrix:

$$\mathbf{V} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 2 & & N \\ 0 & 1 & 2^2 & & N^2 \\ \vdots & & & \ddots & \vdots \\ 0 & 1^N & 2^N & \cdots & N^N \end{bmatrix} \qquad (4.24)$$

and **v** is:

$$\mathbf{v} = \begin{bmatrix} 1 & D & D^2 & \cdots & D^N \end{bmatrix}^T \qquad (4.25)$$

The solution to (4.21) is equal to the classical *Lagrange interpolation* formula and the coefficients are obtained by making the interpolation polynomial pass through a given set of data values. This leads to the solution:

$$h(n) = \prod_{\substack{k=0 \\ k \neq n}}^{N} \frac{D-k}{n-k} \quad \textit{for } n = 0, 1, 2, \ldots N \tag{4.26}$$

If N equals 1, this corresponds to *linear interpolation* between two samples. In this specific case, $h(0) = 1 - D$ and $h(1) = D$. The magnitude and phase delay responses of low-order Lagrange interpolation filters are shown in Fig. 4.5 and Fig. 4.6.

As we can see, the even length filters (L = 2, 4, and 6) are linear phase. However, their magnitude responses fall to zero at $\omega = \pi$. On the other hand, the odd length filters (L = 3 and 5) have a better magnitude response, but their phase delays are worse. An important property that holds for all Lagrange interpolation filters is that their frequency response never exceeds 0 dB. This is important in our case since stability must be ensured in the feedback network.

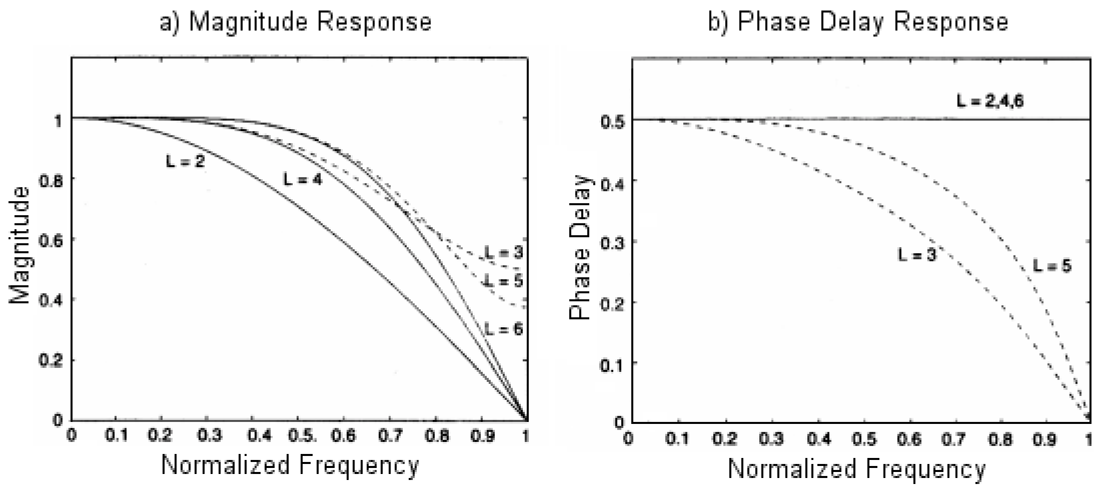


Fig. 4.5. Lagrange interpolating filters of length L = 2, 3, 4, 5, and 6 with d = 0.5 (from [25]).

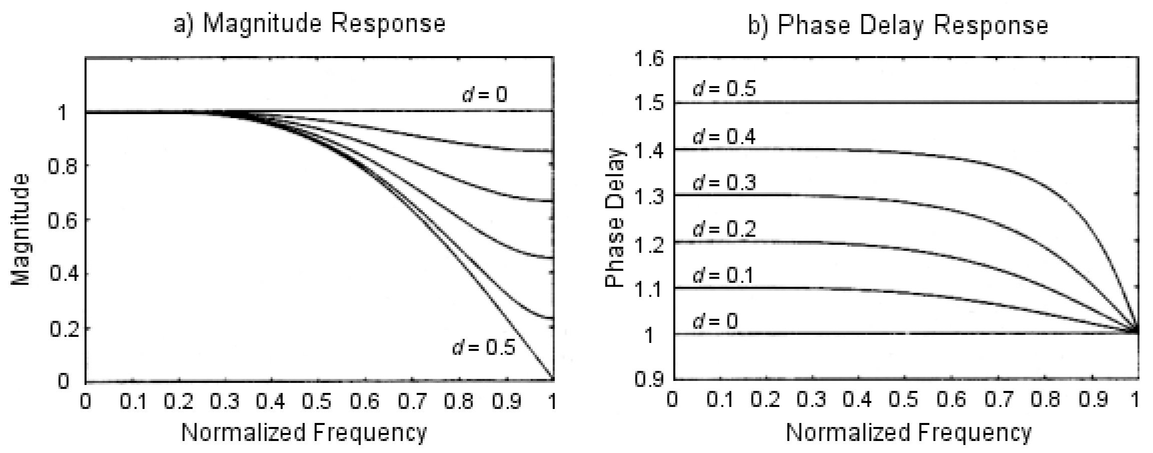


Fig. 4.6. Lagrange interpolating filter of length L = 4, with $d = 0$ to 0.5 (from [25]).

4.2.3 Maximally Flat Group Delay Design of All-Pass Filters (IIR)

In general, recursive IIR filters are able to achieve the same frequency domain specifications as FIR filters with a smaller number of multiplications. However, their design is more complicated and they can be unstable if some of the poles move outside the unit circle during real-time coefficient updates. The z transfer function of an N^{th}-order all-pass filter is of the form:

$$A(z) = \frac{z^{-N} D(z^{-1})}{D(z)} = \frac{a_N + a_{N-1} z^{-1} + \ldots + a_1 z^{-(N-1)} + z^{-N}}{1 + a_1 z^{-1} + \ldots + a_{N-1} z^{-(N-1)} + a_N z^{-N}} \qquad (4.27)$$

where the numerator is the mirrored version of the denominator $D(z)$.

The simplest IIR filter design is based on Thiran's analytic solution for an all-pole low-pass filter with a maximally flat group delay response at the zero frequency. Since the group delay of an all-pass filter is twice the corresponding all-pole filter, each delay value of the all-pole Thiran formulas must be doubled. The solution for the all-pass filter coefficients approximating the delay $D = N + d$ is:

$$a_k = (-1)^k \binom{N}{k} \prod_{n=0}^{N} \frac{D-N+n}{D-n+k+n} \quad for \;\; k = 0,\, 1,\, 2,\, \ldots\, N \tag{4.28}$$

where $\binom{N}{k} = \dfrac{N!}{k!(N-k)!}$ is a binomial coefficient [25]. Coefficient a_0 is always equal to 1, so the polynomial is automatically scaled as desired. Thiran also proved that for large enough delay D, the resulting all-pass filter would always be stable. Phase delay curves of some maximally flat group delay designs are shown in Fig. 4.7 and Fig. 4.8. We can see that the delay response is linear over a large frequency range, even for small order filters.

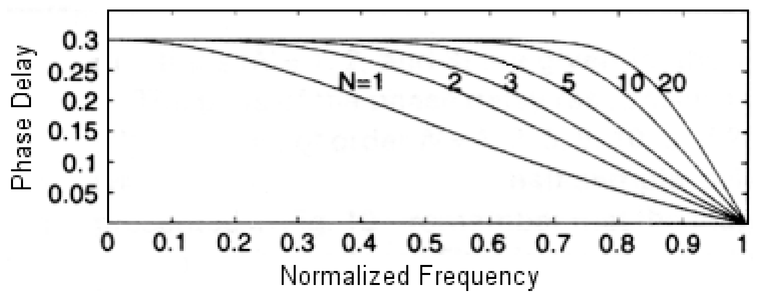


Fig. 4.7. Phase delay curves of 1, 2, 3, 5, 10, and 20th-order all-pass filters with $d = 0.3$ (from [25]).

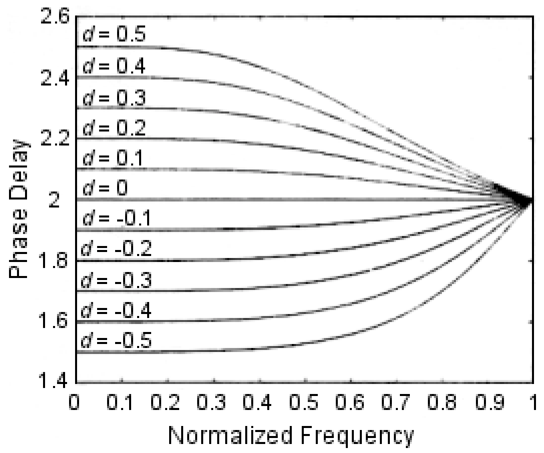


Fig. 4.8. Phase delay curves of 2nd-order all-pass filter with $d = -0.5$ to 0.5 (from [25]).

Dattorro [4] also suggested that equation (4.28) could be approximated by:

$$a_k = (-1)^k \binom{N}{k} \prod_{n=0}^{N} D - N + n \quad for\ \ k = 0,\ 1,\ 2,\ \ldots\ N \qquad (4.29)$$

where $\binom{N}{k} = \frac{N!}{k!(N-k)!}$. This approximation increases the phase distortion of the filter, but as he noted, "in some musical contexts the induced phase distortion is not subjectively objectionable." In these applications, using this approximation can save computations.

4.3 Previous Usages of Modulation in Artificial Reverberation

The idea of using delay length modulation in reverberation algorithms is not new. It was first used in early hardware reverberation units such as the original EMT model 250 (released in 1975). In those days, memory was limited and the use of modulation to randomize delay lengths was imperative to emulate the eigenmode density of a good sounding room. Even today, many reverberation applications still have limited memory and computation resources (such as game platforms or multi-media applications). Note that high-end commercial reverberation units such as the Lexicon 480L and the TC Electronics M3000 also use modulation to produce richer reverberation. However, very little has been available in the literature on time-varying techniques.

4.3.1 Dattoro's Plate Reverberator

As mentioned in section 2.7.2 when Dattoro's plate reverberator was described, modulation can be use to reduce the coloration of the reverberation tail. Dattorro suggested the use of low-frequency oscillators (LFO) to modulate some (or all) of the delay lines in the tank. He suggested the use of linear (1st order Lagrange) interpolation or even all-pass interpolation for better results. He noted that the use of modulation was avoiding the build-up of resonances, especially when using high-frequency content input signals such as drums or percussions. However, he also noted that modulation must be used with care to avoid audible vibrato (pitch modulation inherent to the modulation of the delay lengths) on instruments like piano.

For applications where processing power is available, he suggested the use of different modulation rates and depths for each delay line. For applications where processing time is limited, he suggested the use of sinusoids oscillators in quadrature to modulate only two of the four delay lengths of the tank.

4.3.2 Smith's Application Notes Related to Modulation

At the end of his first paper on artificial reverberation using digital waveguides [44], Julius Smith included in [45] a few notes on the use of modulation. He mostly emphasized modulating the scattering coefficients of waveguide reverberation algorithms (which is equivalent to modulating the feedback matrix of a FDN network). However, he also discussed varying the algorithm's delay lengths. He compared this delay length variation to the movement of the walls of a room. To avoid energy modulation, he suggested reducing the length of one delay line while simultaneously increasing the length of another delay line by the same amount. We will use this method in our final implementation since it also helps mask the perceived pitch change during the reverberation tail.

4.3.3 Griesinger's Time-variant Synthetic Reverberation

Griesinger presented a way to improve the acoustics of a room through time-variant synthetic reverberation [13]. He used artificial reverberation in an electroacoustical sound reinforcement system to increase the perceived reverberation of a hall during musical performances. To reduce feedback normally present in such a reinforcement system, he modulated the delay lengths in the reverberation algorithm.

He noted that interpolation was a necessity to avoid noise and clicks in the reverberation tail, as mentioned earlier. He also noted that since delay modulation results in a pitch shift, the modulation sources must be selected with care. For example, if RNG were used as modulators on all delay lines, all of them could go in the same direction at the same time and thus would create a noticeable shift in the pitch of the reverberation.

5 IMPLEMENTATION AND RESULTS

The general time-invariant FDN we used as a starting point for our experiments was described in detail in chapter 1 and is shown in Fig. 3.8. Before adding any modulation to the network, we had to find some basic parameters such as a feedback matrix, the delay lines' lengths, coefficients of vectors **b** and **c**, and absorption and the transfer function of the tone correction filter. The selection procedure we used to find these parameters will be detailed in section 5.1.

Once we had a time-invariant algorithm with sufficient echo and frequency density to simulate a good sounding room, we added modulation to the network's delay lines and investigated the effect of each component participating in the modulation process. All our tests were performed with a variety of input signals, including dry recordings of piano, saxophone, and impulse responses. Section 5.2 will describe the effect of a change in the modulation source rate and depth. Section 5.3 will then evaluate different interpolation types, using the best sounding modulation rate and depth determined in section 5.2.

We will then analyze each modulation and interpolation method's computation requirement, and propose some algorithm optimizations to reduce the overall processing requirements. The chapter will conclude with the presentation of the results of a listening test evaluating the proposed algorithm.

5.1 Choice of the general FDN parameters

The choice of the feedback matrix is important in order to get a good echo density, while keeping the computations to a minimum. The most common lossless feedback matrices were presented in section 3.3.3. We saw that some specific matrices, such as the identity matrix and triangular matrices, lead to the well known parallel comb filter and series all-pass filters, respectively. However, as we saw previously, these matrices do not provide a sufficient level of echo density (using a reasonable number of delay lines), because they contain many null coefficients.

Rocchesso and Smith's circulant matrices and Jot's Householder matrices described previously can both provide a maximum echo density while enabling relatively fast matrix multiplications. We chose to use Jot's class of Housholder matrix (defined in (3.25), where **J** is a circular permutation matrix) in our tests because it was the one that allowed the most efficient matrix multiplication.

We used the unit vector $\mathbf{b} = [1 \ 1 \ \ldots \ 1]^T$ as the injection matrix to obtain the maximum echo density. We generated a stereo output by using **c** as an N x 2 matrix as suggested in section 3.3.1, instead of an N x 1 vector:

$$\mathbf{c} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & -1 \\ \vdots & \vdots \\ 1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & -1 \end{bmatrix} \qquad (5.1)$$

We set up the algorithm so that the first column of **c** corresponds to the left channel output of the reverberation algorithm and the second corresponds to the right channel. The first column was chosen to avoid clicks in the reverberation tail (as explained in section 3.3.3). Since listening tests [15] [39] suggested that improved spatial impression can be obtained by decorrelating the reverberation presented to the listeners' ears, the second column of **c** (corresponding to the right channel output) was chosen to be as different as possible from the first one. This will produce two outputs that will be perceived as being uncorrelated, thus improving the spatial image of the reverberation.

The common way is to choose the delay lengths to be incommensurate to avoid superposition of peaks in the time and frequency responses, as mentioned in section 3.2. We found that prime numbers helped in selecting delay line lengths that did not overlap in the time domain. To avoid superposition of peaks in the frequency domain, we plotted the frequency response of the reverberation using different prime delay line length combinations and picked the set of delay length that produced the flatter frequency response.

At least 12 delay lines were needed to obtain the echo and time density for a reverberation of two seconds using a small amount of memory:

$$\mathbf{D}(z) = diag\left(z^{-601} \quad z^{-691} \quad z^{-773} \quad z^{-839} \quad z^{-919} \quad z^{-997} \quad z^{-1061} \quad z^{-1093} \quad z^{-1129} \quad z^{-1151} \quad z^{-1171} \quad z^{-1187}\right) \tag{5.2}$$

Note that these delay lengths lead to a frequency density $D_f = 0.26$ and a memory consumption of less than 12k words (about one quarter of a second on our 44.1 kHz sampling rate system). To put this memory consumption in perspective, a complete reverberation algorithm on a Sony PlayStation I and II can consume up to 100 kBytes.

By using modulation, we would like to achieve the same perceived reverberation quality as this 12 delay line system, using only 8 delay lines selected as:

$$\mathbf{D}(z) = diag\left(z^{-601} \quad z^{-691} \quad z^{-773} \quad z^{-839} \quad z^{-919} \quad z^{-997} \quad z^{-1061} \quad z^{-1129}\right) \tag{5.3}$$

This system has a frequency density $D_f = 0.16$ and uses a total memory of less than 7k words (less than 160 ms). This represents about 60% of the memory used by the 12 delay line algorithm.

Selecting the delay lines' length was an important factor in the overall sound of the reverberation algorithm. According to (3.5), the total amount of delay in the algorithm corresponds to the frequency density of the algorithm. Section 2.4 also mentioned that the reverberation of a medium sized hall with an RT of 1.8 seconds has a frequency response consisting of peaks spaced by an average of 2.2 Hz over 20 Hz. This leads to a frequency density $D_f = 0.45$ and would thus require a total delay length of 450 ms to simulate artificially (about 20k words at a sampling rate of 44.1 kHz). However, by selecting the delay lengths judiciously, we were able to achieve a time-invariant reverberation algorithm with RT = 2 seconds that had a limited amount of resonant frequencies with almost half of this frequency density ($D_f = 0.26$). Our objective is to achieve the same perceived frequency density using a time-variant reverberation with a modal density of only 0.16 modes per Hertz.

We already described in section 3.2.4 the absorption filters and tone correction filters used in section 3.2.4. During the development process, we used a reverberation time of 3 seconds for the low frequency and 1.25 seconds for the high frequency, as these values are close to the average reverberation times of good sounding halls. The tone correction filter was set to correct the effects of the absorption filters using the method discussed in section 3.2.4. The EDR of the non-modulated algorithm with 12 delay lines is shown in Fig. 5.1.

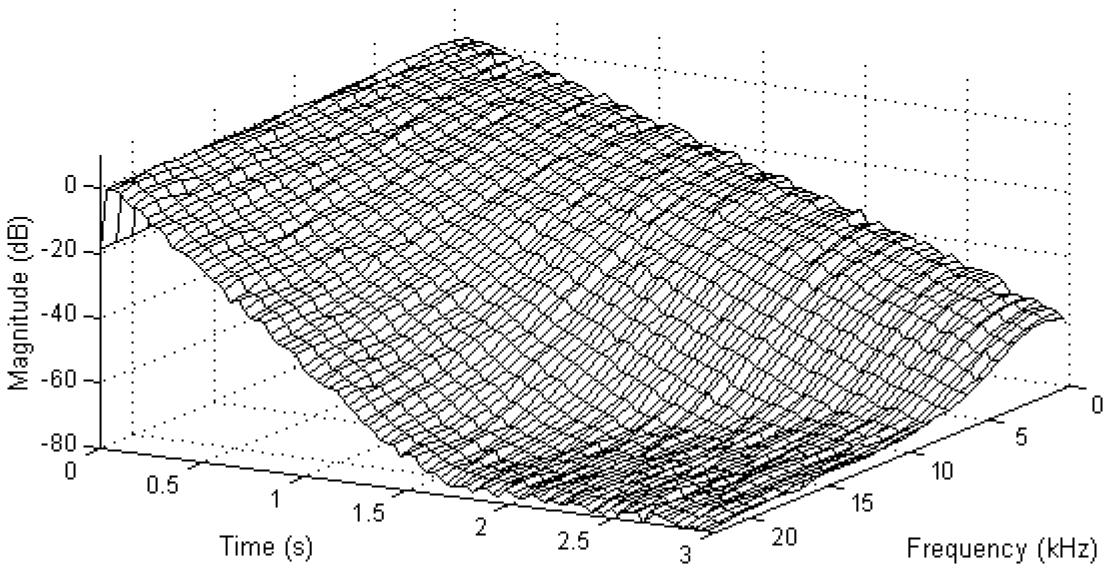


Fig. 5.1. Energy decay relief of a time invariant FDN with 12 delay lines.

5.2 Choice of Modulation

5.2.1 Modulation Rate and Depth

To clearly see the effect of different modulation rates and depth, we first tried a simple network consisting of only one delay line. We used a sinusoidal tone generator as the modulation source, combined with high-quality interpolation. We tested nominal delay lengths of 600, 1000, and 1500 samples.

The efficiency of modulation to avoid build-up of resonant frequency is proportional to the modulation depth. However, the depth of modulation is also directly related to the amount of pitch change in the reverberation. It is thus important to select the modulation depth carefully to avoid chorusing effects or vibrato in the reverberation tail. We found that a modulation depth of 8 or more samples resulted in a perceivable pitch change on input signals that had sustained tones such as saxophone or piano. On the other hand, we found that a modulation depth of 4 samples did not help to break the resonances in the reverberation tail. We thus chose a modulation depth of 6 samples in our final implementation since it is a good compromise between modulation efficiency and perceivable pitch change.

The optimum modulation rate was more difficult to determine, since every delay length seemed to have a specific modulation frequency that worked better than the others. The only common rule was that a modulation of less than 0.5 Hz resulted in almost no improvement to the equivalent unmodulated output. On the other hand, modulation rates of more than 2 Hz could be perceived quite easily on some combinations of delay lines and input signals.

We then tried a complete FDN with 8 delay lines to see if these hypotheses would still hold. We found that when we combined all modulators in phase, we were able to hear a clear pitch change, even with moderate modulation depths and rates. However, with a phase difference of 45° or 90° between each modulator, the pitch modulation was more confused, and there was no noticeable change in the overall pitch. We found that even with a modulation depth of 6 samples, there was no apparent pitch change.

5.2.2 Modulation Type

All the previous tests where done with sinusoids, however pseudo-random sequences where also used as modulators. We found that we were able to get a smoother reverberation tail, with no specific pitch change, when using a different seed in the modulator of each delay line. However, even with different seeds, there is always a risk that all modulators move in the same direction at the same time. This punctual behavior creates a distinct pitch change in the output. There is also a risk of having low amplitude output from all modulators during a certain period of time. This would cause the modulation to be less efficient. Using sinusoidal modulators solves both problems since all the modulators are always offset by the same phase amount. It also creates a more consistent result in reducing resonant frequencies since the amplitudes of the modulators are constant.

5.3 Choice of Interpolation Type

Using a sinusoidal modulator with a rate of 2 Hz and a depth of 6 samples, we then tried different interpolation types. We saw in the previous chapter that Lagrange interpolation (an FIR design) caused the signal's high frequencies to be attenuated. Clearly, for order $N = 1$ (linear interpolation) and $N = 2$, this roll-off is not acceptable since it is too pronounced for a natural sounding reverberation. A fourth order or higher Lagrange interpolation is required to obtain an almost flat perceived frequency response.

Luckily, an IIR all-pass design, such as the maximally flat group delay interpolation, enables us to achieve an all-pass response with a first-order design. However, this design may cause some distortion due to transients arising from the time-varying filter coefficients. However, this distortion is low enough to be acceptable in most cases. We tried both the original design (equation (4.28)), and the approximation of the original design (equation (4.29)). In all the inputs we tried, we were not able to hear any difference between the two versions of the all-pass interpolation algorithm.

5.4 Implementation Details

This section will look at some of the interesting optimizations and implementation methods that were used to make the overall reverberation algorithms more efficient. The algorithm was coded as a 32-bit application in C++, designed for the Intel Pentium processor family. However, we should note that a commercial application for this platform would typically require further optimization using assembly language.

5.4.1 Sinusoidal Tone Generator Implementation

The sinusoidal algorithm we used (described in section 4.1.1) consists of only one multiplication and one addition. Fig. 5.2 shows the implementation of this algorithm. Note that $a = 2 \cdot \cos\left(\frac{2\pi f_{osc}}{f_s}\right)$, where f_{osc} is the tone frequency and f_{s} is the sampling frequency.

```
float Sin::GetCurrentValue(void)
{
      double out = a * buffer1 - buffer2;

      buffer2 = buffer1;
      buffer1 = out;

      return (float) out;
}
```

Fig. 5.2. Implementation of a sinusoidal tone generator.

It is important to note that this algorithm is not stable for all frequencies, even with double precision, due to round-off errors. For example, coefficient a must be stored in memory using a finite word length. For low frequencies, the value of the cosine used to compute coefficient a is closed to 1. For example, if f_{osc} = 2 Hertz and f_s = 44.1 kHz, we have $a = 2\cos\left(\frac{2\pi f_{osc}}{f_s}\right) = 1.999999918802521$. The equivalent binary representation of coefficient a is then: 1.1111 1111 1111 1111 1111 1110 1010 0010. Thus, a processor using 16 or 24-bit coefficients would not be able to store this value without introducing a large error in the oscillation frequency. Since obtaining the exact frequency is not critical in our application, this error may be acceptable. There would also be large errors in the storage of intermediate values such as *buffer1* and *buffer2*. Unlike the error on coefficient a, these errors would accumulate every period of the oscillator. In practice, these round-off errors can cause the system to be unstable for some frequencies (even with 32 or 64-bit word lengths).

It is thus desirable to ensure that the system will be stable for any frequency. Also, the output of the system has to stay in the interval [-1.0, 1.0]. The easiest way to accomplish these requirements is to reset the algorithm when the output goes above 1.0 or below -1.0 as shown in Fig. 5.3, where $resetValue = \sin\left(\frac{\pi}{2} - \frac{2\pi f_{osc}}{f_s}\right)$, f_{osc} is the tone frequency, and f_s is the sampling frequency.

```
float Sin::GetCurrentValue(void)
{
      double out = a * buffer1 - buffer2;

      // Two special cases to avoid instability due to
      // round-off errors
      if(out >= 1.0) {
         out = 1.0;
         buffer2 = resetValue;
      }
      else if(out <= -1.0) {
         out = -1.0;
         buffer2 = -resetValue;
      }
      else
         buffer2 = buffer1;

      buffer1 = out;

      return (float) out;
}
```

Fig. 5.3. Implementation of a sinusoidal tone generator with overflow prevention.

This modification to the original algorithm causes some abrupt, but slight changes in the frequency of the tone when the stabilization mechanism has to reset the algorithm. This causes a change in the modulation rate. However, this has no effect since it is too small to be noticed when used in the context of our application.

5.4.2 Filtered Random Number Generator Implementation

The RNG algorithm itself consists of 2 multiplications and 2 additions. Note that the modulo operation is done automatically by overflowing the processor’s 32-bit register. The resulting algorithm is shown in Fig. 5.4.

```
#define SCALING_FACTOR  0.000000000465661287416f   //  2/(2^32-1)

float RNG::GetCurValue(void)
{
   // Generate a random number from 0 to 2^32-1
   buffer = a * buffer + c;

   // Scale to [-1.0, 1.0]
   float out = (float)buffer * SCALING_FACTOR - 1.0f;

   return out;
}
```

Fig. 5.4. Implementation of a linear RNG.

As mentioned in the previous chapter, this sequence must be filtered by a low-pass filter. The filter we used, described in section 4.1.2, was implemented with two stages of second order biquad filters. Each biquad section can be implemented in a direct form I implementation, as shown in Fig. 5.5.

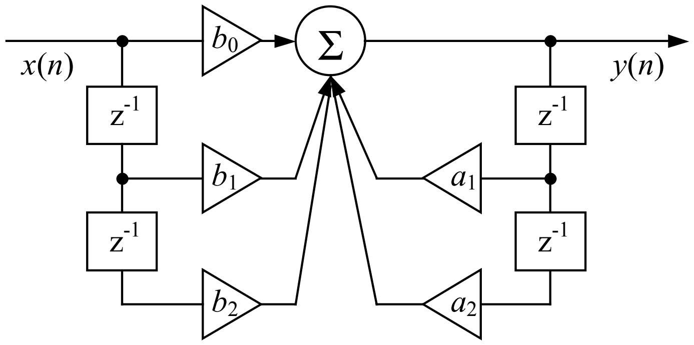
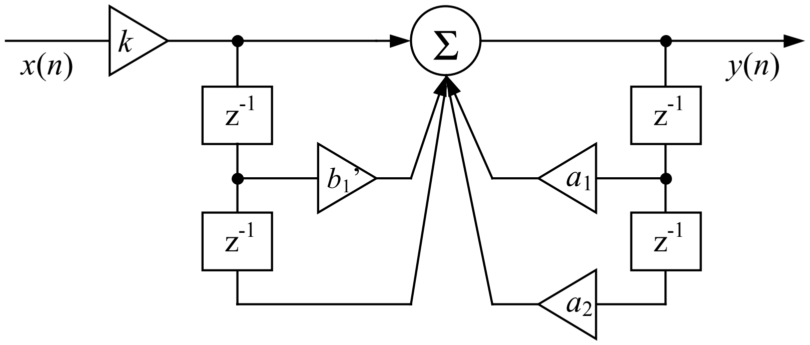


Fig. 5.5. Block diagram of the direct form 1 implementation of a biquad filter.

However, in our specific filter, coefficients b_0 and b_2 were the same so we were able to optimize the algorithm. The final implementation (shown in Fig. 5.6) requires only 4 multiplication and 4 additions:



```
void LPF::Process(const float in, float &out)
{
   float temp = in * k;
   out = temp + bufferX1*b1 + bufferX2 - bufferY1*a1 - bufferY2*a2;

   bufferX2 = bufferX1;
   bufferX1 = temp;
   bufferY2 = bufferY1;
   bufferY1 = out;
}
```

Fig. 5.6. Modified direct form 1 a) block diagram and b) implementation of a biquad filter (where $k = b_0 = b_2$, and $b_1' = b_1/k$).

5.4.3 *Interpolation Implementation*

As we saw in section 4.2.2, Lagrange coefficients can be computed using (4.26), where the denominator factors are pre-stored. This requires a number of multiplications proportional to $(N+1)^2$, where N is the filter order. However, Murphy [28] proposed a way to compute the coefficients that requires only $4(N+1)-8$ multiplications, saving about 50% of the computation time for a 4th order filter. Using this method, the numerator factors are computed by first calculating the partial products:

$$\begin{aligned} ppf_0 &= D \\ ppf_k &= ppf_{k-1} \cdot (D-k) \qquad \text{for} \quad k = 1,2,\dots,N-1 \\ ppr_0 &= (D-N) \\ ppr_k &= ppr_{k-1} \cdot (D-N+k) \qquad \text{for} \quad k = 1,2,\dots,N-1 \end{aligned} \tag{5.4}$$

where $ppf_{0\dots N\text{-}1}$ and $ppr_{0\dots N\text{-}1}$ represents the forward and reverse partial products that have to be temporarily stored. These are then combined to give the numerators of (4.26) using:

$$\begin{aligned} n_0 &= ppr_{N-1} \\ n_{N-1} &= ppf_{N-1} \\ n_1 &= n_0 + ppr_{N-2} \\ n_{N-2} &= n_N - ppf_{N-2} \\ n_k &= ppf_{k-1} \cdot ppr_{N-1-k} \qquad \text{for} \quad k = 2,\dots,N-2 \end{aligned} \tag{5.5}$$

One way to optimize any interpolation process is to update the interpolation coefficients every N^{th} samples. For example, we can set the modulation source frequency to 0.2 Hz, instead of 2 Hz. We can then take its output at 1/10th of the sampling rate to compute the interpolation coefficients once every 10 audio samples. We found that using this method to update the coefficients every 25 samples or less does not result in any audible distortion. Updating them at a rate of 1 update every 50 samples creates a very subtle noise during the reverberation decay. A rate higher than 75 adds audible noise during decay.

5.5 Computation Requirements

The algorithms were executed using floating-point precision on an Intel Pentium II 266MHz processor, with 64 MBytes of RAM. Although every algorithm module such as delay lines and modulators were C++ objects, all critical functions small enough were computed *inline* for ease of design. Therefore, there was basically no overhead associated with function calls. All compiler optimizations were set towards maximum execution speed. The reverberation algorithm processes blocks of 5512 audio samples and an average processing time of more than 200 blocks was used in the following tables as the algorithm's average processing time. We should also note that the algorithm always includes a constant pre-delay, but no early reflections and no tone-correction filter. In a real-world situation, we would add those modules to the algorithm and this would increase the total computation time averages of the algorithm by a constant factor.

Table I shows the average processing time of the original algorithm (without modulation), using 8, 12, and 16 delay lines. We can see that the total computation time of the algorithm is roughly proportional to the number of delay lines, which is about 0.24 μs per delay line.

TABLE I – PROCESSING TIMES OF THE FILTER WITHOUT MODULATION

Algorithm Description	**Total Processing Time**
$N = 16$, with no modulation	3.7 μs
$N = 12$, with no modulation	2.8 μs
$N = 8$, with no modulation	1.9 μs

Table II shows computation time of the same basic algorithm, with 8 modulated delay lines. As we can see, the required processing time of the entire algorithm has increased dramatically compared to the original non-modulated algorithm. We also computed the modulator's algorithm processing time by themselves (shown in the second column). We found out that the sinusoidal tone modulator and RNG modulators took about 30% and 50% of the added computation time while the interpolation algorithms took the remaining 70% and 50%, respectively.

TABLE II – PROCESSING TIMES OF DIFFERENT MODULATOR TYPES

Algorithm Description	**Modulators Processing Time**	**Total Processing Time***
Sinusoidal tone generator	8 x 0.18 μs = 1.4 μs	6.9 μs
Filtered RNG sequence	8 x 0.43 μs = 3.4 μs	8.7 μs

* with $N = 8$ and approximated 1st order all-pass interpolation.

Since we found in section 5.2.2 that the sinusoidal tone modulator gave more consistent results (and was more efficient) than the RNG modulator, we decided to keep it in our final implementation.

Table III shows computation time of different interpolation methods, using the sinusoidal tone modulator. We selected the approximated 1st order all-pass IIR interpolation, because it sounds almost as good as the 4th order Lagrange FIR interpolation and requires about 55% of the computation needed for the latter.

TABLE III – PROCESSING TIMES OF DIFFERENT INTERPOLATION TYPES

Algorithm Description	**Interpolation Time**	**Total Processing Time***
1st order Lagrange FIR	4.0 μs	7.3 μs
2nd order Lagrange FIR	5.0 μs	8.3 μs
3rd order Lagrange FIR	5.4 μs	8.7 μs
4th order Lagrange FIR	6.5 μs	9.8 μs
1st order All-Pass IIR (Approx.)	3.6 μs	6.9 μs
1st order All-Pass IIR	5.0 μs	8.3 μs

* with $N = 8$ and a sinusoidal tone modulator.

One way to optimize the interpolation process is to update the interpolation coefficients only every N^{th} samples. Table IV shows the computation time of the algorithm using approximated 1st order all-pass IIR interpolation, where the coefficients are updated every 1, 10, 50, and 100 samples. We found that updating the interpolation coefficients every 50 samples was a good compromise between the resulting added noise and the required computation efficiency.

TABLE IV – PROCESSING TIMES WITH DIFFERENT COEFFICIENT UPDATE RATES

Coefficients Update Rate	**Total Processing Time***
Every sample	6.9 μs
Every 10 samples	4.1 μs
Every 50 samples	3.7 μs
Every 100 samples	3.6 μs

* with $N = 8$, a sinusoidal tone modulator, and approximated 1st order all-pass interpolation.

Finally, we can make a compromise in sound quality to further reduce the algorithm's total processing time by modulating only few delay lines of the complete network. However, the benefits of the modulation will be decreased almost proportionally. Table V shows the total processing time of the algorithm using 8, 4, and 0 modulated delay lines, out of 8 delay lines.

TABLE V – PROCESSING TIMES WITH A VARIABLE RATIO OF MODULATED DELAY LINES

Number of modulated delay lines	**Total Processing Time***
8 modulated delay lines out of 8	3.7 μs
4 modulated delay lines out of 8	2.7 μs
0 modulated delay lines out of 8	1.9 μs

* with $N = 8$, a sinusoidal tone modulator, and approximated 1st order all-pass interpolation.

It is interesting to note that the processing time of the algorithm where all 8 delay lines are modulated is equivalent to the processing time of an algorithm that would use 16 non-modulated delay lines. However, the important result is that the required computation time of the algorithm where 4 out of 8 delay lines are modulated (2.7 μs) is a little bit less than an algorithm using 12 non-modulated delay lines (2.8 μs). Using the final algorithm, we have thus gained 60% reduction in memory consumption and about 4% reduction in processing time. In the next section, we will present results of the listening test in which the sound quality of these last two algorithms was compared.

5.6 Listening Test

A listening test was conducted to determine if a reverberation algorithm where 4 out of 8 delay lines are modulated (that we will refer to as algorithm B) would sound as good or better than our reference – an algorithm using 12 non-modulated delay lines (algorithm A). Algorithm B was found to be slightly faster than the reference (section 5.5), while using 60% of the memory required by algorithm A (section 5.1). We would thus like to prove that an algorithm using modulation can achieve a reverberation quality that is equivalent (or even better) than a non-modulating algorithm containing more delay lines.

5.6.1 Procedure

To test different aspects of the algorithms, four different studio recordings were used in the test. The first audio sample used is an acoustic guitar recording that sounds natural and contains a wide range of frequencies. The second recording is a solo male vocal. The third audio sample is an alto saxophone playing a jazz solo with arpeggios. The last recording is a full drum set playing a rock beat ending with a snare hit. All these instruments were recorded digitally at 44.1 kHz where microphones were positioned close to the instruments. The audio samples (without reverberation) were about 3 to 9 seconds long.

The two reverberation types were then added to each sample. Our purpose was to use the artificial reverberation as it would be used in a typical professional studio session. Reverberation parameters such as reverberation time and reverberation level were chosen differently for each instrument in order to produce a natural ambiance. Of course, for a given instrument, both reverberation types were created using the same parameters. Early reflections were also added to both algorithms to make the reverberation more realistic.

A CD was then compiled with 10 tracks per instrument, where each track contained three audio segments. The first segment used reverberation type A, the second one used type B, and the third one used type X; where X was randomly chosen to be either A or B.

The first part of the test procedure, a typical ABX listening test, was used to determine if the subjects could tell the difference between reverberation type A and B. We asked the subjects to listen to the 40 tracks and to identify in each track which reverberation type was the third audio segment. In other words, they had 10 trials per instrument to identify reverberation type X. Once this first part was completed, we proceeded to the second part of the test procedure. The objective of the second part was to find which reverberation type the listeners preferred. For each of the instruments, listeners were asked to write few words about the reverberation type they preferred (A or B) and why.

The CD was played on a portable Sony CD player, using Sony MDR-V600 studio headphones, in a quiet room. A total of 20 subjects where asked to take the listening test. All of them were university students (both graduate and undergraduate) having a musical background and some studio experience. They were allowed to listen to the audio samples as many times as they wanted.

5.6.2 *Results Significance*

To determine the significance of the test results, we had to determine if a specific score (for example, 8 out of 10 correct identifications) was due to an audible difference as opposed to chance. This requires choosing between two hypotheses. The first one, the null hypothesis (H_0), holds that the score was due to chance. The other hypothesis (H_1) holds that the score was due to an audible difference. For any score, statistics supplies the probability that it was due to chance, termed the significance level α. If the significance level α of a score is low enough, we conclude that the result was not the result of chance (we reject H_0), and infer that it was due to an audible difference (we accept H_1). The threshold below which we consider that α is low enough is called the criterion of significance α'. It is determined using common sense, and a value of α' = 0.05 is the most commonly used value.

When we make the decision of accepting or rejecting H_0, there are two kinds of error risks. The first one, type 1 error, is also labeled α' and is the risk of rejecting H_0 (because a listener had a score of 8/10 or more, for example) when it was actually true. The second one, type 2 error, is labeled β and is the risk of accepting hypothesis H_0 (because a listener had a score of 7/10 or less, for example), when H_0 in fact was true.

The probability that *c* correct identifications out of *n* total trials are due to chance is given by a binomial distribution of parameter *n* and *p* [2]:

$$P(c) = \binom{n}{c} \cdot p^n \qquad (5.6)$$

where $\binom{n}{c} = \frac{n!}{c!(n-c)!}$, and *p* is the probability of a correct identification due to chance alone ($p = 0.5$ in ABX testing). Thus, the significance level is the probability that *c* or more correct identifications in *n* trials are due to chance:

$$\alpha = P(c) + P(c+1) + ... + P(n) \qquad (5.7)$$

Using (5.6) and (5.7) with $n = 10$ trials, we find that the significance level of a score of 8/10 is α = 0.058. The same computation for a score of 9/10 gives α = 0.011. We decided to use α' = 0.058 as our criterion of significance (corresponding to 8 correct answers out of 10), since it is the closest to the usual value α' = 0.05. That means that if a listener is able to identify the correct reverberation type 8 times or more for a given instrument, we will assume that he or she is able to tell the difference between reverberation algorithms A and B (hypothesis H_1 will be accepted).

5.6.3 Results

The results of the ABX test are shown in Table VI. For each instrument, the number of listeners that were able to tell the difference between reverberation algorithms A and B is shown. We can see that on average, 60% of the subjects were able to perceive a difference in the guitar and drum samples, while only 40% and 30% differentiated the reverberation types on the vocal and saxophone recordings, respectively. Only 3 subjects were able to identify the reverberation algorithm correctly in all 4 instruments, while 5 subjects did not identify any algorithm. We should also note that even those who where able to tell the difference between the algorithms said that the two reverberation types were really hard to differentiate.

TABLE VI – RESULTS OF THE ABX LISTENING TEST: Number of listeners that were able to find a difference.

Guitar	**Vocal**	**Saxophone**	**Drums**
12/20 (60%)	8/20 (40%)	6/20 (30%)	12/20 (60%)

In the second part of the test procedure, listeners that were able to perceive a difference in identifying the reverberation type on an instrument were then asked to tell which algorithm sounded better. The results of this second part are shown in Table VII.

TABLE VII – RESULTS OF THE SECOND PART OF THE TEST: Number of listeners that preferred reverberation type B*.

Guitar	Vocal	Saxophone	Drums
7/12 (58%)	6/8 (75%)	5/6 (83%)	4/12 (33%)

* Only listeners that could hear the difference in the corresponding instrument where asked to give their opinion.

We can see that for 3 of the 4 instruments, the listeners that were able to differentiate the reverberation algorithms preferred type B. Most listeners had different preferences depending on the source type. For example, some listeners preferred reverberation A on the drums, but reverberation B on everything else. Listeners that preferred algorithm A found that it sounded in general more natural and full, and that algorithm B sounded a bit more artificial. For the vocal and saxophone recordings especially, they almost all agreed that algorithm B had a smoother decay and that they were able to hear some beats or fluttering in the tail of reverberation A. In other words, reverberation B may sound more artificial, but it helps attenuate the resonant tones present in the tail of reverberation A. That explains the fact that listeners preferred reverberation A on the drums, since reverberation A had no audible resonant frequencies in its decay for this specific recording. Interestingly, only one of the 20 listeners heard a slight pitch variation in the tail of algorithm B (associated to an excessive modulation depth) in one of the instruments.

The fact that the modulated algorithm (type B) was judged to sound generally better than the reference algorithm may seem surprising, but we have to keep in mind that the reference algorithm is not perfect. Even an algorithm using 12 delay lines (or even more) will not provide a flat frequency response. That is why the use of modulation is so powerful in reducing the audible effects of peaks in the frequency response of an artificial reverberation algorithm.

6 CONCLUSION

This thesis proposed the use of modulation to reduce the memory and processing requirements of artificial reverberation algorithms. Several modulation sources have been evaluated and the sinusoidal tone generator has been retained as one of the most efficient and was found appropriate for use as a modulator. Values of parameters such as modulation depth and rate have been suggested to achieve a good sounding reverberation, although they could be set differently according to the type of input. Different interpolation methods have also been reviewed. The modified maximally flat group delay design method to design an all-pass IIR interpolator has been retained as the most efficient and sonically transparent interpolation type. Several implementation methods and optimizations were then proposed to make the modulated algorithm more efficient. This led to the design of a reverberator using 4 modulated delay lines out of 8. This time-varying design performs slightly faster and requires only 7k words; about 60 % of the memory of the conventional design using 12 non-modulated delay lines. A listening test proved that this design sounded better than the conventional design on three recordings out of four.

Improvements could still be made to the design by trying other modulation types or more efficient random number generator modules such as shift register RNGs. Coefficients of the feedback matrix could also be modulated, instead of (or in addition to) the feedback delay lengths. The output matrix could also be modulated to change in real-time the proportion of each delay line in the output.

A typical implementation of this time-variant algorithm would include additional early reflections during the first 70 ms of reverberation. On a 16-bit gaming platform with a sampling rate of 48 kHz, the complete algorithm would take about 20 kBytes of RAM. As we already mentioned, the built-in reverberation algorithm on a PlayStation I takes up to 100 kBytes of RAM. Using our time-varying algorithm would thus result in a maximum saving of about 80 kBytes of memory. This represent a gain of almost 8% of the total RAM (1 Mbytes) allocated for all audio files and effects!

Other applications with limited memory resources would benefit as well from the use of modulation in their reverberation algorithms. For example, most hardware synthesizers and sound cards [6] with built-in digital signal processor (DSP) have limited RAM to execute reverberation algorithms. In fact, most inexpensive DSP only have an internal RAM of 1 to 10 kBytes. Some virtual reality applications [18] [19] [22] [37], multi-media applications [20] [21], and computer music applications [32] are also extremely demanding in term of performance and have very limited memory resources. Finally, we should mention again that high-end reverberation units could benefit as well from the use of modulation to enhance the quality of their algorithms.

7 REFERENCES

[1] F. A. Beltrán, J. R. Beltrán, J. Arregui, B. Gracia, "A Real-Time DSP-based Reverberation System with Computer Interface," in *Proc. 1st COST-G6 Workshop on Digital Audio Effects (DAFX98)* (Barcelona, Spain), 1998. http://www.iua.upf.es/dafx98/papers/.

[2] H. Burstein, "Approximation Formulas for Error Risk and Sample Size in ABX testing," *J. Audio Eng. Soc.*, vol. 36, pp. 879-883, Nov. 1988.

[3] J. Dattorro, "Effect Design Part 1: Reverberator and Other Filters," *J. Audio Eng. Soc.*, vol. 45, pp. 660-684, Sept. 1997.

[4] J. Dattorro, "Effect Design Part 2: Delay-Line Modulation and Chorus," *J. Audio Eng. Soc.*, vol. 45, pp. 764-788, Oct. 1997.

[5] B. Gardner, "A realtime multichannel room simulator*,"* *J. Acoust. Soc. Am.*, vol. 92 (4 (A)):2395, 1992. http://alindsay.www.media.mit.edu/papers.html.

[6] W. G. Gardner, "Reverb – a reverberator design tool for Audiomedia," in *Proc. Int. Comp. Music Conf.* (San Jose, CA), 1992. http://alindsay.www.media.mit.edu/papers.html.

[7] W. G. Gardner, *The virtual acoustic room*. Master's thesis, MIT Media Lab, 1992. http://alindsay.www.media.mit.edu/papers.html.

[8] W. G. Gardner, D. Griesinger, "Reverberation Level Matching Experiments," in *Proc. Of the Sabine Centennial Symposium* (Cambridge, MA), pp. 263-266, 1994. Acoustical Society of America.

[9] W. G. Gardner, "Efficient Convolution without Input-Output Delay," *J. Audio Eng. Soc.*, vol. 43, pp. 127-136, March 1995.

[10] W. G. Gardner, "Reverberation Algorithms," in *Applications of DSP to Audio and Acoustics*, ed. M. Kahrs, K. Brandenburg, Kluwer Academic Publishers, pp. 85-131, 1998. ISBN number: 0-7923-8130-0

[11] M. A. Gerzon, "Unitary (energy preserving) multichannel networks with feedback," *Electronic Letters*, vol. 12, no. 11, pp. 278-279, 1976.

[12] D. Griesinger, "Practical processors and programs for digital reverberation," in *Proc. Audio Eng. Soc. 7th Int. Conf.,* pp. 187-195, 1989.

[13] D. Griesinger, “Improving Room Acoustics Through Time-Variant Synthetic Reverberation,” in *Proc. 90th Conv. Audio Eng. Soc.*, Feb. 1991, preprint 3014.

[14] D. Griesinger, “How Loud is my Reverberation?,” in *Proc. Audio Eng. Soc. 98th Conv.*, 1995, preprint 3943.

[15] T. Hidaka, L. L. Beranek, and T. Okano, “Interaural cross-correlation, lateral fraction, and low- and high-frequency sound levels as measures of acoustical quality in concert halls,” *J. Acoust. Soc. Am.*, vol. 98, no. 2, pp. 988-1007-65, August 1995.

[16] J.-M. Jot and A. Chaigne, “Digital delay networks for designing artificial reverberators,” in *Proc. 90th Conv. Audio Eng. Soc.*, Feb. 1991, preprint 3030.

[17] J.-M. Jot, “An analysis/synthesis approach to real-time artificial reverberation,” in *Proc. IEEE Int. Conf. Acoust., Speech and Signal Proc, vol. 2*, pp 221-224, 1992.

[18] J.-M. Jot, O. Warusfel, “Le spatialisateur (The spatializer, in French),” *Le son & l'espace. Rencontres Musicales Pluridisciplinaires Informatique et Musique. GRAME. Musiques en Scène 1995* (Lyon), March 1995. http://mediatheque.ircam.fr/articles/index-e.html.

[19] J.-M. Jot, O. Warusfel, “A Real-Time Spatial Sound Processor for Music and Virtual Reality Applications,” in *Proc. Int Computer Music Conf.*, Sept. 1995. http://mediatheque.ircam.fr/articles/index-e.html.

[20] J.-M. Jot, “Synthesizing Three-Dimensional Sound Scenes in Audio or Multimedia Production and Interactive Human-Computer Interfaces,” in *Proc. 5th International Conference: Interface to Real & Virtual Worlds*, May 1996. http://mediatheque.ircam.fr/articles/index-e.html.

[21] J.-M. Jot, O. Warusfel, “Techniques, algorithmes et modèles de representation pour la spatialisation des sons appliquée aux services multimédia (Techniques, algorithms, and representation models for sounds spatialization applied to multimedia, in French),” in *Proc. CORESA97*, March 1997. http://mediatheque.ircam.fr/articles/index-e.html.

[22] J.-M. Jot, “Efficient Models for Reverberation and Distance Rendering in Computer Music and Virtual Reality,” *Int. Computer Music Conf.* (Thessaloniki, Greece), Sept. 1997. http://mediatheque.ircam.fr/articles/index-e.html.

[23] D. E. Knuth, *The Art of Computer Programming: Volume 2 / Seminumerical Algorithms.* Second Edition. Reading, MA: Addison-Wesley Publishing Company, 1969.

[24] H. Kuttruff, *Room Acoustics.* Elsevier Science Publishing Company (New York, NY), 1991.

[25] T. I. Laakso, V. Välimäki, M. Karjalainen, and U. K. Laine, "Splitting the Unit Delay – Tools for Fractional Delay Filter Design," *IEEE Signal Processing Mag.*, vol. 13, pp. 30-60, Jan. 1996.

[26] J. Martin, D. Van Maercke, and J-P. Vian, "Binaural simulation of concert halls: A new approach for the binaural reverberation process," *J. Acoust. Soc. Am.*, vol. 94, no. 6, pp. 3255-3264, December 1993.

[27] J. A. Moorer, "About This Reverberation Business," *Computer Music Journal*, vol. 3, no. 2, pp. 13-28, 1979.

[28] P. Murphy, A. Krukowski and A. Tarczynski, "An efficient fractional sample delayer for digital beam steering," in *Proc. Int. Conf. on Acoust., Speech and Signal Processing* (Munich, Germany), April 1997. http://www.cmsa.westminster.ac.uk/~artur/papers/Paper14/Paper14.htm.

[29] E. Piirilä, T. Lokki, V. Välimäki, "Digital Signal Processing Techniques for Non-exponentially Decaying Reverberation," in *Proc. 1st COST-G6 Workshop on Digital Audio Effects (DAFX98)* (Barcelona, Spain), 1998. http://www.iua.upf.es/dafx98/papers/.

[30] Roads, C., *The computer music tutorial*. Cambridge, MA: The MIT Press, 1994.

[31] D. Rocchesso, and J. O. Smith, "Circulant and Elliptic Feedback Delay Networks for Artificial Reverberation," *IEEE Transactions on Speech and Audio*, vol. 5, no. 1, pp. 51-60, Jan. 1996. http://ccrma-www.stanford.edu/~jos/index.html.

[32] D. Rocchesso, "The Ball within the Box: A Sound-processing Metaphor," *Computer Music Journal*, vol. 19, no.4, pp. 47-57, winter 1997.

[33] W. C. Sabine, "Reverberation," in Lindsay, R. B., editor, *Acoustics: Historical and Philosophical Development*, Stroudsburg, PA: Dowden, Hutchinson, and Ross, 1972. Originally published in 1900.

[34] L. Savioja, T. J. Rinne, T. Takala, "Simulation of Room Acoustics with a 3-D Finite Difference Mesh," in *Proc. Int. Computer Music Conf. (ICMC'94)* (Århus, Denmark), pp. 463-466, Sept. 1994. http://www.tml.hut.fi/~las/publications.html.

[35] L. Savioja, J. Backman, A. Järvinen, T. Takala, "Waveguide Mesh Method for Low-Frequency Simulation of Room Acoustics," in *Proc. Int. Congress on Acoustic (ICA'95)* (Trondheim, Norway), pp. 637-641, June 1995. http://www.tml.hut.fi/~las/publications.html.

[36] L. Savioja, M. Karjalainen, and T. Takala, "DSP Formulation of a Finite Difference Method for Room Acoustics Simulation," In *NORSIG'96 IEEE Nordic Signal Processing Symposium* (Espoo, Finland), pp. 455-458, Sept. 1996. http://www.tml.hut.fi/~las/publications.html.

[37] L. Savioja, J. Huopaniemi, T. Lokki, and R. Väänänen, "Creating Interactive Virtual Acoustic Environments," *J. Audio Eng. Soc.*, vol. 47, no. 9, pp. 675-705, Sept. 1998.

[38] M. R. Schroeder and B. F. Logan, "'Colorless' Artificial Reverberation," *J. Audio Eng. Soc.*, vol. 9, no. 3, pp. 192-197, July 1961.

[39] M. R. Schroeder, "Natural Sounding Artificial Reverberation," *J. Audio Eng. Soc.*, vol. 10, no. 3, pp. 219-223, July 1962.

[40] M. R. Schroeder, "New method of measuring reverberation time," *J. Acoust. Soc. Am.*, vol. 37, pp. 409-412, 1965.

[41] M. R. Schroeder, D. Gottlob, and K. F. Siebrasse, "Comparative study of European concert halls: correlation of subjective preference with geometric and acoustic parameters," *J. Acoust. Soc. Am.*, vol. 56, no. 4, pp. 1195-1201, Oct. 1974.

[42] M. R. Schroeder, "Statistical Parameters of the Frequency Response Curves of Large Rooms," *J. Audio Eng. Soc.*, vol. 35, no. 5, pp. 299-306, May 1987.

[43] Sonic Foundry, *Sound Forge 4.0 plug-in manual: Acoustics modeler Direct X audio plug-in*. Madison, WI:, 1997

[44] J. O. Smith, "A new approach to digital reverberation using closed waveguide networks," in *Proc. 1985 Int. Computer Music Conf.* (Vancouver, Canada), pp. 47-53, 1985. Computer Music Association. Also available in [45].

[45] J. O. Smith, *Music applications of digital waveguides*. Technical Report STAN-M-39, CCRMA, Music Dept., Stanford University, 1987. A compendium containing four related papers and presentation overheads on digital waveguide reverberation, synthesis, and filtering. CCRMA technical reports can be ordered by calling (415) 723-4971 or by sending an email request to info@ccrma.standford.edu.

[46] J. O. Smith, and D. Rocchesso, "Connections between Feedback Delay Networks and Waveguide Networks for Digital Reverberation," in *Proc. 1994 Int. Computer Music Conf.* (Århus, Denmark), pp. 376-377, 1994. Computer Music Association.

[47] J. Stautner, and M. Puckette, "Designing Multi-Channel Reverberators," *Computer Music Journal*, vol. 6, no. 1, pp. 52-65, Spring 1982.

[48] J. R. VandeKieft, *Computational Improvements to Linear Convolution With Multirate Filtering Methods*. Master's thesis, University of Miami, 1998.

[49] B. Vercoe and Miller Puckette, "Synthetic Spaces – Artificial Acoustic Ambience from Active Boundary Computation," unpublished NSF proposal, 1985. Available from Music and Cognition office at MIT Media Lab.

[50] U. Zölzer, N. Fleige, M. Schonle, and M. Schusdziara, "Multirate Digital Reverberation System," in *Proc. Audio Eng. Conv*., 1990, preprint 2968.

[51] U. Zölzer, *Digital Audio Signal Processing.* John Wiley & Sons (New York, NY), 1997.

8 APPENDIX :
C++ Code for a VST Plug-In Implementation

Code Summary:

AFdnReverbMain.cpp:	Main() of the reverberation plug-in
AFdnReverb.hpp:	Header of the reverberation plug-in
AFdnReverb.cpp:	Implementation of the reverberation plug-in
Modules.hpp:	Library of useful C++ modules such as delay lines, filters, oscillators, etc
Modules.cpp:	Implementation of useful C++ modules such as delay lines, filters, oscillators, etc
PlgMem.hpp:	Library of memory management utilities
PlgMem.cpp:	Implementation of memory management utilities

```
/*---------------------------------------------------------------------------

   File:        AFdnReverbMain.cpp

   Description: main() of the time-varyiant FDN reverberation plug-in

   Author:      Jasmin Frenette
   Date:        11/27/2000

---------------------------------------------------------------------------*/

#include "AFdnReverb.hpp"

bool oome = false;

#if MAC
#pragma export on
#endif

// prototype of the export function main
#if BEOS
#define main main_plugin
extern "C" __declspec(dllexport) AEffect *main_plugin (audioMasterCallback
                                                       audioMaster);

#else
AEffect *main (audioMasterCallback audioMaster);
#endif

AEffect *main (audioMasterCallback audioMaster)
{
   // get vst version
   if (!audioMaster (0, audioMasterVersion, 0, 0, 0, 0))
      return 0;  // old version

   AudioEffect* effect = new AFdnReverb (audioMaster);
   if (!effect)
      return 0;
   if (oome)
   {
      delete effect;
      return 0;
   }
   return effect->getAeffect ();
}

#if MAC
#pragma export off
#endif

#if WIN32
#include <windows.h>
void* hInstance;
BOOL WINAPI DllMain (HINSTANCE hInst, DWORD dwReason, LPVOID lpvReserved)
{
   hInstance = hInst;
   return 1;
}
#endif
```

```
/*------------------------------------------------------------------------------

   File:        AFdnReverb.hpp

   Description: Header of the time-variant FDN reverberation plug-in

   Author:      Jasmin Frenette
   Date:        11/27/2000

------------------------------------------------------------------------------*/

#ifndef _AFDNREVERB_HPP
#define _AFDNREVERB_HPP

#include <windows.h>
#include <stdio.h>
#include <time.h>

#include "audioeffectx.h"
#include <string.h>

#include "modules.hpp"


// FILE_OUTPUT generates a raw file (with stereo 16-bit PCM signed integers
//             stored in "little endian" format) containing the output of the
//             algorithm (turn this option OFF while measuring the performance
//             of the algorithm)
//#define FILE_OUTPUT

// QUERY_PERFORMANCE generates a text file containing the algorithm processing
//                   times for every audio buffer
//#define QUERY_PERFORMANCE

// NB_DELAYS is the number of delay lines (has to be 1, 8, 12, or 16)
#define NB_DELAYS            8

// NB_MOD_DELAYS is the number of modulated delay lines (from 1 to NB_DELAYS)
#define NB_MOD_DELAYS      4

// USE_VARIABLE_RATE uses variable interpolation coefficient update rates
#define USE_VARIABLE_RATES

// USE_XXX_MODULATION choses the kind of modulation source
#define USE_OSC_MODULATION
//#define USE_RNG_MODULATION


#define EARLY_REF_FIR_SIZE   0.080   // FIR max delay (in seconds)
#define NB_TAPS            4      // Number of early reflection taps
#define NB_FIXED_DELAYS      (NB_MOD_DELAYS - NB_DELAYS)


enum
{
   kOut,

   kNumParams
};

class AFdnReverb;

class AFdnReverbProgram
```

```
{
friend class AFdnReverb;
public:
   AFdnReverbProgram();
   ~AFdnReverbProgram() {}
private:
   float fOut;
   char name[24];
};

class AFdnReverb : public AudioEffectX
{
public:
   AFdnReverb(audioMasterCallback audioMaster);
   ~AFdnReverb();

   virtual void process(float **inputs, float **outputs, long sampleframes);
   virtual void setProgram(long program);
   virtual void setProgramName(char *name);
   virtual void getProgramName(char *name);
   virtual void setParameter(long index, float value);
   virtual float getParameter(long index);
   virtual void getParameterLabel(long index, char *label);
   virtual void getParameterDisplay(long index, char *text);
   virtual void getParameterName(long index, char *text);
   virtual float getVu();
   virtual void suspend();

private:
   float fOut;
   float vu;

   AFdnReverbProgram *programs;

   //============================
   // Parameters
   //============================
   float   fDCRevTime;                  // Reverb time at 0 Hz (in seconds)
   float   fPIRevTime;                  // Reverb time at Fs/2 Hz (in seconds)
   long   lPreDelay;                  // Pre delay (in samples)
   long   lTau[NB_DELAYS];            // Delay lines time constants (in samples)
#if (NB_MOD_DELAYS>0)
      long   lModDepth[NB_MOD_DELAYS];   // Delay lines mod depths (in samples)
      long   lModRate[NB_MOD_DELAYS];      // Delay lines mod rates (in samples)
   #ifdef USE_OSC_MODULATION
      float   fModFreq[NB_MOD_DELAYS];      // Delay lines mod frequency (in
Hertz)
      float   fModPhase[NB_MOD_DELAYS];   // Delay lines mod phase (in Degrees)
   #elif defined USE_RNG_MODULATION
      unsigned long   ulRngA[NB_MOD_DELAYS];      // Delay lines rngs' a params
      unsigned long   ulRngC[NB_MOD_DELAYS];      // Delay lines rngs' c params
      unsigned long   ulRngSeed[NB_MOD_DELAYS];   // Delay lines rngs' seeds
      long   lRngRate[NB_MOD_DELAYS];            // Delay lines rngs' rates
   #endif
#endif
   float   fA[NB_DELAYS][NB_DELAYS];   // Feedback matrix
   float   fFeedConstant;            // Feedback constant: -2/NB_DELAYS
   float   fcL[NB_DELAYS];            // Left delay lines' output gains
   float   fcR[NB_DELAYS];            // Right delay lines' output gains
   float   fkp[NB_DELAYS];            // IIR LPFs gain
   float   fbp[NB_DELAYS];            // IIR LPFs feedback gain
   long   lTapTimeL[NB_TAPS];         // Left Early ref tap times (in samples)
   long   lTapTimeR[NB_TAPS];         // Right Early ref tap times (in samples)
   float   fTapGainL[NB_TAPS];         // Left Early ref tap gains
   float   fTapGainR[NB_TAPS];         // Right Early ref tap gains

   // Modules
#if (NB_MOD_DELAYS>0)

      // Choose the appropriate interpolation type here:
      MSVRAP1ModDelayLine modDelayLine[NB_MOD_DELAYS];

   #ifdef USE_OSC_MODULATION
      MSinOsc mod[NB_MOD_DELAYS];
   #elif defined USE_RNG_MODULATION
      MRNG rng[NB_MOD_DELAYS];
      MLPF2b rngLpf1[NB_MOD_DELAYS];
      MLPF2b rngLpf2[NB_MOD_DELAYS];
      MLPFMod mod[NB_MOD_DELAYS];
   #endif
#endif

   // We only use the last (NB_DELAYS - NB_MOD_DELAYS) of the following array:
   MDelayLine delayLine[NB_DELAYS];
   MLPF1a dampingLPF[NB_DELAYS];
   MDelayLine earlyRefFIR;

#ifdef FILE_OUTPUT
   int fh;
#endif

#ifdef QUERY_PERFORMANCE
   LARGE_INTEGER      timeFreq;
   LARGE_INTEGER      timeBegin;
   LARGE_INTEGER      timeEnd;
   double            timeDelta, timeSum;
   long               counter;
   FILE*               pf;
#endif
};

#endif // _AFDNREVERB_HPP
```

```
/*--------------------------------------------------------------------------------

   File:        AFdnReverb.cpp

   Description: Implementation of the time-varying FDN reverberation plug-in

   Author:      Jasmin Frenette
   Date:        11/27/2000

--------------------------------------------------------------------------------*/

#include <windows.h>
#include <io.h>
#include <FCNTL.H>
#include <SYS\STAT.H>
#include <stdio.h>
#include <math.h>
#include <string.h>

#include "AFdnReverb.hpp"
#include "AEffEditor.hpp"

//--------------------------------------------------------------------------------

AFdnReverbProgram::AFdnReverbProgram ()
{
   fOut = (float) 0.5;
   strcpy (name, "Init");
}

//--------------------------------------------------------------------------------

AFdnReverb::AFdnReverb (audioMasterCallback audioMaster)
   : AudioEffectX (audioMaster, 16, kNumParams)
{
   long i, j;

   programs = new AFdnReverbProgram[numPrograms];
   fOut = vu = 0;

   //
   // Parameters Initialization
   //

   fDCRevTime = 3.00f;   // Reverberation time at 0 Hz (in seconds)
   fPIRevTime = 1.25f;   // Reverberation time at Fs/2 Hz (in seconds)
   lPreDelay = 1102;      // Pre delay (in samples)

   // delay lines time constants (in samples)
#if (NB_DELAYS == 1)
   lTau[0] = 1000;
#elif (NB_DELAYS == 4)
   lTau[0] = 601;
   lTau[1] = 691;
   lTau[2] = 773;
   lTau[3] = 839;
#elif (NB_DELAYS == 8)
   lTau[0] = 601;
   lTau[1] = 691;
   lTau[2] = 773;
   lTau[3] = 839;
   lTau[4] = 919;
   lTau[5] = 997;
   lTau[6] = 1061;
   lTau[7] = 1129;
#elif (NB_DELAYS == 12)
   lTau[0] = 601;
   lTau[1] = 691;
   lTau[2] = 773;
   lTau[3] = 839;
   lTau[4] = 919;
   lTau[5] = 997;
   lTau[6] = 1061;
   lTau[7] = 1093;
   lTau[8] = 1129;
   lTau[9] = 1151;
   lTau[10] = 1171;
   lTau[11] = 1187;
#elif (NB_DELAYS == 16)
   lTau[0] = 919;
   lTau[1] = 997;
   lTau[2] = 1061;
   lTau[3] = 1093;
   lTau[4] = 1129;
   lTau[5] = 1151;
   lTau[6] = 1171;
   lTau[7] = 1187;
   lTau[8] = 1213;
   lTau[9] = 1237;
   lTau[10] = 1259;
   lTau[11] = 1283;
   lTau[12] = 1303;
   lTau[13] = 1319;
   lTau[14] = 1327;
   lTau[15] = 1361;
#endif

#if (NB_MOD_DELAYS>0)
   // delay lines modulation depth (in samples)
   for(i=0;i<NB_MOD_DELAYS;i++){
      lModDepth[i] = 6;
   }

   // Delay lines interpolators' update rates
   for(i=0;i<NB_MOD_DELAYS;i++){
      lModRate[i] = 50;   // in samples
   }

   #ifdef USE_OSC_MODULATION
      // delay lines modulation frequency (in Hertz)
      for(i=0;i<NB_MOD_DELAYS;i++){
         fModFreq[i] = (float)(2);
      }

      // delay lines modulation phase (in Degrees)
      for(i=0;i<NB_MOD_DELAYS;i++){
         fModPhase[i] = (float)(i*45);
      }

   #elif defined USE_RNG_MODULATION
      // Delay lines RNG's "a" parameters
      for(i=0;i<NB_MOD_DELAYS;i++){
         ulRngA[i] = 1664525;
      }

      // Delay lines RNG's "c" parameters
      for(i=0;i<NB_MOD_DELAYS;i++){
         ulRngC[i] = 32767;
      }

      // Delay lines RNG's seeds
      for(i=0;i<NB_MOD_DELAYS;i++){
         ulRngSeed[i] = i;
      }

      // Delay lines rngs' refresh rates
      for(i=0;i<NB_MOD_DELAYS;i++){
         lRngRate[i] = 441;   // in samples
      }
   #endif
#endif

   // Feedback matrix
   float fTwoDivByNbDelays = (float)(2.0/NB_DELAYS); // Coeff. used for speed
   for(i=0;i<NB_DELAYS;i++){
      for(j=0;j<NB_DELAYS;j++){
         if((j==(i+1))||((i==(NB_DELAYS-1))&&(j==0)))
            fA[i][j] = 1;
         else
            fA[i][j] = 0;
         fA[i][j] -= fTwoDivByNbDelays;
      }
   }
   fFeedConstant = (float)(-2.0)/NB_DELAYS;

   // Early reflexions tap times
   lTapTimeL[0] = lPreDelay + (long)(0.0090 * getSampleRate());
   lTapTimeL[1] = lPreDelay + (long)(0.0118 * getSampleRate());
   lTapTimeL[2] = lPreDelay + (long)(0.0213 * getSampleRate());
   lTapTimeL[3] = lPreDelay + (long)(0.0205 * getSampleRate());
   lTapTimeR[0] = lPreDelay + (long)(0.0145 * getSampleRate());
   lTapTimeR[1] = lPreDelay + (long)(0.0100 * getSampleRate());
   lTapTimeR[2] = lPreDelay + (long)(0.0205 * getSampleRate());
   lTapTimeR[3] = lPreDelay + (long)(0.0230 * getSampleRate());

   // Early reflexions tap gains
   fTapGainL[0] = (float)  1.35;
   fTapGainL[1] = (float) -1.15;
   fTapGainL[2] = (float) -1.14;
   fTapGainL[3] = (float)  1.15;
   fTapGainR[0] = (float)  1.35;
   fTapGainR[1] = (float) -1.16;
   fTapGainR[2] = (float) -1.00;
   fTapGainR[3] = (float)  1.14;

   // Set early ref FIR max delay
   earlyRefFIR.SetMaxDelay((long)(EARLY_REF_FIR_SIZE*getSampleRate()));

   float T = 1/getSampleRate();   // Sampling period
   float alpha = fPIRevTime/fDCRevTime;

   for(i=0;i<NB_DELAYS;i++){
      // IIR LPFs gain
      fkp[i] = (float)pow(10,-3*lTau[i]*T/fDCRevTime);

      // IIR LPFs feedback gain
      fbp[i] = (float)(20*log10(fkp[i])*log(10)/80*(1 - 1/pow(alpha,2)));
      dampingLPF[i].SetCoeffs(fkp[i]*(1-fbp[i]),(-1)*fbp[i]);
   }
```

```
#if (NB_MOD_DELAYS>0)
   for(i=0;i<NB_MOD_DELAYS;i++){
   #ifdef USE_OSC_MODULATION
      // Modulators
      #ifdef USE_VARIABLE_RATES
         mod[i].SetFreq(fModFreq[i]*lModRate[i], getSampleRate(),fModPhase[i]);
      #else
         mod[i].SetFreq(fModFreq[i], getSampleRate(), fModPhase[i]);
      #endif
   #elif defined USE_RNG_MODULATION
      rng[i].SetParams(ulRngA[i], ulRngC[i], ulRngSeed[i]);
      // Coeffs for a 2nd order Chebichev Butterworth LPF
      rngLpf1[i].SetCoeffs(0.0000003258000341709795, 2, -1.999758737383959,
                           0.9997587663202134);
      rngLpf2[i].SetCoeffs(0.0000003258000341709795, 2, -1.99989998432827,
                           0.999900070678988);
      mod[i].SetRefreshRate(lRngRate[i]);
      mod[i].SetModulator(&rng[i]);
      mod[i].SetLPF1(&rngLpf1[i]);
      mod[i].SetLPF2(&rngLpf2[i]);
   #endif

      // Set delay lines's parameters
      modDelayLine[i].SetNomDelay(lTau[i]);
      modDelayLine[i].SetModDepth(lModDepth[i]);
   #ifdef USE_VARIABLE_RATES
      modDelayLine[i].SetModRate(lModRate[i]);
   #endif
      modDelayLine[i].SetModulator(&mod[i]);
   }
#endif

   for(i=NB_MOD_DELAYS;i<NB_DELAYS;i++){
      // Set delay lines's length
      delayLine[i].SetMaxDelay(lTau[i]);
   }

   for(i=0;i<NB_DELAYS;i++){
      // Left delay lines' output gains

      /*
      fcL[i] = (float)lTau[i]/lTau[NB_DELAYS-1];
      */
      fcL[i] = 1;  // No weighting for fcL if A is not the identity matrix

      if((i%2)!=0)
         fcL[i] *= -1;
      if((i==1)||(i==2)||(i==5)||(i==6)||(i==9)||(i==10)||(i==13)||(i==14))
         fcR[i] = -1 * fcL[i];
      else
         fcR[i] = fcL[i];
   }

#ifdef FILE_OUTPUT
   while((fh = _open("D:\\output.raw", _O_TRUNC | _O_WRONLY | _O_CREAT
      | _O_BINARY, _S_IWRITE)) == -1 ){
      MessageBox(NULL,"FDN Reverb","Unable to open output file",MB_OK);
      DebugBreak();
   }
   _close(fh);
#endif
```

```
#ifdef QUERY_PERFORMANCE
   timeSum = 0;
   counter = 0;
   pf = fopen("D:\\performance.txt","w");
   fclose(pf);
#endif

   if (programs)
      setProgram (0);

   setNumInputs (1);
   setNumOutputs (2);
   hasVu ();
   setUniqueID ('FDNR');

   suspend ();      // flush buffer
}

//-------------------------------------------------------------------------------
AFdnReverb::~AFdnReverb ()
{
   if (programs)
      delete[] programs;
}

//-------------------------------------------------------------------------------
void AFdnReverb::setProgram (long program)
{
   AFdnReverbProgram * ap = &programs[program];

   curProgram = program;
   setParameter (kOut, ap->fOut);
}

//-------------------------------------------------------------------------------
void AFdnReverb::setProgramName (char *name)
{
   strcpy (programs[curProgram].name, name);
}

//-------------------------------------------------------------------------------
void AFdnReverb::getProgramName (char *name)
{
   if (!strcmp (programs[curProgram].name, "Init"))
      sprintf (name, "%s %d", programs[curProgram].name, curProgram + 1);
   else
      strcpy (name, programs[curProgram].name);
}

//-------------------------------------------------------------------------------
void AFdnReverb::suspend ()
{
   long i;

   earlyRefFIR.Suspend();
#if (NB_MOD_DELAYS>0)
   for(i=0;i<NB_MOD_DELAYS;i++){
      modDelayLine[i].Suspend();
   }
#endif
   for(i=NB_MOD_DELAYS;i<NB_DELAYS;i++){
      delayLine[i].Suspend();
   }
   for(i=0;i<NB_DELAYS;i++){
      dampingLPF[i].Suspend();
   }
}

//-----------------------------------------------------------------------------
float AFdnReverb::getVu ()
{
   float cvu = vu;

   vu = 0;
   return cvu;
}

//-----------------------------------------------------------------------------
void AFdnReverb::setParameter (long index, float value)
{
   AFdnReverbProgram * ap = &programs[curProgram];

   switch (index)
   {
      case kOut :            fOut = ap->fOut = value; break;
   }
   if (editor)
      editor->postUpdate ();
}

//-----------------------------------------------------------------------------
float AFdnReverb::getParameter (long index)
{
   float v = 0;

   switch (index)
   {
      case kOut :            v = fOut; break;
   }
   return v;
}

//-----------------------------------------------------------------------------
void AFdnReverb::getParameterName (long index, char *label)
{
   switch (index)
   {
      case kOut :            strcpy (label, " Volume "); break;
   }
}

//-----------------------------------------------------------------------------
void AFdnReverb::getParameterDisplay (long index, char *text)
{
   switch (index)
   {
      case kOut :            dB2string (fOut, text); break;
   }
}

//-----------------------------------------------------------------------------
void AFdnReverb::getParameterLabel (long index, char *label)
{
   switch (index)
   {
      case kOut :            strcpy (label, "   dB   ");   break;
   }
}

//--------------------------------------------------------------------------------
void AFdnReverb::process(float **inputs, float **outputs, long sampleframes)
{
   float cvu = vu;

   long p, count;
   float xn, yn1, yn2;
   float factor;
   float fQ[NB_DELAYS], fS[NB_DELAYS];

   float *in = *inputs;
   float *out1 = outputs[0];
   float *out2 = outputs[1];

#ifdef FILE_OUTPUT
   short *piFileOut = (short*)GlobalAlloc(GPTR,2*sampleframes*sizeof(short));
#endif

#ifdef QUERY_PERFORMANCE
   QueryPerformanceFrequency(&timeFreq);
   QueryPerformanceCounter(&timeBegin);
#endif

   for(count=0;count<sampleframes;count++)
   {
        xn = *in++;

#ifdef USE_1_DELAY
   #if (NB_MOD_DELAYS==1)
      yn1 = xn + modDelayLine[0].GetCurValue();
   #else
      yn1 = xn + delayLine[0].GetLast();
   #endif
      dampingLPF[0].Process(yn1, yn1);
      delayLine[0].Push(yn1);
      yn2 = yn1;
#else
      // xn *= fOut; // Optional global output level (VST real-time parameter)

      yn1 = yn2 = xn;

/*
      // Pre-Delay (optional)
      earlyRefFIR.Push(xn);
      xn = earlyRefFIR.GetAt(lPreDelay);

      // Get early reflections (optional)
      for(int i=0;i<NB_TAPS;i++){
         yn1 += fTapGainL[i]*earlyRefFIR.GetAt(lTapTimeL[i]);
         yn2 += fTapGainR[i]*earlyRefFIR.GetAt(lTapTimeR[i]);
      }

      // Tone correction filter should be implemented here (optional)
*/

      // Get the delay lines output, filter them and compute output
#if (NB_MOD_DELAYS>0)
      for(p=0;p<NB_MOD_DELAYS;p++){
         fQ[p] = modDelayLine[p].GetCurValue();
         dampingLPF[p].Process(fQ[p], fQ[p]);
         yn1 += fcL[p]*fQ[p];
         yn2 += fcR[p]*fQ[p];
      }
#endif
      for(p=NB_MOD_DELAYS;p<NB_DELAYS;p++){
         fQ[p] = delayLine[p].GetLast();
         dampingLPF[p].Process(fQ[p], fQ[p]);
         yn1 += fcL[p]*fQ[p];
         yn2 += fcR[p]*fQ[p];
      }

      // Compute feedback
      factor = 0;
      for(p=0;p<NB_DELAYS;p++){
         factor += fQ[p];
      }
      factor *= fFeedConstant;
      for(p=1;p<NB_DELAYS;p++){
         fS[p-1] = fQ[p] + factor;
      }
      fS[NB_DELAYS-1] = fQ[0] + factor;

      // Add the input and push the samples in the delay lines
#if (NB_MOD_DELAYS>0)
      for(p=0;p<NB_MOD_DELAYS;p++){
         fS[p] += xn;
         modDelayLine[p].Push(fS[p]);
      }
#endif
      for(p=NB_MOD_DELAYS;p<NB_DELAYS;p++){
         fS[p] += xn;
         delayLine[p].Push(fS[p]);
      }
#endif

#ifdef FILE_OUTPUT
      piFileOut[2*count]   = (short)(32767*yn1);
      piFileOut[2*count+1] = (short)(32767*yn2);
#endif
      (*out1++) += yn1;   // store to buss
      (*out2++) += yn2;
   }
   vu = 0; // not implemented

#ifdef FILE_OUTPUT
   fh = _open("D:\\output.raw", _O_WRONLY | _O_CREAT | _O_BINARY
           | _O_APPEND, _S_IWRITE);
   _write(fh, piFileOut, 2*sampleframes*sizeof(short));
   _close(fh);
   if(piFileOut != NULL)
      GlobalFree(piFileOut);
#endif

#ifdef QUERY_PERFORMANCE
   QueryPerformanceCounter(&timeEnd);
   timeDelta =   (double)(timeEnd.QuadPart-timeBegin.QuadPart)
           /(double)(timeFreq.QuadPart);
   timeSum += timeDelta/sampleframes;
   counter++;
   pf = fopen("D:\\performance.txt","a");
   fprintf(pf, "%2d: %2.2f us Avrg: %2.2f us\n", counter,
         (timeDelta/sampleframes*1000000),
         (timeSum/counter*1000000));
   fclose(pf);
#endif
}
```

```
/*-------------------------------------------------------------------------------

   File:        Modules.hpp

   Description: Library of useful C++ modules such as delay lines, filters
                oscillators, etc.

   Author:      Jasmin Frenette
   Date:        11/27/2000

-------------------------------------------------------------------------------*/

#ifndef _MODULES_HPP
#define _MODULES_HPP

#define PI                  3.14159265358979323846264338327 95
#define TWOPI               6.28318530717958647692528676655901
#define PI_DIV_BY_2         1.57079632679489661923132169163975
#define SCALING_FACTOR     0.0000000004656612874161594f        // 2/(2^32-1)

class Module
{
public:
   Module() {}
   ~Module() {}

   virtual void Suspend() {}
   virtual long GetMem() {return 0;}

   void UpdateMemory(long* &pMem, long newSize);
   void UpdateMemory(float* &pMem, long newSize);
   void UpdateMemory(float** &pMem, long newSize);
};


/*
//-------------------------------------------------------------------------------
// 1st order FIR Low Pass Filter Module (version a)
//-------------------------------------------------------------------------------

in -----[b0]-->0-------------> out
               ^         |
               |       [z-1]
               |         |
                --[-a1]--
*/
class MLPF1a : public Module
{
public:
   MLPF1a();
   ~MLPF1a() {}

   void SetCoeffs(float in_b0, float in_a1);
   void GetCoeffs(float* out_b0, float* out_a1);
   void Suspend(void);

   void Process(float &in, float &out){
      out = in*b0 - buffer*a1;
      buffer = out;
   }

private:
   float buffer;
   float b0, a1;         // filter coefficients
};


/*
//----------------------------------------------------------------------------
// 1st order FIR Low Pass Filter Module (version b)
//----------------------------------------------------------------------------

  in --[k]---------->0------------> out
           |       ^         |
         [z-1]     |       [z-1]
           |       |         |
            -------^--[-a1]--
*/
class MLPF1b: public Module
{
public:
   MLPF1b();
   ~MLPF1b() {}

   void SetCoeffs(float in_k, float in_a1);
   void GetCoeffs(float* out_k, float* out_a1);
   void Suspend(void);

   void Process(float &in, float &out){
      float temp;
      temp = in * k;
      out = temp + bufferX - bufferY*a1;
      bufferX = temp;
      bufferY = out;
   }

private:
   float bufferX, bufferY;
   float k, a1;         // filter coefficients
};


/*
//----------------------------------------------------------------------------
// 2nd order FIR Low Pass Filter Module
//----------------------------------------------------------------------------

  in --[k]--------->0------------> out
           |        ^         |
         [z-1]      |       [z-1]
           |        |         |
            --[b1]--^--[-a1]--
           |        |         |
         [z-1]      |       [z-1]
           |        |         |
            --------^--[-a2]--
*/
class MLPF2b: public Module
{
public:
   MLPF2b();
   ~MLPF2b() {}

   void SetCoeffs(double in_k, double in_b1, double in_a1, double in_a2);
   void GetCoeffs(double* out_k, double* out_b1, double* out_a1, double*
out_a2);
```

```
   void Suspend(void);

   void Process(double &in, double &out){
      double temp;
      temp = in * k;
      out = temp + bufferX1*b1 + bufferX2 - bufferY1*a1 - bufferY2*a2;
      bufferX2 = bufferX1;
      bufferX1 = temp;
      bufferY2 = bufferY1;
      bufferY1 = out;
   }

private:
   double k, b1, a1, a2;         // filter coefficients
   double bufferX1, bufferX2, bufferY1, bufferY2;
};


//------------------------------------------------------------------------------
// Modulator (base class)
//------------------------------------------------------------------------------
class MModulator
{
public:
   MModulator() {}
   ~MModulator() {}

   virtual float GetCurValue() = 0;
};


//------------------------------------------------------------------------------
// Sinusoidal Oscillator Module
//------------------------------------------------------------------------------
class MSinOsc : public MModulator
{
public:
   MSinOsc();
   ~MSinOsc();

   void   SetFreq(float freq, float sampleRate, float phase);
   float   GetFreq();
   void   Suspend(void);

   float   GetCurValue(void){
      double out;
      out = a * buffer1 - buffer2;

      // Two special cases to avoid instability due to
      // round-off errors.
      if(out >= 1.0) {
         out = 1.0;
         buffer2 = resetBuffer2;
      }
      else if(out <= -1.0) {
         out = -1.0;
         buffer2 = -resetBuffer2;
      }
      else buffer2 = buffer1;

      buffer1 = out;
      return (float) out;
   }
```

```
private:
   float      fFreq;         // Osc. Frequency (in Hertz)
   double   a;               // Coefficient: a=2*cos(w)
   double   buffer1;         // Osc. buffer1
   double   buffer2;         // Osc. buffer2
   double   resetBuffer2;   // Osc. reset value of buffer2
};


//-------------------------------------------------------------------------------
//  Random Number Generator Module
//  x(n+1) = (ax(n) + c) mod 2^32
//-------------------------------------------------------------------------------
class MRNG : public MModulator
{
public:
   MRNG();
   ~MRNG();

   void   SetParams(unsigned int in_a, unsigned int in_c,
                    unsigned int in_seed);
   void   GetParams(unsigned int* out_a, unsigned int* out_c,
                    unsigned int* out_seed);
   void   Suspend(void);

   float   GetCurValue(void){
      float out;

      // Generate a random number from 0 to (2^32)-1
      buffer = a * buffer + c;
      // Scale to [-1.0, 1.0]
      out = (float)buffer * SCALING_FACTOR - 1.0f;

      return out;
   }


private:
   unsigned long   a, c, seed;
   unsigned long   buffer;
};


//-------------------------------------------------------------------------------
//  Low-Pass Filtered (4rth order) Modulator
//-------------------------------------------------------------------------------
class MLPFMod : public MModulator
{
public:
   MLPFMod();
   ~MLPFMod();

   void   SetRefreshRate(long rate);
   void   SetModulator(MModulator* mod);
   void   SetLPF1(MLPF2b* lpf);
   void   SetLPF2(MLPF2b* lpf);
   long   GetRefreshRate(void);
   MModulator* GetModulator(void);
   MLPF2b* GetLPF1(void);
   MLPF2b* GetLPF2(void);

   float   GetCurValue(void);
```

```
private:
   long         lRate;   // Refresh rate (in samples)
   MModulator*   pMod;      // modulator
   MLPF2b*      pLPF1;   // 1st Low-Pass Filter
   MLPF2b*      pLPF2;   // 2nd Low-Pass Filter

   long index;
};


//------------------------------------------------------------------------------
// Delay Line Module
//------------------------------------------------------------------------------
class MDelayLine : public Module
{
public:
   MDelayLine();
   ~MDelayLine();

   void   SetMaxDelay(long delay);
   long   GetMaxDelay(void);
   void   Suspend(void);
   long   GetMem(void);

   void   Push(float &value){
      *(pfInPos) = value;
      if(++pfInPos >= pfBufferEnd)
         pfInPos -= lBufferSize;
   }

   float   GetLast(void){
      float out = *(pfOutPos);
      if(++pfOutPos >= pfBufferEnd)
         pfOutPos -= lBufferSize;
      return out;
   }

   float   GetAt(long index){
      if(index>=lBufferSize)
         DebugBreak();
      float* pfRealIndex = pfInPos - index - 1; // Minus one because input has
                                          // been incremented after the push
      if(pfRealIndex < pfBuffer)
         pfRealIndex += lBufferSize;
      return *(pfRealIndex);
   }

protected:
   // Buffer parameters
   float   *pfBuffer, *pfBufferEnd;
   long   lBufferSize;
   float   *pfInPos, *pfOutPos;

   long   lMaxDelay;      // delay (in samples)
};


//------------------------------------------------------------------------------
// Modulated Delay Line Module (base class)
//------------------------------------------------------------------------------
class MModDelayLine : public MDelayLine
{
```

```
public:
   MModDelayLine();
   ~MModDelayLine() {}

   virtual float   GetCurValue(void) = 0;

   void   SetNomDelay(long delay);
   void   SetModDepth(long depth);
   void   SetModulator(MModulator* mod);
   long   GetNomDelay(void);
   long   GetModDepth(void);
   MModulator* GetModulator(void);

protected:
   // Buffer parameters
   float         fModOutPos;   // modulation output position

   long         lNomDelay;   // nominal delay (in samples)
   long         lModDepth;   // modulation depth (in samples)
   MModulator*   pMod;         // modulation module
};



//------------------------------------------------------------------------------
// Simplified 1st order All-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------
class MSAP1ModDelayLine : public MModDelayLine
{
public:
   MSAP1ModDelayLine() {buffer = 0;}
   ~MSAP1ModDelayLine() {}

   float   GetCurValue(void);

private:
   float buffer;
};


//------------------------------------------------------------------------------
// 1st order All-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------
class MAP1ModDelayLine : public MModDelayLine
{
public:
   MAP1ModDelayLine() {buffer = 0;}
   ~MAP1ModDelayLine() {}

   float   GetCurValue(void);

private:
   float buffer;
};


//------------------------------------------------------------------------------
// 1st order Low-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------
class MLP1ModDelayLine : public MModDelayLine
{
public:
   MLP1ModDelayLine() {}
   ~MLP1ModDelayLine() {}

   float   GetCurValue(void);
};


//------------------------------------------------------------------------------
// 2nd order Low-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------
class MLP2ModDelayLine : public MModDelayLine
{
public:
   MLP2ModDelayLine() {}
   ~MLP2ModDelayLine() {}

   float   GetCurValue(void);
};


//------------------------------------------------------------------------------
// 3rd order Low-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------
class MLP3ModDelayLine : public MModDelayLine
{
public:
   MLP3ModDelayLine() {}
   ~MLP3ModDelayLine() {}

   float   GetCurValue(void);
};


//------------------------------------------------------------------------------
// 4th order Low-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------
class MLP4ModDelayLine : public MModDelayLine
{
public:
   MLP4ModDelayLine() {}
   ~MLP4ModDelayLine() {}

   float   GetCurValue(void);
};


//------------------------------------------------------------------------------
// Simplified Variable Rate 1st order All-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------
class MSVRAP1ModDelayLine : public MModDelayLine
{
public:
   MSVRAP1ModDelayLine() {
      index = 0; lModRate = 1; buffer = 0; fOne_m_D = 0;
   }
   ~MSVRAP1ModDelayLine() {}

   float   GetCurValue(void);

   void   SetModRate(long rate);
   long   GetModRate(void);

private:
   long   index, lModRate;
   float fOne_m_D;
   float buffer;
};

#endif   // _MODULES_HPP
```

```
/*------------------------------------------------------------------------------

   File:        Modules.cpp

   Description: Implementation of useful C++ modules such as delay lines,
                filters, oscillators, etc.

   Author:      Jasmin Frenette
   Date:        11/27/2000

------------------------------------------------------------------------------*/

#include <windows.h>
#include <math.h>
#include <assert.h>
#include <crtdbg.h>
#include <memory.h>

#include "PlgMem.hpp"
#include "Modules.hpp"



//------------------------------------------------------------------------------
// Memory Management Procedures
//------------------------------------------------------------------------------

void Module::UpdateMemory(long* &pMem, long newSize)
{
   ChangeMemory((void*&)pMem, newSize*sizeof(long));
}

void Module::UpdateMemory(float* &pMem, long newSize)
{
   ChangeMemory((void*&)pMem, newSize*sizeof(float));
}

void Module::UpdateMemory(float** &pMem, long newSize)
{
   ChangeMemory((void*&)pMem, newSize*sizeof(float*));
}



//------------------------------------------------------------------------------
// 1st order FIR Low Pass Filter Module (version a)
//------------------------------------------------------------------------------

MLPF1a::MLPF1a() : Module()
{
   b0 = a1 = 0;

   Suspend();      // flush buffer
}

void MLPF1a::Suspend(void)
{
   buffer = 0;
}

void MLPF1a::SetCoeffs(float in_b0, float in_a1)
{
   b0 = in_b0;
   a1 = in_a1;
}

//-------------------------------------------------------------------------------
//   Get Functions

void MLPF1a::GetCoeffs(float* out_b0, float* out_a1)
{
   *out_b0 = b0;
   *out_a1 = a1;
}



//-------------------------------------------------------------------------------
// 1st order FIR Low Pass Filter Module (version b)
//-------------------------------------------------------------------------------

MLPF1b::MLPF1b() : Module()
{
   k = a1 = 0;

   Suspend();      // flush buffer
}

void MLPF1b::Suspend(void)
{
   bufferX = bufferY = 0;
}

void MLPF1b::SetCoeffs(float in_k, float in_a1)
{
   k = in_k;
   a1 = in_a1;
}

//-------------------------------------------------------------------------------
//   Get Functions

void MLPF1b::GetCoeffs(float* out_k, float* out_a1)
{
   *out_k = k;
   *out_a1 = a1;
}



//-------------------------------------------------------------------------------
// 2nd order FIR Low Pass Filter Module (version b)
//-------------------------------------------------------------------------------
MLPF2b::MLPF2b() : Module()
{
   k = b1 = a1 = a2 = 0;

   Suspend();      // flush buffer
}

void MLPF2b::Suspend(void)
{
   bufferX1 = bufferX2 = bufferY1 = bufferY2 = 0;
}
```

```
void MLPF2b::SetCoeffs(double in_k, double in_b1, double in_a1, double in_a2)
{
   k = in_k;
   b1 = in_b1;
   a1 = in_a1;
   a2 = in_a2;
}

//--------------------------------------------------------------------------------
//   Get Functions

void MLPF2b::GetCoeffs(double* out_k,double* out_b1,double* out_a1,double*
out_a2)
{
   *out_k = k;
   *out_b1 = b1;
   *out_a1 = a1;
   *out_a2 = a2;
}




//--------------------------------------------------------------------------------
// Sinusoidal Oscillator Module
//--------------------------------------------------------------------------------

MSinOsc::MSinOsc() : MModulator()
{
   fFreq = 0;
   a = 0;

   Suspend();      // flush buffer
}

MSinOsc::~MSinOsc()
{
}

void MSinOsc::Suspend(void)
{
   buffer1 = buffer2 = resetBuffer2 = 0;
}

void MSinOsc::SetFreq(float freq, float sampleRate, float phase)
{
   fFreq = freq;
   double w = TWOPI*fFreq/sampleRate;
   a = 2*cos(w);
   buffer2 = sin(TWOPI*phase/360 - w);
   buffer1 = sin(TWOPI*phase/360);
   resetBuffer2 = sin(PI_DIV_BY_2 - w);
}

float MSinOsc::GetFreq()
{
   return fFreq;
}




//--------------------------------------------------------------------------------
```

```
//  Random Number Generator Module
//---------------------------------------------------------------------------

MRNG::MRNG() : MModulator()
{
   a = c = seed = 0;

   Suspend();
}

MRNG::~MRNG()
{
}

void MRNG::Suspend(void)
{
   buffer = seed;
}

void MRNG::SetParams(unsigned int in_a, unsigned int in_c,
                     unsigned int in_seed)
{
   a = in_a;
   c = in_c;
   seed = in_seed;

   Suspend();
}

void MRNG::GetParams(unsigned int* out_a, unsigned int* out_c,
                     unsigned int* out_seed)
{
   *out_a = a;
   *out_c = c;
   *out_seed = seed;
}




//---------------------------------------------------------------------------
//  Low-Pass Filtered (4rth order) Module
//---------------------------------------------------------------------------

MLPFMod::MLPFMod() : MModulator()
{
   lRate = 0;
   pMod = NULL;
   pLPF1 = NULL;
   pLPF2 = NULL;

   index = 0;
}

MLPFMod::~MLPFMod()
{
}

void MLPFMod::SetRefreshRate(long rate)
{
   lRate = rate;
}
```

```
void MLPFMod::SetModulator(MModulator* mod)
{
   assert(mod!=NULL);
   pMod = mod;
}

void MLPFMod::SetLPF1(MLPF2b* lpf)
{
   assert(lpf!=NULL);
   pLPF1 = lpf;
}

void MLPFMod::SetLPF2(MLPF2b* lpf)
{
   assert(lpf!=NULL);
   pLPF2 = lpf;
}

long MLPFMod::GetRefreshRate(void)
{
   return lRate;
}

MModulator* MLPFMod::GetModulator(void)
{
   return pMod;
}

MLPF2b* MLPFMod::GetLPF1(void)
{
   return pLPF1;
}

MLPF2b* MLPFMod::GetLPF2(void)
{
   return pLPF2;
}

float MLPFMod::GetCurValue(void){
   double out;

   if(index == 0)
      out = pMod->GetCurValue();
   else
      out = 0;

   if(++index == lRate)
      index = 0;

   pLPF1->Process(out, out);
   pLPF2->Process(out, out);

   return (float) out;
}



//-----------------------------------------------------------------------------
// Delay Line Module
//-----------------------------------------------------------------------------

MDelayLine::MDelayLine() : Module()
```

```
{
   // Buffer parameters
   pfBuffer = pfBufferEnd = NULL;
   lBufferSize = 0;
   pfInPos = pfOutPos = NULL;

   lMaxDelay = 0;

   Suspend();      // flush buffer
}

MDelayLine::~MDelayLine()
{
   if(pfBuffer)
      UpdateMemory(pfBuffer,0);
}

void MDelayLine::Suspend(void)
{
   if(pfBuffer)
      memset(pfBuffer, 0, lBufferSize * sizeof(float));
}

long MDelayLine::GetMem(void)
{
   return (long) (lBufferSize * sizeof(float));
}

void MDelayLine::SetMaxDelay(long delay)
{
   lMaxDelay = delay;

   lBufferSize = lMaxDelay + 10 + 1;   // keeps 10 extra samples for interp.

   UpdateMemory(pfBuffer,lBufferSize);
   pfBufferEnd = pfBuffer + lBufferSize;

   pfInPos = pfBuffer;

   pfOutPos = pfInPos - lMaxDelay;
   if(pfOutPos < pfBuffer)
      pfOutPos += lBufferSize;
}

//-------------------------------------------------------------------------------
//   Get Functions

long MDelayLine::GetMaxDelay(void)
{
   return lMaxDelay;
}




//-------------------------------------------------------------------------------
// Modulated Delay Line Module
//-------------------------------------------------------------------------------

MModDelayLine::MModDelayLine() : MDelayLine()
{
   fModOutPos = 0;
   lNomDelay = 0;
   lModDepth = 0;
   pMod = NULL;
}

void MModDelayLine::SetNomDelay(long delay)
{
   lNomDelay = delay;

   SetMaxDelay(lNomDelay+lModDepth);

   // update output position (buffer size has changed)
   fModOutPos = (float)(pfInPos - pfBuffer - lNomDelay);
   if(fModOutPos < 0)
      fModOutPos += lBufferSize;
}

void MModDelayLine::SetModDepth(long depth)
{
   if(depth>=lNomDelay){
      DebugBreak();
      depth = lNomDelay-1;
      return;
   }

   lModDepth = depth;

   SetMaxDelay(lNomDelay+lModDepth);

   // update output position (buffer size has changed)
   fModOutPos = (float)(pfInPos - pfBuffer - lNomDelay);
   if(fModOutPos < 0)
      fModOutPos += lBufferSize;
}

void MModDelayLine::SetModulator(MModulator* mod)
{
   assert(mod!=NULL);
   pMod = mod;
}

//-----------------------------------------------------------------------------
//   Get Functions

long MModDelayLine::GetNomDelay(void)
{
   return lNomDelay;
}

long MModDelayLine::GetModDepth(void)
{
   return lModDepth;
}

MModulator* MModDelayLine::GetModulator(void)
{
   return pMod;
}




//-----------------------------------------------------------------------------
// Simplified 1st order All-Pass Modulated Delay Line Module
```

```
//-----------------------------------------------------------------------------

float MSAP1ModDelayLine::GetCurValue(void)
{
   float fOne_m_D, fOutIndex;
   long  lIndex, lIndex_p_1;
   float out;

   fOutIndex = (float)(fModOutPos + pMod->GetCurValue()*lModDepth);

   if(fOutIndex>=0)
      lIndex = (long)(fOutIndex);
   else
      lIndex = (long)(fOutIndex-1);

   lIndex_p_1 = lIndex + 1;
   fOne_m_D = fOutIndex - lIndex;

   // Boundary check
   if(lIndex_p_1 >= lBufferSize){
      lIndex_p_1 -= lBufferSize;
      if(lIndex >= lBufferSize)
         lIndex -= lBufferSize;
   }
   else if(lIndex < 0){
      lIndex += lBufferSize;
      if(lIndex_p_1 < 0){
         lIndex_p_1 += lBufferSize;
      }
   }

   // Interpolation
   out =  *(pfBuffer + lIndex);
   out += fOne_m_D * (*(pfBuffer + lIndex_p_1) - buffer);
   buffer = out;

   if(++fModOutPos >= lBufferSize)
      fModOutPos -= lBufferSize;

   return out;
}



//-----------------------------------------------------------------------------
// 1st order All-Pass Modulated Delay Line Module
//-----------------------------------------------------------------------------

float MAP1ModDelayLine::GetCurValue(void)
{
   float fOne_m_D, fOne_p_D, fOutIndex;
   long  lIndex, lIndex_p_1;
   float out;

   fOutIndex = (float)(fModOutPos + pMod->GetCurValue()*lModDepth);

   if(fOutIndex>=0)
      lIndex = (long)(fOutIndex);
   else
      lIndex = (long)(fOutIndex-1);

   lIndex_p_1 = lIndex + 1;
   fOne_m_D = fOutIndex - lIndex;
   fOne_p_D = 1 + lIndex_p_1 - fOutIndex;

   // Boundary check
   if(lIndex_p_1 >= lBufferSize){
      lIndex_p_1 -= lBufferSize;
      if(lIndex >= lBufferSize)
         lIndex -= lBufferSize;
   }
   else if(lIndex < 0){
      lIndex += lBufferSize;
      if(lIndex_p_1 < 0){
         lIndex_p_1 += lBufferSize;
      }
   }

   // Interpolation
   out =  *(pfBuffer + lIndex);
   out += fOne_m_D / fOne_p_D * (*(pfBuffer + lIndex_p_1) - buffer);
   buffer = out;

   if(++fModOutPos >= lBufferSize)
      fModOutPos -= lBufferSize;

   return out;
}




//------------------------------------------------------------------------------
// 1st order Low-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------

float MLP1ModDelayLine::GetCurValue(void)
{
   float fD, fOutIndex;
   long  lIndex, lIndex_p_1;
   float out;

   fOutIndex = (float)(fModOutPos + pMod->GetCurValue()*lModDepth);

   if(fOutIndex>=0)
      lIndex = (long)(fOutIndex);
   else
      lIndex = (long)(fOutIndex-1);

   lIndex_p_1 = lIndex + 1;
   fD = fOutIndex - lIndex;

   // Boundary check
   if(lIndex_p_1 >= lBufferSize){
      lIndex_p_1 -= lBufferSize;
      if(lIndex >= lBufferSize)
         lIndex -= lBufferSize;
   }
   else if(lIndex < 0){
      lIndex += lBufferSize;
      if(lIndex_p_1 < 0){
         lIndex_p_1 += lBufferSize;
      }
   }
```

```
/*   if ((lIndex<0) || (lIndex>=lBufferSize)
      || (lIndex_p_1<0) || (lIndex_p_1>=lBufferSize))
      DebugBreak();*/

   // Interpolation
   out =  fD * (*(pfBuffer + lIndex_p_1));
   out += (1-fD) * (*(pfBuffer + lIndex));

   if(++fModOutPos >= lBufferSize)
      fModOutPos -= lBufferSize;

   return out;
}




//------------------------------------------------------------------------------
// 2nd order Low-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------

float MLP2ModDelayLine::GetCurValue(void)
{
   float fD, fOutIndex;
   long  lIndex, lIndex_p_1, lIndex_p_2;
   float out;

   fOutIndex = (float)(fModOutPos + pMod->GetCurValue()*lModDepth);

   if(fOutIndex>=0)
      lIndex = (long)(fOutIndex);
   else
      lIndex = (long)(fOutIndex-1);

   lIndex_p_1 = lIndex + 1;
   lIndex_p_2 = lIndex_p_1 + 1;
   fD = fOutIndex - lIndex;

   // Boundary check
   if(lIndex_p_2 >= lBufferSize){
      lIndex_p_2 -= lBufferSize;
      if(lIndex_p_1 >= lBufferSize){
         lIndex_p_1 -= lBufferSize;
         if(lIndex >= lBufferSize)
            lIndex -= lBufferSize;
      }
   }
   else if(lIndex < 0){
      lIndex += lBufferSize;
      if(lIndex_p_1 < 0){
         lIndex_p_1 += lBufferSize;
         if(lIndex_p_2 < 0){
            lIndex_p_2 += lBufferSize;
         }
      }
   }

   float ppf0 = fD;
   float ppf1 = ppf0*(fD-1);

   float ppr0 = (fD-2);
   float ppr1 = ppr0*(fD-1);
   // Interpolation
   out = ppr1*(float)0.5 * (*(pfBuffer + lIndex));
   out -= (ppf0*ppr0) * (*(pfBuffer + lIndex_p_1));
   out += ppf1*(float)0.5 * (*(pfBuffer + lIndex_p_2));

   if(++fModOutPos >= lBufferSize)
      fModOutPos -= lBufferSize;

   return out;
}




//------------------------------------------------------------------------------
// 3rd order Low-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------

float MLP3ModDelayLine::GetCurValue(void)
{
   float fD, fOutIndex;
   long  lIndex, lIndex_p_1, lIndex_p_2, lIndex_p_3;
   float out;

   fOutIndex = (float)(fModOutPos + pMod->GetCurValue()*lModDepth);

   if(fOutIndex>=0)
      lIndex_p_1 = (long)(fOutIndex);
   else
      lIndex_p_1 = (long)(fOutIndex-1);

   lIndex = lIndex_p_1 - 1;
   lIndex_p_2 = lIndex_p_1 + 1;
   lIndex_p_3 = lIndex_p_1 + 2;
   fD = fOutIndex - lIndex;

   // Boundary check
   if(lIndex_p_3 >= lBufferSize){
      lIndex_p_3 -= lBufferSize;
      if(lIndex_p_2 >= lBufferSize){
         lIndex_p_2 -= lBufferSize;
         if(lIndex_p_1 >= lBufferSize){
            lIndex_p_1 -= lBufferSize;
            if(lIndex >= lBufferSize)
               lIndex -= lBufferSize;
         }
      }
   }
   else if(lIndex < 0){
      lIndex += lBufferSize;
      if(lIndex_p_1 < 0){
         lIndex_p_1 += lBufferSize;
         if(lIndex_p_2 < 0){
            lIndex_p_2 += lBufferSize;
            if(lIndex_p_3 < 0)
               lIndex_p_3 += lBufferSize;
         }
      }
   }

   float ppf0 = fD;
   float ppf1 = ppf0*(fD-1);
   float ppf2 = ppf1*(fD-2);

   float ppr0 = (fD-3);
   float ppr1 = ppr0*(fD-2);
   float ppr2 = ppr1*(fD-1);

   out = ppr2*(float)(-0.16666666666666667) * (*(pfBuffer + lIndex));
   out += (ppr2+ppr1)*(float)0.5 * (*(pfBuffer + lIndex_p_1));
   out -= (ppf2-ppf1)*(float)0.5 * (*(pfBuffer + lIndex_p_2));
   out += ppf2*(float)0.16666666666666667 * (*(pfBuffer + lIndex_p_3));

   // 6 2 2 6

   if(++fModOutPos >= lBufferSize)
      fModOutPos -= lBufferSize;

   return out;
}




//------------------------------------------------------------------------------
// 4th order Low-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------

float MLP4ModDelayLine::GetCurValue(void)
{
   float fD, fOutIndex;
   long  lIndex, lIndex_p_1, lIndex_p_2, lIndex_p_3, lIndex_p_4;
   float out;

   fOutIndex = (float)(fModOutPos + pMod->GetCurValue()*lModDepth);

   if(fOutIndex>=0)
      lIndex_p_1 = (long)(fOutIndex);
   else
      lIndex_p_1 = (long)(fOutIndex-1);

   lIndex = lIndex_p_1 - 1;
   lIndex_p_2 = lIndex_p_1 + 1;
   lIndex_p_3 = lIndex_p_1 + 2;
   lIndex_p_4 = lIndex_p_1 + 3;
   fD = fOutIndex - lIndex;

   // Boundary check
   if(lIndex_p_4 >= lBufferSize){
      lIndex_p_4 -= lBufferSize;
      if(lIndex_p_3 >= lBufferSize){
         lIndex_p_3 -= lBufferSize;
         if(lIndex_p_2 >= lBufferSize){
            lIndex_p_2 -= lBufferSize;
            if(lIndex_p_1 >= lBufferSize){
               lIndex_p_1 -= lBufferSize;
               if(lIndex >= lBufferSize)
                  lIndex -= lBufferSize;
            }
         }
      }
   }
   else if(lIndex < 0){
      lIndex += lBufferSize;
      if(lIndex_p_1 < 0){
         lIndex_p_1 += lBufferSize;
         if(lIndex_p_2 < 0){
            lIndex_p_2 += lBufferSize;
            if(lIndex_p_3 < 0){
               lIndex_p_3 += lBufferSize;
               if(lIndex_p_4 < 0)
                  lIndex_p_4 += lBufferSize;
            }
         }
      }
   }

   float ppf0 = fD;
   float ppf1 = ppf0*(fD-1);
   float ppf2 = ppf1*(fD-2);
   float ppf3 = ppf2*(fD-3);

   float ppr0 = (fD-4);
   float ppr1 = ppr0*(fD-3);
   float ppr2 = ppr1*(fD-2);
   float ppr3 = ppr2*(fD-1);

   // Interpolation
   out  = ppr3*(float)0.041666666666666667 * (*(pfBuffer + lIndex));
   out -= (ppr3+ppr2)*(float)0.16666666666666667 * (*(pfBuffer + lIndex_p_1));
   out += (ppf1*ppr1)*(float)0.25 * (*(pfBuffer + lIndex_p_2));
   out -= (ppf3-ppf2)*(float)0.16666666666666667 * (*(pfBuffer + lIndex_p_3));
   out += ppf3*(float)0.041666666666666667 * (*(pfBuffer + lIndex_p_4));

   // 24 6 4 6 24

   if(++fModOutPos >= lBufferSize)
      fModOutPos -= lBufferSize;

   return out;
}




//------------------------------------------------------------------------------
// Simplified Variable Rate 1st order All-Pass Modulated Delay Line Module
//------------------------------------------------------------------------------

void MSVRAP1ModDelayLine::SetModRate(long rate)
{
   assert(rate<=lBufferSize);
   lModRate = rate;
}

long MSVRAP1ModDelayLine::GetModRate(void)
{
   return lModRate;
}

float MSVRAP1ModDelayLine::GetCurValue(void)
{
   float fOutIndex, out;
   long lIndex;

   if(((index++)%lModRate) == 0){
      fOutIndex = (float)(fModOutPos + pMod->GetCurValue()*lModDepth);
      if(fOutIndex >= 0.0f)
         lIndex = (long)(fOutIndex);
      else
         lIndex = (long)(fOutIndex-1);

      pfOutPos   = pfBuffer + lIndex;

      // Boundary check
      if(pfOutPos < pfBuffer)
         pfOutPos += lBufferSize;
      if(pfOutPos >= pfBufferEnd)
         pfOutPos -= lBufferSize;

      fOne_m_D = fOutIndex - lIndex;

      fModOutPos += lModRate;
      if(fModOutPos >= lBufferSize)
         fModOutPos -= lBufferSize;
   }

   // Interpolation
   out = *(pfOutPos++);

   // Boundary check
   if(pfOutPos >= pfBufferEnd)
      pfOutPos -= lBufferSize;

   // Interpolation
   out += fOne_m_D * (*(pfOutPos) - buffer);
   buffer = out;

   return out;
}
```

```
/*-------------------------------------------------------------------------------

   File:        PlgMem.hpp

   Description: Memory management utilities

   Author:      Jasmin Frenette
   Date:        11/27/2000

-------------------------------------------------------------------------------*/
#ifndef _PLGMEM_HPP
#define _PLGMEM_HPP

void ChangeMemory(void* &pMem, long newSize);

#endif // _PLGMEM_HPP
```

```
/*-------------------------------------------------------------------------------

   File:        PlgMem.cpp

   Description: Implementation of memory management utilities

   Author:      Jasmin Frenette
   Date:        11/27/2000

-------------------------------------------------------------------------------*/

#include <windows.h>
#include "PlgMem.hpp"

void ChangeMemory(void* &pMem, long newSize)
{
   void*   pTemp;

   if(pMem == NULL){      // There was no previous memory allocated
      if(newSize!=0)
         pMem = (void*) GlobalAlloc(GPTR,newSize);
   }
   else{
      if(newSize!=0){   // Add or remove memory space
         pTemp = (void*) GlobalAlloc(GPTR,newSize);
         memcpy(pTemp, pMem, min(GlobalSize(pMem), (unsigned long)newSize));
         GlobalFree(pMem);
         pMem = pTemp;
      }
      else{             // Free the pointer
         GlobalFree(pMem);
         pMem = NULL;
      }
   }
}
```