Long Sea - Feedback Delay Network Based Reverberation Algorithm for Imaginary Spaces

Simon Rostami Mosen, student nb.: 20176458

January 2021

Contents

1 Introduction

Since Schröder developed the first digital signal processing (DSP) based approach to artificial reverberation (reverb) in 1961 [6], a multitude of different algorithms has been invented [9]. While early approaches were based on *comb-filters* and *allpass-filters*[6], Gerzon proposed a way of creating reverb from matrix-based feedback of delay lines, today known as feedback delay network (FDN) [2].

In contrast to much of the early research of artificial reverb which focused on synthesizing naturally sounding reverb (such as [5]), this project - Long Sea - documents the technical aspects of a reverb algorithm for *imaginary spaces* using FDNs. While conceptually, this project is inspired by the work of Tom Erbe (Erbe-Verb)[1] and Sean Costello (Valhalla DSP)[4], the structure of the implementation is inspired by Eric Tarr's Hack Audio [8].

2 Structure

The structure of this algorithm is a novel take on FDN-based reverberation as it engages three cascading 16-delay line FDNs tuned-in at different delay lengths (described in section 5). While the FDNs are implemented using a Fractional Delay class (described in section 3, the FDNs themselves have a separate class (described in section 4). A Hadamard feedback matrix is used for the FDN (described in section 4.2). To apply width to the reverberation, a pseudo-stereo effect is created by splitting the wet signal and simply delaying one channel by a small amount (described in section 7).

Long Sea is developed using JUCE[1] and is exported as a VST3 and tested using Ableton Live 10[2].

2.1 Block diagrams

A diagram illustrating the overall structure of the algorithm can be seen below in figure 1. The figure above illustrates how the input signal travels through a predelay, simply

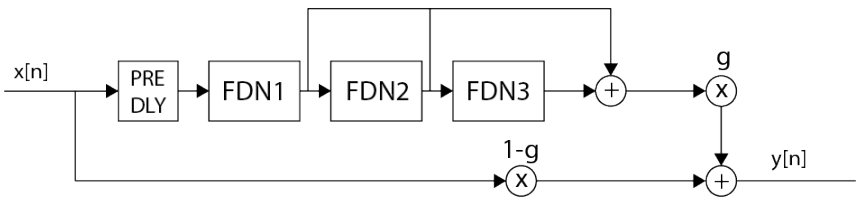


Figure 1: *Overview of algorithm*

being a variable delay and from there going into the cascading FDNs which also passes their outputs individually to be summed with the output of the full series of FDNs.

[1] https://juce.com/
[2] https://ableton.com

As the nature of a 16-delay line FDN is relatively extensive, the full diagram of the structure of each FDN can be found in Appendix A. To illustrate the shape of a FDN, a block diagram of a 4-delay line FDN is found below on figure 2.

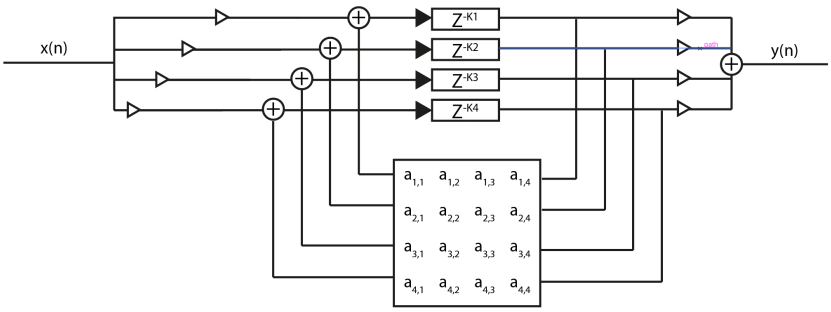


Figure 2: *4-delay line FDN*

2.2 Code setup

The implementation of this algorithm is made in JUCE and the code-structure of this implementation is inspired by Eric Tarr's implementation of a 4-delay line FDN VST [8]. The JUCE project consists of the following files:

- *PluginProcessor.cpp*: where all DSP happens
- *PluginProcessor.h*: a header file for the *PluginProcessor.cpp*
- *PluginEditor.cpp*: where all GUI related code is contained
- *PluginEditor.h*: a header file for the *PluginEditor.cpp*
- *FDN.cpp*: contains the implementation of the methods from the FDN class
- *FDN.h*: contains the class definition and methods declarations for *FDN.cpp*
- *FracDelay.cpp*: contains the implementation of the methods from the FractDelay class.
- *FractDelay.h*: contains the class definition and methods declarations for *FracDelay.cpp*

.

3 Fractional Delay & Circular Buffer

A class was used to handle all delay-lines. As delay-lines are being modulated (see sec. 6), it is sensible to use fractional delays to avoid undesirable artefacts. Instead of relying only on whole samples, fractional delays (also known as interpolated delays) estimate values between samples using interpolation [7]. In this implementation linear interpolation is used. Linear interpolation estimates values based on a linear combination of the two neighbouring samples [7]. Linear interpolation can be written as:

$$(1 - frac) \cdot x_m + (frac) \cdot x_{m+1}$$

where $frac$ is the fraction of the sample value exceeding the whole sample, meaning that $0 < frac < 1$, x_m is one of the neighbouring samples and x_{m+1} is the other neighbouring sample [8]. $1 - frac$ and $frac$ is thereby used to 'weight' the two samples in the interpolation. In the implementation of linear interpolation, the $frac$ and $1 - frac$ was found. The `floor` function neglects all decimals and returns the biggest whole integer.

Listing 1 Finding weights for interpolated delay

```
// define neighbouring samples
int d1 =floor(delay+lfo); int d2 =d1 +1;

// subtracting the floored version from the non-floored version,
   leaving decimals only. Equivalent to frac in the formula.
float weight2 =delay +lfo -(float)d1;
// 1-frac
float weight1 =1.0f -weight2;
```

On the code-snippet below, the weight1 and weight2 (1-frac and frac) variables are seen in the implementation of linear interpolation.

```
float y =weight1 *delayBuffer[readIndexD1][channel] +weight2 *
   delayBuffer[readIndexD2][channel];
```

This implementation uses a circular buffer to facilitate delay-lines. A circular buffer works similarly to linear buffers, but instead of shifting all elements of the buffer, only read and write pointers are shifted, resulting in less operations and thereby a lower CPU usage [8]. A 2-dimensional float matrix was initialised in the FracDelay.h file. 96000 indicates the maximal buffer size and 2 is the number of channels (stereo).

Listing 2 Defining delayBuffer variable

```
float delayBuffer[96000][2] ={0.0f};
```

As this buffer is based on fractional delays, it consists of one write pointer and two read pointers one sample apart. All pointers are 'circulated' by constantly being incremented and by setting their value to 0 once they approach the `MAX_BUFFER_SIZE`. The implementation of this is seen below.

Listing 3 Circulating read and write pointers

```
  // circulate read indices. x= MAX_BUFFER_SIZE acts as modulo
readIndexD1 =writeIndex[channel] -d1;
    if (readIndexD1 <0){
        readIndexD1 +=MAX_BUFFER_SIZE; }

readIndexD2 =writeIndex[channel] -d2;
        if (readIndexD2 <0){
            readIndexD2 +=MAX_BUFFER_SIZE; }
[...]
delayBuffer[writeIndex[channel]][channel] =x;

// circulate write index
if (writeIndex[channel] <MAX_BUFFER_SIZE -1){
    writeIndex[channel]++; }else{
    writeIndex[channel] =0;}
```

This class is very much based on a Eric Tarr's tutorial on FDN's from his Patreon [8].

4 FDN Class

The FDN class is in interplay with the Fractional Delay class the very backbone of this algorithm. The FDN class uses the Fractional Delay class to create 16 instances of delay lines at different delay lengths and with different modulation speeds, given in the constructor instantiating the FDN.

4.1 Constructor

As this algorithm uses three FDNs in series, the creation of three instances of the FDN class with different delay lengths and modulation speeds was made possible. The constructor of the FDN class is seen below:

```
FDN::FDN(float delay1, float speed1, float delay2, float speed2, [
    ...]float delay16, float speed16){
```

The instance variables given in the constructor are then to set delay and modulation speed values in the different instances of delay lines.

4.2 Feedback Matrix

One of the cornerstones of FDNs is the feedback matrix (also called the scattering matrix). The feedback matrix is used to contain all feedback gains in the system [7]. The output of each delay line is fed back to each delay line with the gain value given in the matrix [7]. For this algorithm, a 16th-order Hadamard matrix[3] is used. The Hadamard matrix is seen below in table 1:

+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-
+	+	-	-	+	+	-	-	+	+	-	-	+	+	-	-
+	-	-	+	+	-	-	+	+	-	-	+	+	-	-	+
+	+	+	+	-	-	-	-	+	+	+	+	-	-	-	-
+	-	+	-	-	+	-	+	+	-	+	-	-	+	-	+
+	+	-	-	-	-	+	+	+	+	-	-	-	-	+	+
+	-	-	+	-	+	+	-	+	-	-	+	-	+	+	-
+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-
+	-	+	-	+	-	+	-	-	+	-	+	-	+	-	+
+	+	-	-	+	+	-	-	-	-	+	+	-	-	+	+
+	-	-	+	+	-	-	+	-	+	+	-	-	+	+	-
+	+	+	+	-	-	-	-	-	-	-	-	+	+	+	+
+	-	+	-	-	+	-	+	-	+	-	+	+	-	+	-
+	+	-	-	-	-	+	+	-	-	+	+	+	+	-	-
+	-	-	+	-	+	+	-	-	+	+	-	+	-	-	+

Table 1: 16th-order Hadamard Matrix

To implement this matrix, a float variable for each entry of the matrix was created. Another set of variables was, furthermore, created to represent the 16 feedback paths. As seen below, the output of all 16 delay lines are assigned to each feedback variable and is gained by the according value from the Hadamard matrix.

Listing 4 Hadamard matrix implemented as floats

[3] http://neilsloane.com/hadamard/had.16.0.txt

```
float g1_1 =1.f, g1_2 =1.f, g1_3 =1.f, g1_4 =1.f, g1_5 =1.f, g1_6 =1
    .f, g1_7 =1.f, g1_8 =1.f, g1_9 =1.f, g1_10 =1.f, g1_11 =1.f,
    g1_12 =1.f, g1_13 =1.f, g1_14 =1.f, g1_15 =1.f, g1_16 =1.f;
float g2_1 =1.f, g2_2 =-1.f, g2_3 =1.f, g2_4 =-1.f, g2_5 =1.f, g2_6
    =-1.f, g2_7 =1.f, g2_8 =-1.f, g2_9 =1.f, g2_10 =-1.f, g2_11 =1
    .f, g2_12 =-1.f, g2_13 =1.f, g2_14 =-1.f, g2_15 =1.f, g2_16 =-1
    .f;
[...]
float g16_1 =g8_1, g16_2 =g8_2, g16_3 =g8_3, g16_4 =g8_4, g16_5 =
    g8_5, g16_6 =g8_6, g16_7 =g8_7, g16_8 =g8_8, g16_9 =-g8_9,
    g16_10 =-g8_10, g16_11 =-g8_11, g16_12 =-g1_12, g16_13 =-g1_13,
    g16_14 =-g8_14, g16_15 =-g8_15, g16_16 =-g8_16;
```

4.3 Organizing inputs and outputs

To organize the different inputs and outputs in the feedback matrix, a float for the input, as well as the output, of each delay line is created. As seen in the code below, each input variable contains x (input signal) and a feedback variable while each output variable engages the `processSample` method from the Fractional Delay class.

Listing 5 Inputs to the delay-lines and outputs from the delay-lines

```
float inDL1 =x +fb1[channel];
float inDL2 =x +fb2[channel];
[...]
float inDL16 =x +fb16[channel];

float outDL1 =fractionalDelay1.processSample(inDL1, channel);
float outDL2 =fractionalDelay2.processSample(inDL2, channel);
[...]
float outDL16 =fractionalDelay16.processSample(inDL16, channel);
```

The feedback (`fb1, fb2 ...` variables are created by taking all delay-lines and weighing by the gain values given in the matrix. This was done for each of the 16 delay lines. This can be seen below.

Listing 6 Weighing of all outputs of delay lines based on matrix

```
fb1[channel] =(g1_1*outDL1 +g1_2*outDL2 +[...]+g1_16*outDL16) *
    feedbackGain;
```

```
fb2[channel] =(g2_1*outDL1 +g2_2*outDL2 +[...]+g2_16*outDL16) *
    feedbackGain;
[...]
fb16[channel] =(g16_1*outDL1 +g16_2*outDL2 +[...]+g16_16*outDL16) *
    feedbackGain;
```

It can be argued that this approach of implementing a matrix is very cumbersome and hard-coded (see sec. 9).

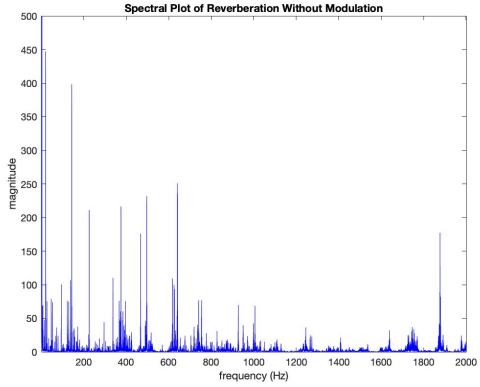
5 Cascading FDNs

The novelty in this algorithm comes from the cascading three FDNs. Each FDN was instantiated with different delay lengths. The first FDN was instantiated with short delay lengths ranging from 353 samples to 6043 samples. The second FDN used delay lengths ranging from 6131 samples to 15787 samples. The last FDN using delay lengths ranging from 15083 samples to 16433 samples, delay lengths that are rarely seen within reverberation design as they approach echo-like delay lengths. All delay lengths were chosen based on prime numbers and experimentation to reduce resonant frequencies building up from the feedback. To invite experimentation, a toggle for each FDN is included in the GUI.

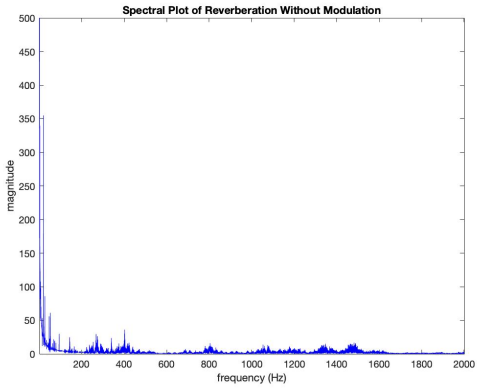
The long As the delay-lengths of the last FDN has the side-effect that the overall reverberated signal is delayed by a minimum of 15083 samples, which is an audible amount. It was not desired to reduce these delay lengths, as this was a key-feature to approach very long reverberation times without feedback. Instead, the audible gap between dry and reverberated signal was eliminated by also feeding the output from the first and second FDN, in parallel, directly to the output, while still feeding into the cascading FDNs. This is also illustrated earlier in fig. 1. Creating three FDNs allowed for a versatile implementation of very lush and thick sounding reverberation as well as refining of specific parts of the reverberation tail.

6 Modulated Delays

Even though delay lengths were picked among prime numbers, resonant frequencies still build up, introducing a 'metallic ringing' effect. To eliminate this, sine-wave modulation for each delay line was implemented. By applying an LFO, the delay lengths could vary by an amount (given in the GUI - range 1 - 30 samples). The effect of the modulation is illustrated below on FFT plots created using the `mirspectrum` from the MIRToolbox for MATLAB [3]. The algorithm was excited by a short burst of white noise.



(a) FFT Plot of Non-Modulated Reverberation



(b) FFT Plot of Modulated Reverberation

Figure 3: FFT Plots of Reverberation

As it is seen on fig. 3a, many resonant frequencies were present. It is, furthermore, illustrated in fig. 3b that modulation of delay lengths (in this case 30 samples) drastically reduces the magnitude of the peak.

7 In Stereo

To introduce a bit of width to the algorithm, a simple 'pseudo-stereo' effect was created. Essentially, this effect was implemented by splitting the wet signal, delaying one of the signals by a certain amount (1800 samples in this case), and assigning these two reverberated paths to each channel (left/right). The implementation is seen below.

Listing 7 Instantiating delay line for pseudo-stereo effect

```
FracDelay stereoDelay{1800.f, 0.0f};
```

Listing 8 Splitting wet path

```
float verbCombined =fdn1Gain *verbEarly -fdn2Gain*0.8f*verbMid +
    fdn3Gain*0.6 *verb;
float verbCombinedDelayed =stereoDelay.processSample(verbCombined,
    channel);
float y =(1.f -wet) *x + verbCombined *wet;
float delayY =(1.f -wet) *x + verbCombinedDelayed *wet;

buffer.getWritePointer(0)[n] =y;
```

```
buffer.getWritePointer(1)[n] =delayY;
```

The first line of the code-snippet above illustrates how the output of all FDNs are gained and summed. The following lines show how the wet path was split into two, to create two separate signals for the left and right channel. The bottom-two lines illustrate how each signal was written to separate channels of the output buffer.

This very simple effect, proved to increase the width and added to the overall experience of the sound.

8 GUI

The GUI consists of knobs for controlling time (0.10 to 0.20, an arbitrary gain value), modulation (1 to 30 samples), predelay (0 to 200 ms) and dry/wet (0 to 100, 0 being all dry and 100 being all wet). Furthermore, three buttons were included to toggle each FDN. The toggles currently lack feedback, but are toggled *on* be default as the plugin is loaded. A screenshot of the GUI can be seen below:

Figure 4: *Overview of GUI*

9 Further work

In future work, investigating a better way of implementing the scattering matrix might prove beneficial, as this could allow for easy experimentation with different matrices.

A low-pass filter at the very end of the wet path could be an interesting feature to allow the user simply to tweak the ”brightness” of the sound.

As seen in the FFT plots on fig. 3a and 3b a resonant peak outside of audible range just above 0 hertz is present. This could be eliminated by a simple high-pass filter.

As big FDNs are considered computationally heavy, it could be worth looking into ways of optimizing the algorithm to have a lower impact on CPU usage. Long Sea currently takes up 27% of CPU power on a mid-2014 Macbook Pro Retina, 2.5 GHz i7.

10 References

[1] Tom Erbe. Building the erbe-verb: Extending the feedback delay network reverb for modular synthesizer use. In *ICMC*, 2015.

[2] Michael A Gerzon. Synthetic stereo reverberation: Part one. *Studio Sound*, 13:632–635, 1971.

[3] Olivier Lartillot and Petri Toiviainen. A matlab toolbox for musical feature extraction from audio. In *International conference on digital audio effects*, pages 237–244. Bordeaux, 2007.

[4] The Audio Programmer. Sean costello (valhalla dsp) - qa with valhalla dsp. `https://www.youtube.com/watch?v=z8eyiIcBT34`.

[5] Manfred R Schroeder. Natural sounding artificial reverberation. *Journal of the Audio Engineering Society*, 10(3):219–223, 1962.

[6] M.R Schroeder and B.F Logan. "colorless" artificial reverberation. *I.R.E. transactions on audio*, AU-9(6):209–214, 1961.

[7] E. Tarr. *Hack Audio: An Introduction to Computer Programming and Digital Signal Processing in MATLAB*. Audio Engineering Society presents. Routledge, 2018.

[8] Eric Tarr. Hack audio. `https://www.patreon.com/hackaudio`.

[9] Vesa Valimaki, Julian D Parker, Lauri Savioja, Julius O Smith, and Jonathan S Abel. Fifty years of artificial reverberation. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(5):1421–1448, 2012.

11 Appendix A: 16 delay-line FDN Block Diagram