ENS 491-2 Graduation Project

Project # 55
Developing an Advanced Sampling System

Submitted in fulfillment of the requirements for the degree of

Bachelor of Science

Can Ince

14241

Supervised by

Müjdat Çetin

November 17, 2015

Faculty of Engineering and Natural Sciences

Sabanci University

Contents

Abstract

This project aims to create a sampling system which can be embedded into several applications as a sample playback engine and can be used as a live sampling software for live performances. The project grows within the scope of filter and processor design.

I. Introduction

With the power of the contemporary CPUs and frameworks, the new media has evolved into a modular canvas. The term "New Media" consists of the mutual correlation of science and art. This correlation gave birth to an era-specific approaches to music and changed it's origins from classical to algorithmic. While this continuous evolution has accelerated in the past couple of years, the compatibility issues has not yet been quite solved.

The New Music has been built onto three main fields; Audio Signal Processing, Embeddable Systems and algorithmic compositions. Modern Digital Audio Workstations (DAW) offer an extensive music production environment with countless plug-ins that expand the variety of sounds and the possibilities of various manipulations but the limitations of the modern workstations can not be overseen. Given the endless possibilities of creating tools for music, the most challenging part is to understand the concepts lying below such as processors, modulations, envelopes and the modular way of thinking to combine those modules. In the course of my project I have decided to design a system which will be able to manipulate an audio file in a way that old school sampler and synthesizer does. There are many different unique approaches to playback an audio file but what is missing is the compatibility and the completeness of those approaches in different areas.

The main focus of the project is based on gen object since it works at single sample level therefore it allows more precise manipulations on signal without having to code them in C. All of the digital signal processing modules were coded in gen and tried several processor designs and combinations through out the project.

II. Overview of Patch

I. Summary of Max/MSP Patch

To get an insight about the processor design I began with an MSP patch. At first, the patch included 8 different filter type, fine-tuning, amplitude and filter envelopes, delay, reverb and several other sample manipulation tools. The individual modules parse the required information such as MIDI notes, ADSR values and control change messages from the various list objects and routes them accordingly. The output of the individual modules gets combined in *mainsamp* sub-patcher and forms the final list object with the given information. After the generation of the specified sample output the signal goes into the processors and filter. With the polyphony, the same sample can be played back with different frequencies eventually effecting the pitch of the sample. However, it was not doable sequential way and it required a dynamic way of programming. For each input the whole system is re-created and evaluated in it's own scope.

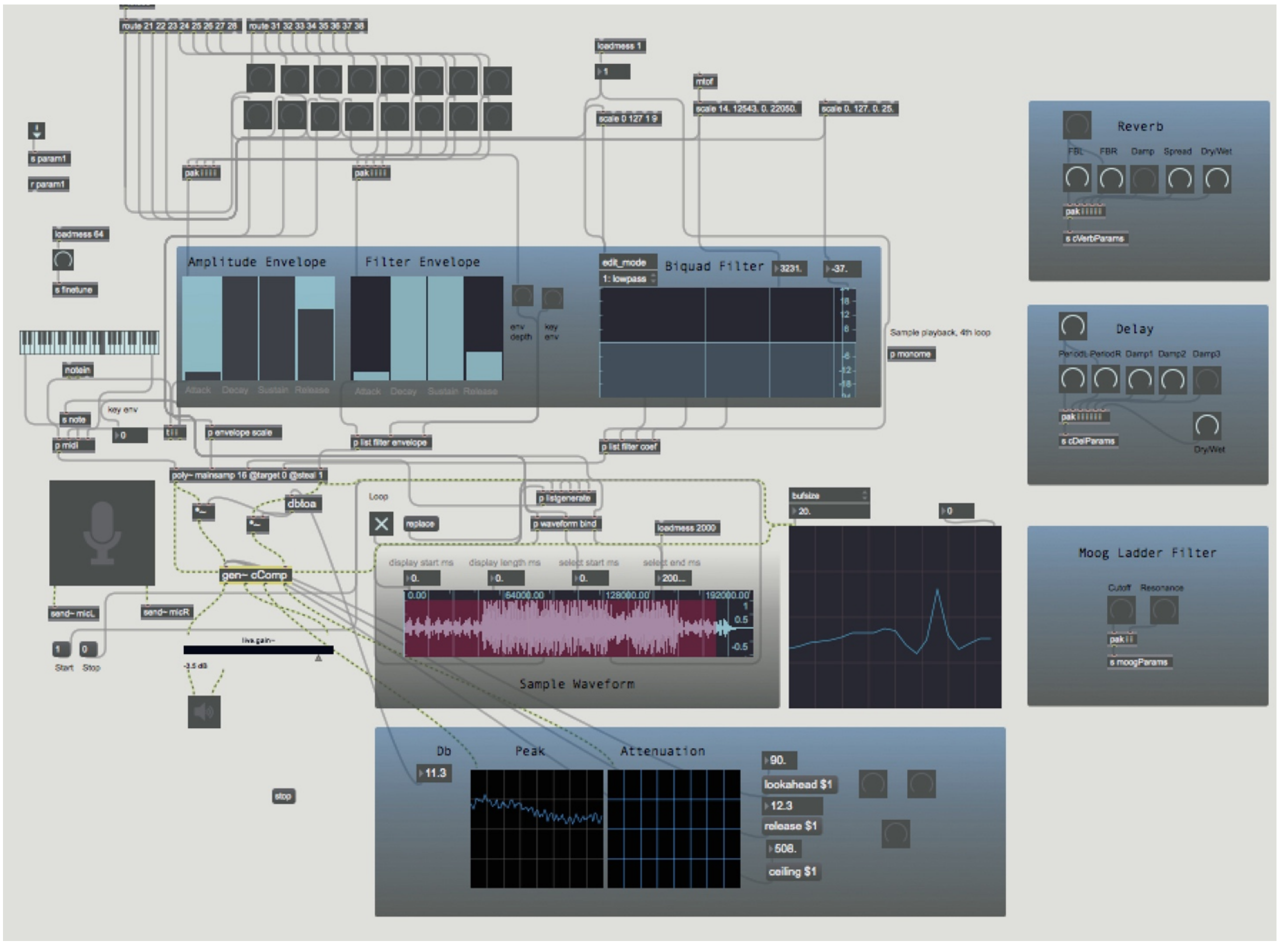

Figure 1 : Overview of Patch

II. Gen object in Max/MSP

To make the sampler more effective and also to extend my knowledge on processors, I studied Max's DSP object *gen~*. It brings a new layer to Max/MSP software. The main difference of ~gen is that concise text based expression language can be used rather than coding in GLSL. It can run on GPU therefore making the algorithms run at it's full performance with the capability of parallel programming. Gen runs on its own runtime and contains several objects that only works in its scope. As can be seen in figure 2, *~gen* works on sample level and capable of interpolating two signals. I built several delay and reverb units inside the *~gen* object and used those derivations to come up with a solution to de-attach the DSP modules from runtime. *History* operator allows feedback in the gen patcher through the insertion of single-sample delay hence it's commonly used in feedback delay networks. *Delay* operator in gen delays the signal by a certain amount of time specified by sample.

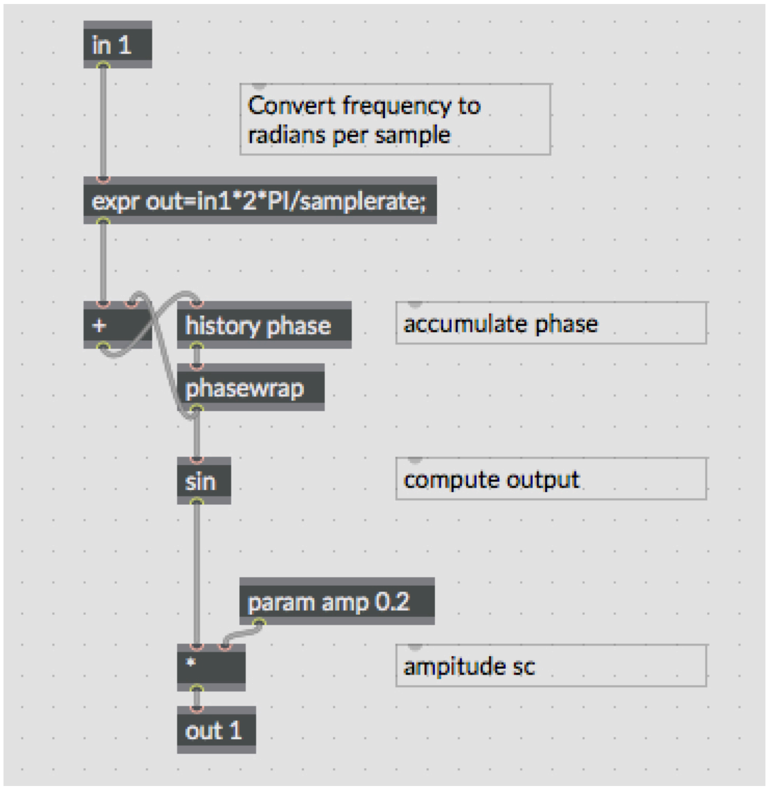

Figure 2: DSP in Gen

III. Modules

I. Delay

As seen in figure 3, inputs of the system are the individual audio channels and individual period frequencies, each frequency controls the delay ratio within it's channel therefore creating a sonic disjointedness. This effect could be interpreted as decreasing the repeated sample space. This change in delay time is referred to time-delay damping theory, damping parameters are mixed with delayed sample and stored in history operator for the next iteration. These damping parameters controls the wideness of the sonic spectrum.

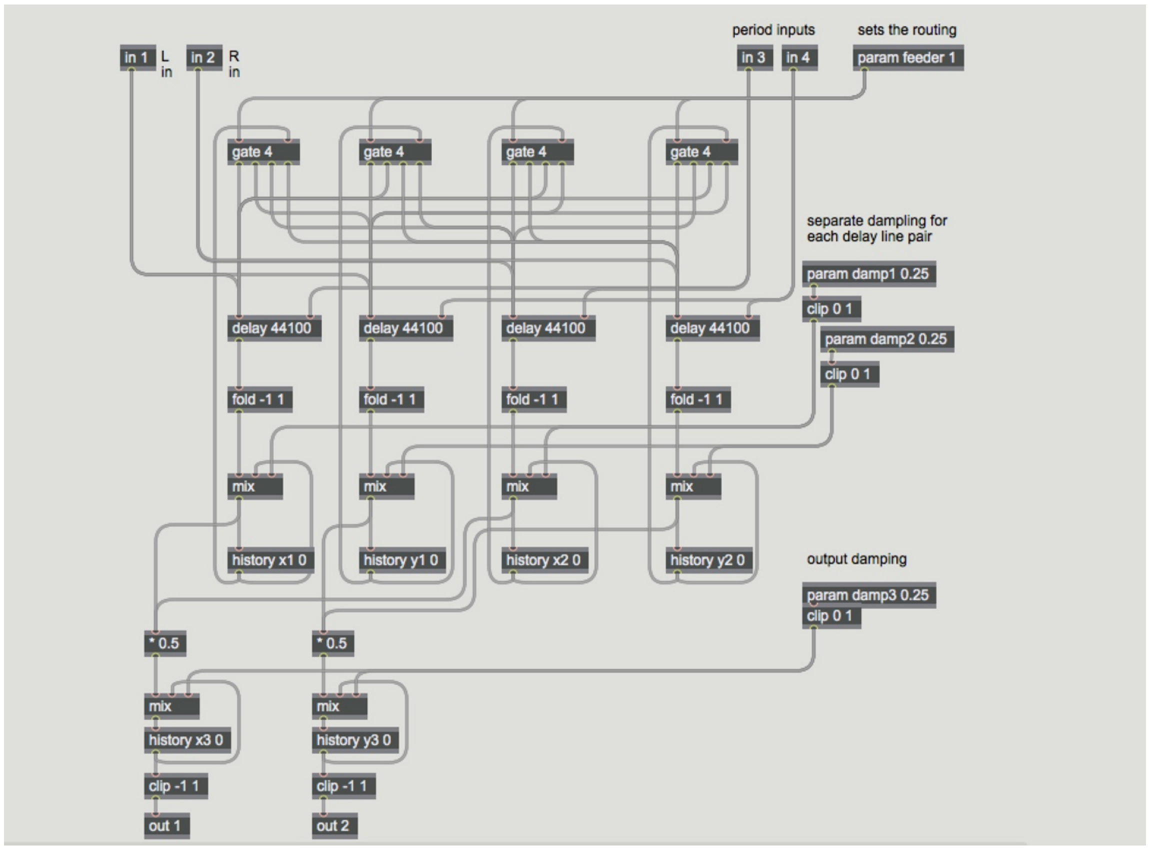

Figure 3: Feedback Delay Network

The pfft~ object is designed to simplify spectral audio processing using the Fast Fourier Transform (FFT). In addition to performing the FFT and the Inverse Fast Fourier Transform (IFFT), pfft~ manages the necessary signal windowing, overlapping and adding needed to create a real-time Short Term

Fourier Transform (STFT) analysis/re-synthesis system. Spectral delay is based on splitting the signal into a large number of frequency bands which can then be delayed individually as can be seen in figure4.

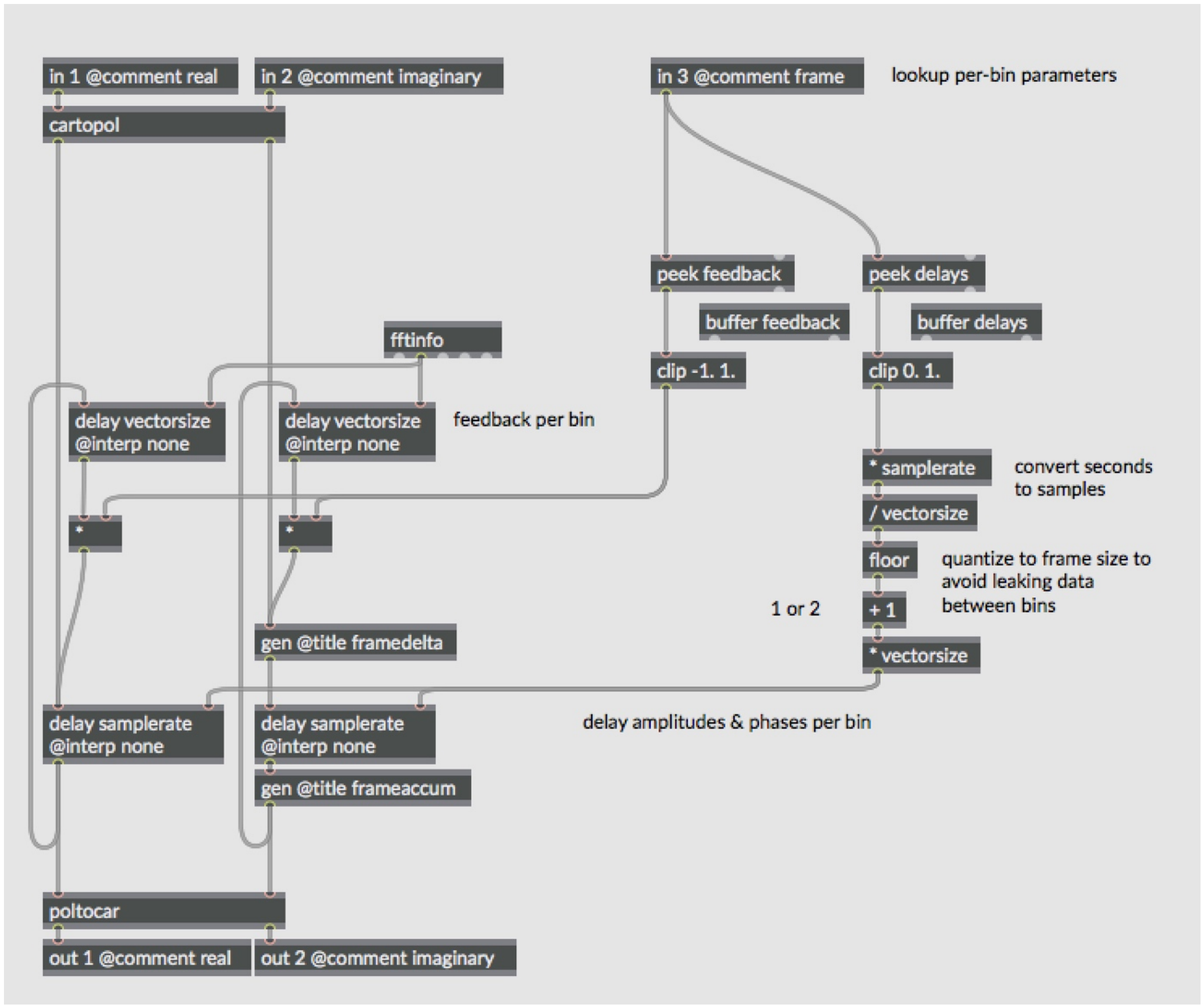

Figure 4: Spectral Delay

This method is effective, but it has several disadvantages. The FFT process is fairly computationally expensive, and so this method can place a heavy load on the computer's processor if real-time operation is needed. Also, FFTs introduce an inherent latency based on the number of frequency bands desired. This is undesirable in real-time applications where responsiveness is important.

The FFT method can also lead to quantization of the delay times allowed for each frequency, depending on how the effect is implemented.

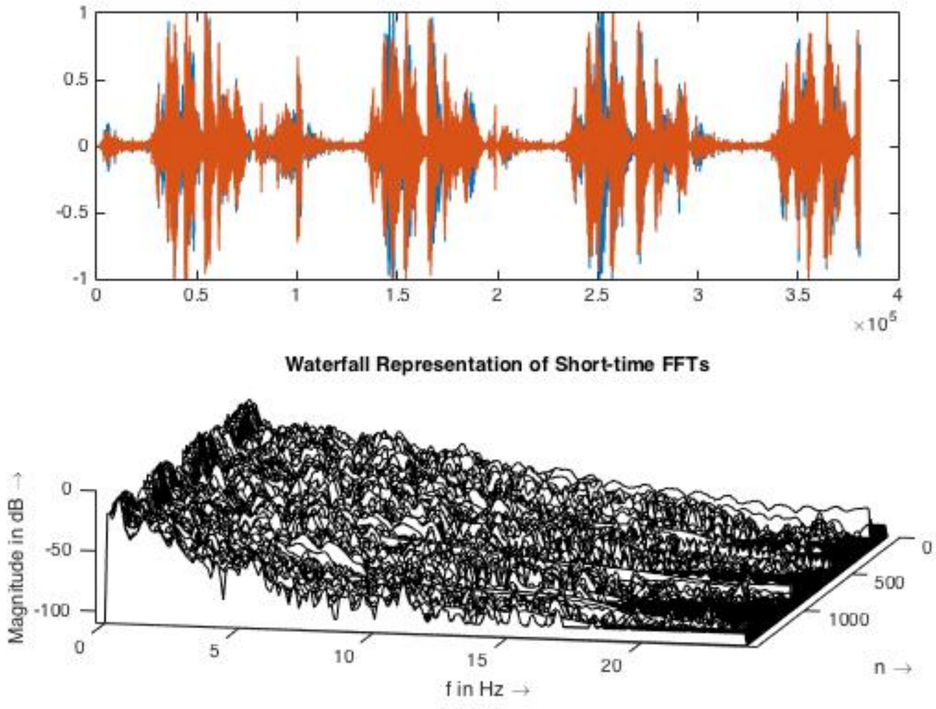

Figure 5 : Spectogram of sample

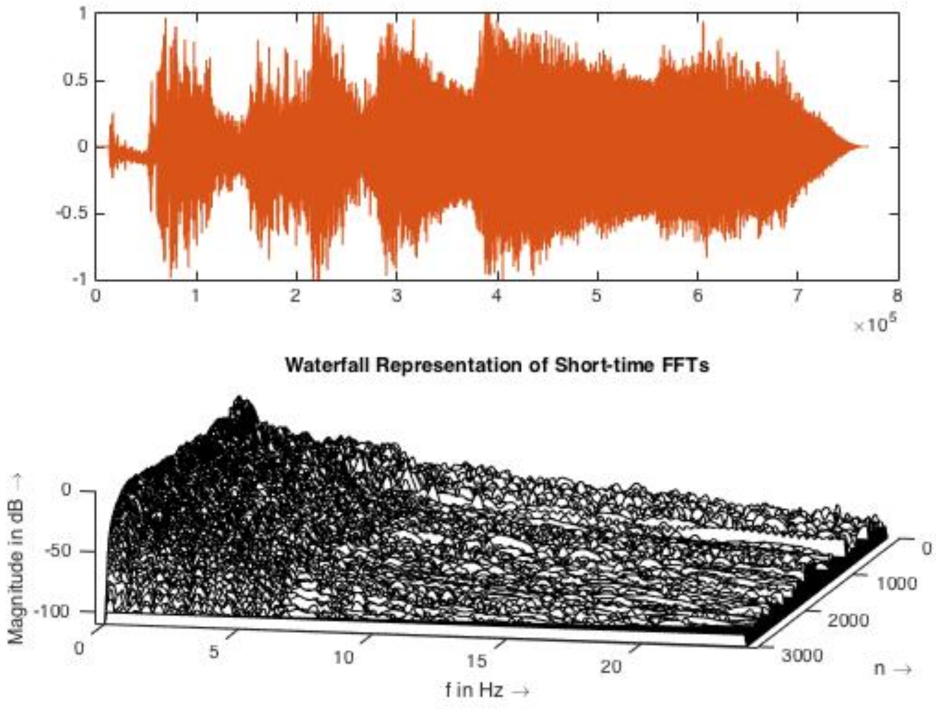

Figure 6: Spectogram of delayed sample

II. Reverb

Reverberation consists of two different approaches, algorithmic model uses an algorithmic model and convolution model uses the impulse response of a particular place. Algorithmic reverb is not based on any physical model therefore it can be used in an unusual way such as as an instrument instead of an effect processor. Reverb lengthens sound and shapes timbre. It can be used to smooth attack and sustaining some overtones more than others. It gives a strong sense of location of the sonic image of source's space and the listener's position. Reverb parameters can be stated as: echo density, coloration, smooth exponential decay, clarity, sparse early reverb, dense late reverb, variable diffusion on early reflections, control of pulsing or repetition. Most of academic researches has been done on creating a good artificial reverb with wide range parameterization. There is a generalized representation of reverberation structures, where you have separate matrices for the inputs, the feedback between delays and the outputs. With this representation any reverberator structure can be expressed. These approaches commonly referred as feedback delay networks. Feedback delay network is a term that describes parallel connected delay lines, connected by a specified kernel function and the output of the operation is fed back into the inputs.

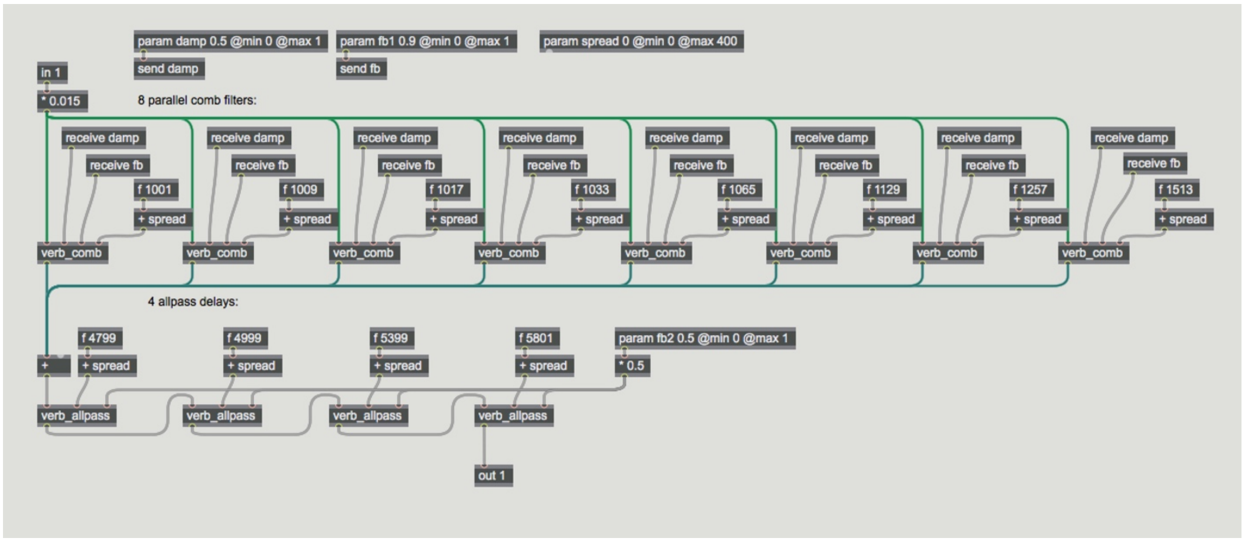

Figure 7: Reverb in Gen

However the complexity issues may arise with this operation, NxN matrix will take up to N^N computations in worst case and this will take N * $\log2_N$ to create matrix that is fully diffusive. In this case using an all-pass feedback delay network with fever branches but more density in each branch was a good idea. The designed model has 8 parallel comb filters connected to 4 sequential all-pass filter.

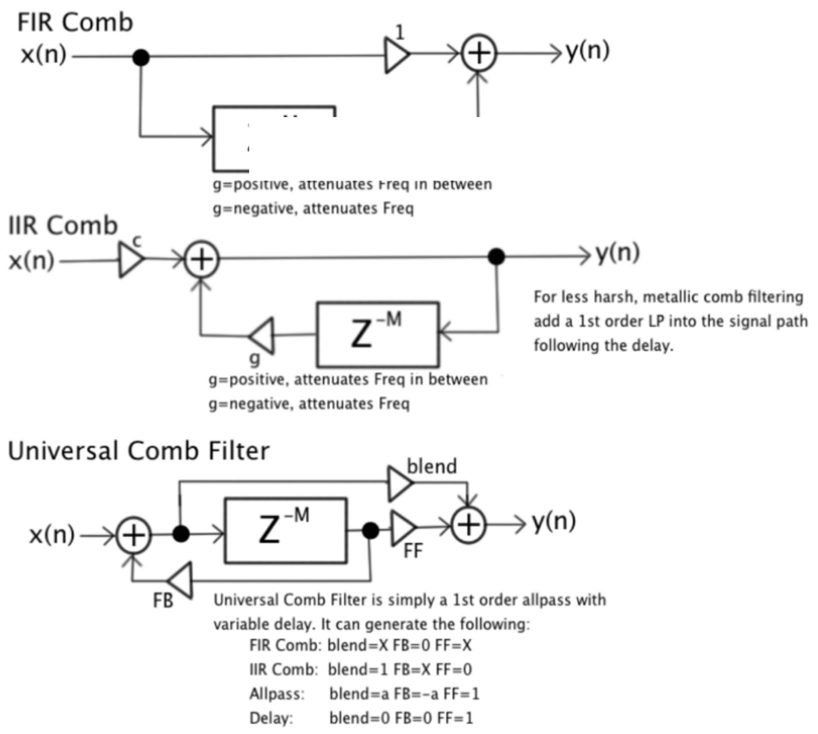
FIR Comb
x(n)
1
y(n)
g=positive, attenuates Freq in between
g=negative, attenuates Freq
IIR Comb
x(n)
c
y(n)
Z-M
g
For less harsh, metallic comb filtering add a 1st order LP into the signal path following the delay.
g=positive, attenuates Freq in between
g=negative, attenuates Freq
Universal Comb Filter
blend
x(n)
Z-M
y(n)
FF
FB
Universal Comb Filter is simply a 1st order allpass with variable delay. It can generate the following:
FIR Comb: blend=X FB=0 FF=X
IIR Comb: blend=1 FB=X FF=0
Allpass: blend=a FB=-a FF=1
Delay: blend=0 FB=0 FF=1

*FIGURE 8: **COMB FILTER***

This model has introduced by Manfred Schroeder, 4 comb filters in parallel and followed by chain of all-pass filters. James Moorer all-pass combs in parallel with low-pass filter to each one, progression of darkness gets darker as the time goes by with low-pass filters so it gave a sense of a real room with notion of early reflections.

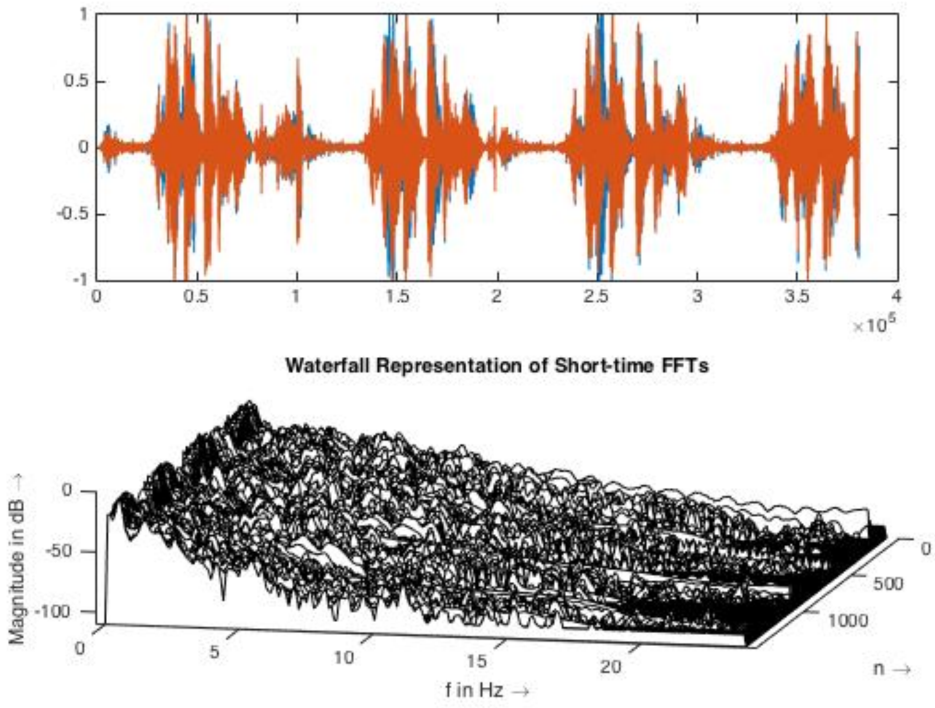

Figure 9: Spectogram of the sample

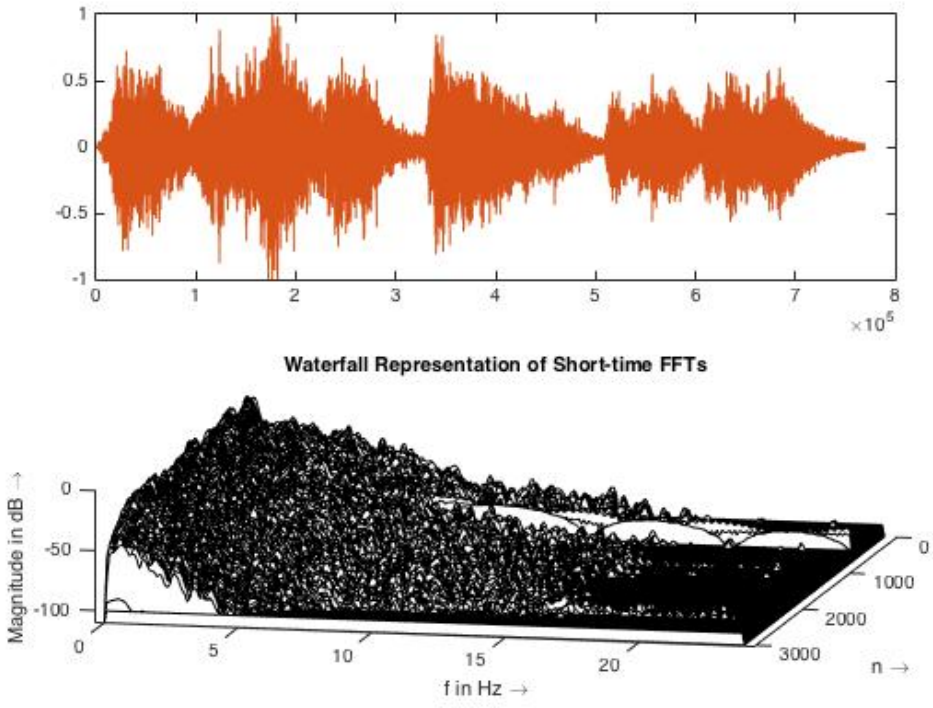

Figure 10: Spectogram of processed sample

III. COMPRESSOR

Compressors are widely used to smooth an incoming signal across a desired range of samples. Each time an incoming value changes, it begins a linear ramp to reach this value. To balance the output signal and get rid of glitch peaks that may occur I used a compressor that is built in gen. The ceiling, lookahead, gain and release are the parameters that effect the nature of the compression. Release sets how long the limiter will keep working after the signal goes below the threshold. Ceiling sets the maximum value (dB) at which the limiter will limit the signal. Lookahead sets the amount of time in milliseconds that the computer uses to look ahead and the gain parameter controls the input gain. The tone generator might produce a steady low-pitched drone, it's expected that with stable ceiling and release values the peaks will disappear since the compressor is turning the entire signal down.

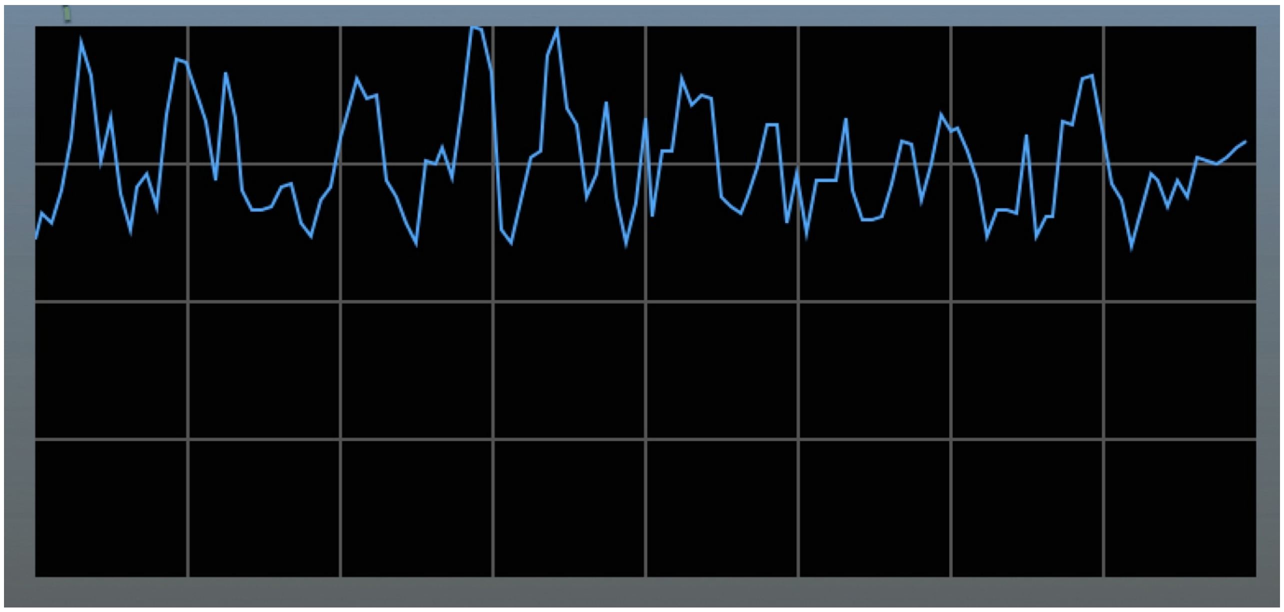

Figure 11: Scope of compressor

Configurations of Figure 11,

- 20 dB gain
- 28.3 sample lookahead,
- 429 ms release
- 30 db ceiling/threshold parameter.

IV. Moog Ladder Filter

The analog 'feel' of the Voltage-Controlled Filters (VCF) are based on nonlinearity which produce a distortion that adds a specific and musically-pleasing “warmth” to the timbre of the sound. Developed by Moog and used in early analog Moog synthesizers of 1960s. Moog's ladder filter is considered to be an important paradigm in real-time sound processing.

"Not only has it been recognized as a milestone in the history of electronic music, but in an attempt to reformulate the challenging solutions in its architecture in the digital domain, the various discrete-time models that were proposed in the last fifteen years to simulate the VCF have given rise to a curious thread of interesting realizations." (Valimaki).

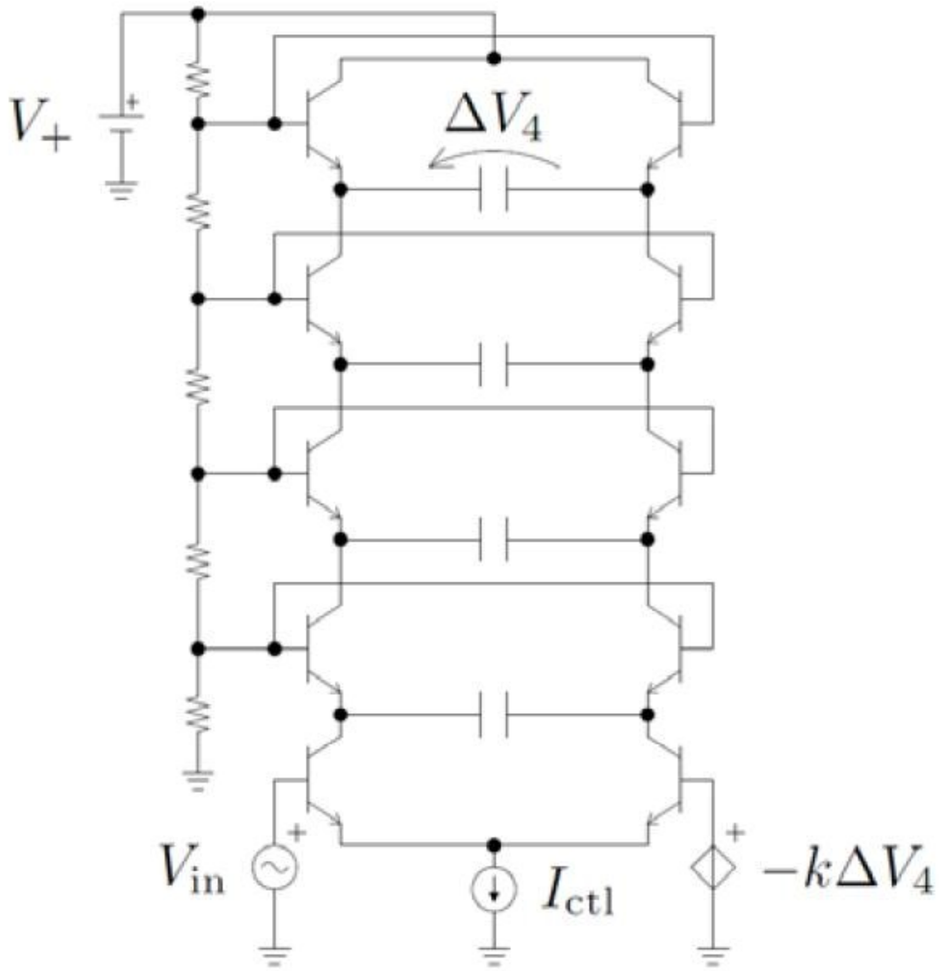

Figure 12 :Moog Ladder Filter

Moog Ladder Filter is an analog resonant low-pass filter design consisting of four stage ladder each composed of a differential pair of NPN transistors and a capacitor.

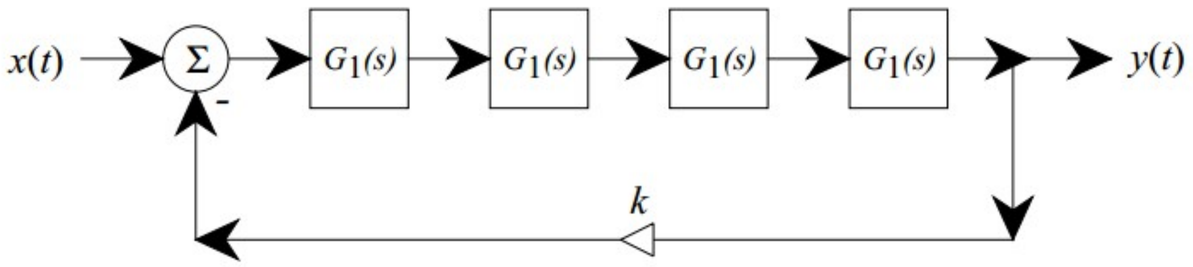

Figure 13: *The Moog VCF*

The overall transfer function with feedback as determined by Smith and Stilson is shown in figure 14.

$$H(s) \triangleq \frac{Y(s)}{X(s)} = \frac{G_1^4(s)}{1+kG_1^4(s)} = \frac{1}{k+(1+s/\omega_c)^4}$$

Figure 14: *Transfer function of the filter.*

Moog Ladder filter has two main features that needs to be converted to the digital domain to fully emulate the effects which tends to be a paradigm in virtual analog synthesis.

i) Nonlinear behaviour

ii) Two-dimensional continuous control, exerted by both parameter changes and the control signals.

Deriving from the previous researches made on digitizing the VCF, it's concluded that the most successful attempt was designed by Huovilainen who introduced a non-linear solution. The differential equation was constituted by analysis of the analog circuitry and the single state transfer function was constructed using Euler's method. The key of emulating the filter lies within feeding the input with an inverted output to produce resonance per iteration, so the implementation will be based on this hypothetical solution.

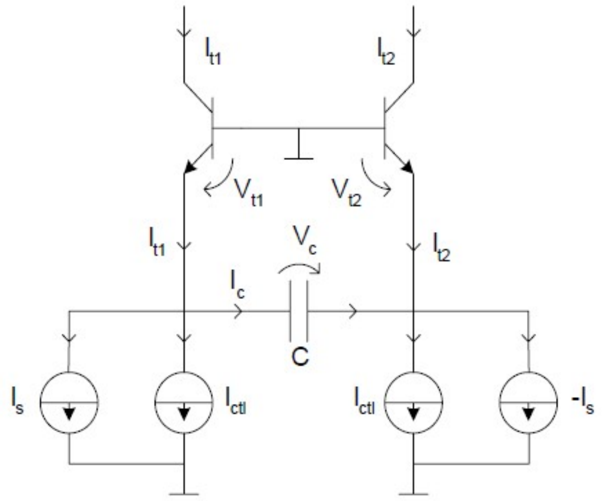

Figure 15: Single Stage of the Moog Ladder

$$\frac{dV_c}{dt} = \frac{I_{ctl}}{C}\left(\tanh\left(\frac{V_{in}}{2V_t}\right) - \tanh\left(\frac{V_c}{2V_t}\right)\right).$$

Figure 16: Differential equation for a single stage

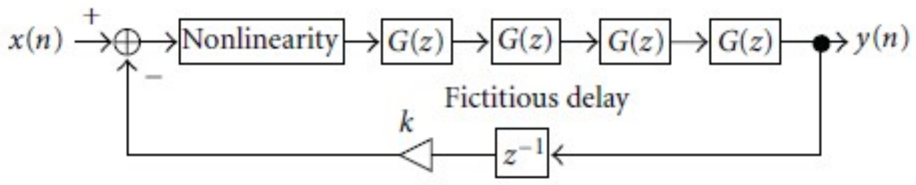

Figure 17: A simplified version of non-linear digital Moog filter

IV. Conclusion

A 16x8 hardware matrix implementation is still in progress, however the patch can be created dynamically and iteration in each layer is possible. CPU issues may occur during the allocation of individual samplers however the overall parameters will be controlled in a single patch therefore a possible chaining operations can be made on effective parameters. Output of the sampler can be recorded live and can be exported as an aiff or wav file. Moog ladder filter is implemented within the filter patch and embedded into the filter module instead of *biquad*.. The overall frequency response of the system can be seen in figure 18. As it can be observed, effect modules has an high impact on the overall sample, it's capable of producing rich soundscapes and different acoustic relations. The combination of parameters both within the delay and reverb changes the characteristics of sound dramatically.

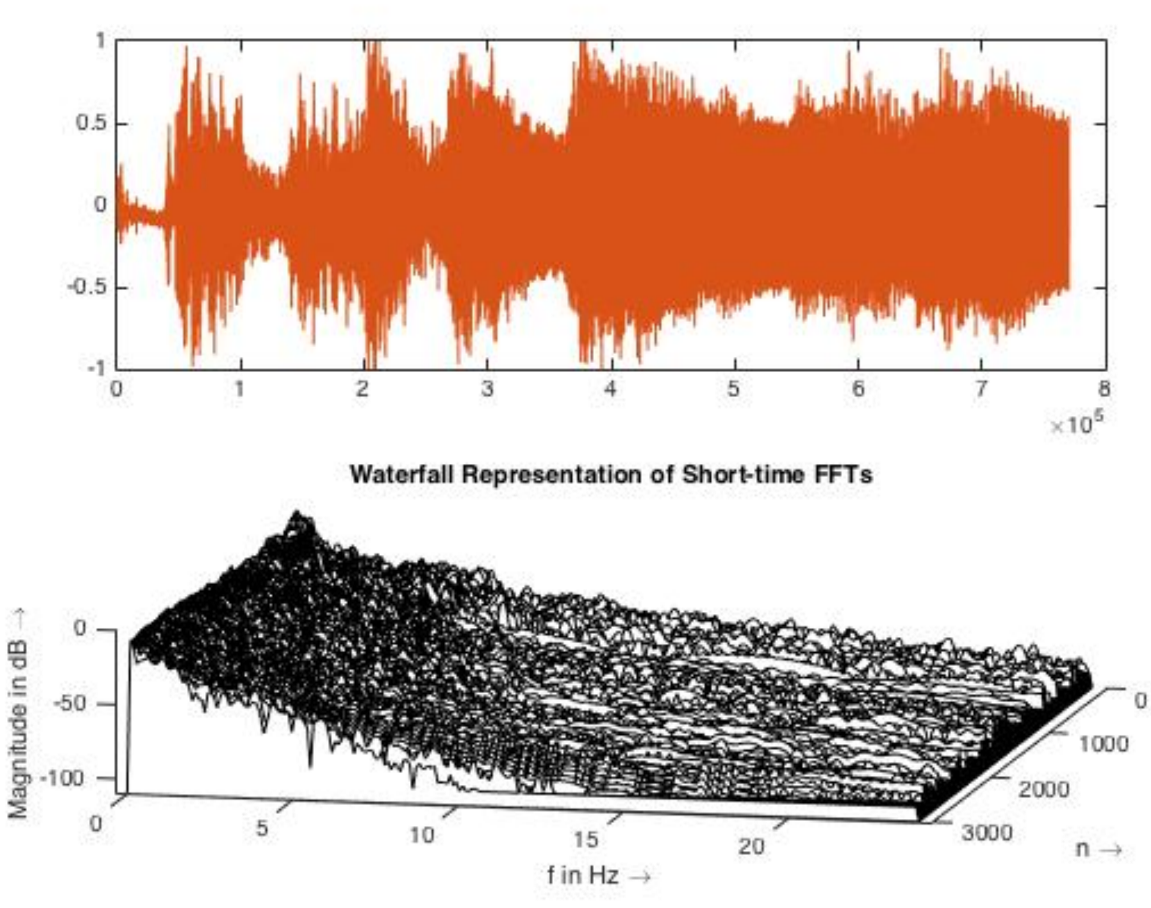

Figure 18: Spectogram of the output

V. References

[1] Alessandro Cipriani, Maurizio Giri. *Electronic Music and Sound Design* - Theory and Practice with Max and Msp Volume 1 (Second Edition) - Contemponet; Upd. for Max 6 edition (June 12, 2013)

[2] Alan V. Oppenheim *Discrete-Time Signal Processing (3rd Edition)* - (Prentice Hall

Signal Processing - Prentice Hall; 3 edition (August 28, 2009)

[3] Eric Lyon, David Zicarelli *Designing Audio Objects for Max/MSP and Pd*

(Computer Music and Digital Audio Series) - A-R Editions; Pap/Cdr edition (October 2012)

[4] Max7 documentation

[5] Monome Community, http://llllllll.co/

[6] R. Moog, "A voltage-controlled low-pass high-pass filter for audio signal processing," in *Proceedings of the 17th AES Convention*, New York, NY, USA, October 1965, preprint 413.

[7] J. Pakarinen, V. Valimaki, Vlimki, F. Fontana, V. Lazzarini, and J. S. Abel, "Recent advances in real- time musical effects, synthesis, and virtual analog models,"EURASIP J. Adv. Signal Process., 2011, article ID 940784.

[8] A. Huovilainen, "Nonlinear digital implementation of the Moog ladder filter," in *Proceedings of the International Conference on Digital Audio Effects*, Naples, Italy, October 2004.

[9] Elif Ecem Ozkan, Sabanci University senior design project report

#583 Design and Implementation of a Virtual Studio Technology Software Interface Progress Report-II, pp. 7-8, March/2015.

[10] DAFX: Digital Audio Effects by Udo Zölzer Wiley; 2 edition (April 18, 2011)

[11] https://ccrma.stanford.edu/~jos/pasp/Schroeder_Reverberators.html

[12] http://dspwiki.com/index.php?title=Reverberation

[13]Gen Export of Reverb Module

```
#include "cVerb.h"

namespace cVerb {




// global noise generator

Noise noise;

static const int GENLIB_LOOPCOUNT_BAIL = 100000;



// The State struct contains all the state and procedures for the gendsp
kernel

typedef struct State {

	CommonState __commonstate;

	Delay m_delay_24;

	Delay m_delay_15;

	Delay m_delay_13;

	Delay m_delay_23;

	Delay m_delay_9;

	Delay m_delay_17;

	Delay m_delay_21;

	Delay m_delay_19;

	Delay m_delay_22;

	Delay m_delay_7;

	Delay m_delay_11;
Delay m_delay_5;
int vectorsize;
int __exception;
t_sample m_spread_1;
t_sample m_history_20;
t_sample samplerate;
t_sample m_history_18;
t_sample m_damp_2;
t_sample m_history_16;
t_sample m_fb_3;
t_sample m_history_10;
t_sample m_history_8;
t_sample m_history_12;
t_sample m_history_14;
t_sample m_fb_4;
t_sample m_history_6;
// re-initialize all member variables;
inline void reset(t_param __sr, int __vs) {
      __exception = 0;
      vectorsize = __vs;
      samplerate = __sr;
      m_spread_1 = 0;
      m_damp_2 = 0.5;
      m_fb_3 = 0.5;
      m_fb_4 = 0.9;
      m_delay_5.reset("m_delay_5", 2000);
      m_history_6 = 0;
      m_delay_7.reset("m_delay_7", 2000);
      m_history_8 = 0;
```

```
        m_delay_9.reset("m_delay_9", 2000);

        m_history_10 = 0;

        m_delay_11.reset("m_delay_11", 2000);

        m_history_12 = 0;

        m_delay_13.reset("m_delay_13", 2000);

        m_history_14 = 0;

        m_delay_15.reset("m_delay_15", 2000);

        m_history_16 = 0;

        m_delay_17.reset("m_delay_17", 2000);

        m_history_18 = 0;

        m_delay_19.reset("m_delay_19", 2000);

        m_history_20 = 0;

        m_delay_21.reset("m_delay_21", 2000);

        m_delay_22.reset("m_delay_22", 2000);

        m_delay_23.reset("m_delay_23", 2000);

        m_delay_24.reset("m_delay_24", 2000);

        genlib_reset_complete(this);


    };

    // the signal processing routine;

    inline int perform(t_sample ** __ins, t_sample ** __outs, int __n)
{

        vectorsize = __n;

        const t_sample * __in1 = __ins[0];

        t_sample * __out1 = __outs[0];

        if (__exception) {

            return __exception;


        } else if (( (__in1 == 0) || (__out1 == 0) )) {

            __exception = GENLIB_ERR_NULL_BUFFER;
            return __exception;

        };
        t_sample mul_106 = (m_fb_3 * 0.5);
        t_sample add_92 = (225 + m_spread_1);
        t_sample add_94 = (341 + m_spread_1);
        t_sample add_104 = (441 + m_spread_1);
        t_sample add_90 = (556 + m_spread_1);
        t_sample damp_60 = m_damp_2;
        t_sample damp_59 = damp_60;
        t_sample damp_61 = damp_60;
        t_sample damp_62 = damp_60;
        t_sample damp_63 = damp_60;
        t_sample damp_64 = damp_60;
        t_sample damp_65 = damp_60;
        t_sample damp_66 = damp_60;
        t_sample add_97 = (1257 + m_spread_1);
        t_sample rsub_67 = (1 - damp_60);
        t_sample add_96 = (1513 + m_spread_1);
        t_sample rsub_117 = (1 - damp_59);
        t_sample add_98 = (1129 + m_spread_1);
        t_sample rsub_123 = (1 - damp_61);
        t_sample add_99 = (1065 + m_spread_1);
        t_sample rsub_140 = (1 - damp_62);
        t_sample add_100 = (1033 + m_spread_1);
        t_sample rsub_152 = (1 - damp_63);
        t_sample add_101 = (1017 + m_spread_1);
        t_sample rsub_162 = (1 - damp_64);
        t_sample add_102 = (1009 + m_spread_1);
```

```
t_sample rsub_171 = (1 - damp_65);
t_sample add_103 = (1001 + m_spread_1);
t_sample rsub_183 = (1 - damp_66);
// the main sample loop;
while ((__n--)) {
     const t_sample in1 = (*(__in1++));
     t_sample mul_108 = (in1 * 0.015);
     t_sample tap_74 = m_delay_5.read_linear(add_97);
     t_sample gen_83 = tap_74;
     t_sample mul_72 = (tap_74 * damp_60);
     t_sample mul_70 = (m_history_6 * rsub_67);
     t_sample add_71 = (mul_72 + mul_70);
     t_sample mul_68 = (add_71 * m_fb_4);
     t_sample add_75 = (mul_108 + mul_68);
     t_sample history_69_next_76 = add_71;
     t_sample tap_114 = m_delay_7.read_linear(add_96);
     t_sample gen_107 = tap_114;
     t_sample mul_115 = (tap_114 * damp_59);
     t_sample mul_116 = (m_history_8 * rsub_117);
     t_sample add_118 = (mul_115 + mul_116);
     t_sample mul_112 = (add_118 * m_fb_4);
     t_sample add_111 = (mul_108 + mul_112);
     t_sample history_69_next_110 = add_118;
     t_sample tap_128 = m_delay_9.read_linear(add_98);
     t_sample gen_82 = tap_128;
     t_sample mul_126 = (tap_128 * damp_61);
     t_sample mul_127 = (m_history_10 * rsub_123);
     t_sample add_129 = (mul_126 + mul_127);
     t_sample mul_125 = (add_129 * m_fb_4);
t_sample add_124 = (mul_108 + mul_125);
t_sample history_69_next_122 = add_129;
t_sample tap_136 = m_delay_11.read_linear(add_99);
t_sample gen_81 = tap_136;
t_sample mul_134 = (tap_136 * damp_62);
t_sample mul_135 = (m_history_12 * rsub_140);
t_sample add_139 = (mul_134 + mul_135);
t_sample mul_142 = (add_139 * m_fb_4);
t_sample add_141 = (mul_108 + mul_142);
t_sample history_69_next_137 = add_139;
t_sample tap_148 = m_delay_13.read_linear(add_100);
t_sample gen_80 = tap_148;
t_sample mul_146 = (tap_148 * damp_63);
t_sample mul_147 = (m_history_14 * rsub_152);
t_sample add_151 = (mul_146 + mul_147);
t_sample mul_154 = (add_151 * m_fb_4);
t_sample add_153 = (mul_108 + mul_154);
t_sample history_69_next_149 = add_151;
t_sample tap_160 = m_delay_15.read_linear(add_101);
t_sample gen_79 = tap_160;
t_sample mul_158 = (tap_160 * damp_64);
t_sample mul_159 = (m_history_16 * rsub_162);
t_sample add_164 = (mul_158 + mul_159);
t_sample mul_166 = (add_164 * m_fb_4);
t_sample add_165 = (mul_108 + mul_166);
t_sample history_69_next_161 = add_164;
t_sample tap_173 = m_delay_17.read_linear(add_102);
t_sample gen_78 = tap_173;
t_sample mul_170 = (tap_173 * damp_65);
```

```
                t_sample mul_172 = (m_history_18 * rsub_171);
                t_sample add_177 = (mul_170 + mul_172);
                t_sample mul_178 = (add_177 * m_fb_4);
                t_sample add_176 = (mul_108 + mul_178);
                t_sample history_69_next_174 = add_177;
                t_sample tap_185 = m_delay_19.read_linear(add_103);
                t_sample gen_77 = tap_185;
                t_sample mul_182 = (tap_185 * damp_66);
                t_sample mul_184 = (m_history_20 * rsub_183);
                t_sample add_188 = (mul_182 + mul_184);
                t_sample mul_190 = (add_188 * m_fb_4);
                t_sample add_189 = (mul_108 + mul_190);
                t_sample history_69_next_186 = add_188;
                t_sample add_105 = ((((((((gen_77 + gen_78) + gen_79) +
gen_80) + gen_81) + gen_82) + gen_107) + gen_83) + 0);
                t_sample tap_88 = m_delay_21.read_linear(add_90);
                t_sample sub_84 = (add_105 - tap_88);
                t_sample mul_86 = (tap_88 * mul_106);
                t_sample add_85 = (add_105 + mul_86);
                t_sample tap_196 = m_delay_22.read_linear(add_104);
                t_sample sub_197 = (sub_84 - tap_196);
                t_sample mul_195 = (tap_196 * mul_106);
                t_sample add_194 = (sub_84 + mul_195);
                t_sample tap_202 = m_delay_23.read_linear(add_94);
                t_sample sub_203 = (sub_197 - tap_202);
                t_sample mul_201 = (tap_202 * mul_106);
                t_sample add_200 = (sub_197 + mul_201);
                t_sample tap_208 = m_delay_24.read_linear(add_92);
                t_sample sub_209 = (sub_203 - tap_208);
                t_sample mul_207 = (tap_208 * mul_106);
t_sample add_206 = (sub_203 + mul_207);

t_sample out1 = sub_209;

m_delay_5.write(add_75);

m_delay_24.write(add_206);

m_delay_23.write(add_200);

m_delay_22.write(add_194);

m_delay_21.write(add_85);

m_history_20 = history_69_next_186;

m_delay_19.write(add_189);

m_history_18 = history_69_next_174;

m_delay_17.write(add_176);

m_history_16 = history_69_next_161;

m_delay_15.write(add_165);

m_history_14 = history_69_next_149;

m_delay_13.write(add_153);

m_history_12 = history_69_next_137;

m_delay_11.write(add_141);

m_history_10 = history_69_next_122;

m_delay_9.write(add_124);

m_history_8 = history_69_next_110;

m_delay_7.write(add_111);

m_history_6 = history_69_next_76;

m_delay_5.step();

m_delay_7.step();

m_delay_9.step();

m_delay_11.step();

m_delay_13.step();

m_delay_15.step();

m_delay_17.step();
```

```
                m_delay_19.step();
                m_delay_21.step();
                m_delay_22.step();
                m_delay_23.step();
                m_delay_24.step();
                // assign results to output buffer;
                (*(__out1++)) = out1;

            };
            return __exception;

        };
        inline void set_spread(t_param _value) {
            m_spread_1 = (_value < 0 ? 0 : (_value > 400 ? 400 : _value));
        };
        inline void set_damp(t_param _value) {
            m_damp_2 = (_value < 0 ? 0 : (_value > 1 ? 1 : _value));
        };
        inline void set_fb2(t_param _value) {
            m_fb_3 = (_value < 0 ? 0 : (_value > 1 ? 1 : _value));
        };
        inline void set_fb1(t_param _value) {
            m_fb_4 = (_value < 0 ? 0 : (_value > 1 ? 1 : _value));
        };

} State;


///
```

```
///   Configuration for the genlib API

///


/// Number of signal inputs and outputs


int gen_kernel_numins = 1;

int gen_kernel_numouts = 1;


int num_inputs() { return gen_kernel_numins; }

int num_outputs() { return gen_kernel_numouts; }

int num_params() { return 4; }


/// Assistive lables for the signal inputs and outputs


const char * gen_kernel_innames[] = { "in1" };

const char * gen_kernel_outnames[] = { "out1" };


/// Invoke the signal process of a State object


int perform(CommonState *cself, t_sample **ins, long numins, t_sample
**outs, long numouts, long n) {

      State * self = (State *)cself;

      return self->perform(ins, outs, n);

}


/// Reset all parameters and stateful operators of a State object


void reset(CommonState *cself) {

      State * self = (State *)cself;

      self->reset(cself->sr, cself->vs);
}


/// Set a parameter of a State object


void setparameter(CommonState *cself, long index, t_param value, void
*ref) {
      State * self = (State *)cself;
      switch (index) {
            case 0: self->set_damp(value); break;
            case 1: self->set_fb1(value); break;
            case 2: self->set_fb2(value); break;
            case 3: self->set_spread(value); break;

            default: break;
      }
}


/// Get the value of a parameter of a State object


void getparameter(CommonState *cself, long index, t_param *value) {
      State *self = (State *)cself;
      switch (index) {
            case 0: *value = self->m_damp_2; break;
            case 1: *value = self->m_fb_4; break;
            case 2: *value = self->m_fb_3; break;
            case 3: *value = self->m_spread_1; break;

            default: break;
      }
}
```

```
/// Get the name of a parameter of a State object

const char *getparametername(CommonState *cself, long index) {
	if (index >= 0 && index < cself->numparams) {
		return cself->params[index].name;
	}
	return 0;
}

/// Get the minimum value of a parameter of a State object

t_param getparametermin(CommonState *cself, long index) {
	if (index >= 0 && index < cself->numparams) {
		return cself->params[index].outputmin;
	}
	return 0;
}
t_param getparametermax(CommonState *cself, long index) {
	if (index >= 0 && index < cself->numparams) {
		return cself->params[index].outputmax;
	}
	return 0;
}
char getparameterhasminmax(CommonState *cself, long index) {
	if (index >= 0 && index < cself->numparams) {
		return cself->params[index].hasminmax;
	}
	return 0;
}

/// Get the units of a parameter of a State object

const char *getparameterunits(CommonState *cself, long index) {
      if (index >= 0 && index < cself->numparams) {
            return cself->params[index].units;
      }
      return 0;
}

/// Get the size of the state of all parameters of a State object

size_t getstatesize(CommonState *cself) {
      return genlib_getstatesize(cself, &getparameter);
}

/// Get the state of all parameters of a State object

short getstate(CommonState *cself, char *state) {
      return genlib_getstate(cself, state, &getparameter);
}
/// set the state of all parameters of a State object

short setstate(CommonState *cself, const char *state) {
      return genlib_setstate(cself, state, &setparameter);
}

///  Allocate  and  configure  a  new  State  object  and  it's  internal
CommonState:
```

```
void * create(t_param sr, long vs) {

	State *self = new State;

	self->reset(sr, vs);

	ParamInfo *pi;

	self->__commonstate.inputnames = gen_kernel_innames;

	self->__commonstate.outputnames = gen_kernel_outnames;

	self->__commonstate.numins = gen_kernel_numins;

	self->__commonstate.numouts = gen_kernel_numouts;

	self->__commonstate.sr = sr;

	self->__commonstate.vs = vs;

	self->__commonstate.params = (ParamInfo *)genlib_sysmem_newptr(4 *
sizeof(ParamInfo));

	self->__commonstate.numparams = 4;

	// initialize parameter 0 ("m_damp_2")

	pi = self->__commonstate.params + 0;

	pi->name = "damp";

	pi->paramtype = GENLIB_PARAMTYPE_FLOAT;

	pi->defaultvalue = self->m_damp_2;

	pi->defaultref = 0;

	pi->hasinputminmax = false;

	pi->inputmin = 0;

	pi->inputmax = 1;

	pi->hasminmax = true;

	pi->outputmin = 0;

	pi->outputmax = 1;

	pi->exp = 0;

	pi->units = "";		// no units defined

	// initialize parameter 1 ("m_fb_4")

	pi = self->__commonstate.params + 1;
```

```
pi->name = "fb1";

pi->paramtype = GENLIB_PARAMTYPE_FLOAT;

pi->defaultvalue = self->m_fb_4;

pi->defaultref = 0;

pi->hasinputminmax = false;

pi->inputmin = 0;

pi->inputmax = 1;

pi->hasminmax = true;

pi->outputmin = 0;

pi->outputmax = 1;

pi->exp = 0;

pi->units = "";       // no units defined

// initialize parameter 2 ("m_fb_3")

pi = self->__commonstate.params + 2;

pi->name = "fb2";

pi->paramtype = GENLIB_PARAMTYPE_FLOAT;

pi->defaultvalue = self->m_fb_3;

pi->defaultref = 0;

pi->hasinputminmax = false;

pi->inputmin = 0;

pi->inputmax = 1;

pi->hasminmax = true;

pi->outputmin = 0;

pi->outputmax = 1;

pi->exp = 0;

pi->units = "";       // no units defined

// initialize parameter 3 ("m_spread_1")

pi = self->__commonstate.params + 3;

pi->name = "spread";
```

```
	pi->paramtype = GENLIB_PARAMTYPE_FLOAT;

	pi->defaultvalue = self->m_spread_1;

	pi->defaultref = 0;

	pi->hasinputminmax = false;

	pi->inputmin = 0;

	pi->inputmax = 1;

	pi->hasminmax = true;

	pi->outputmin = 0;

	pi->outputmax = 400;

	pi->exp = 0;

	pi->units = "";		// no units defined


	return self;

}


/// Release all resources and memory used by a State object:


void destroy(CommonState *cself) {

	State * self = (State *)cself;

	genlib_sysmem_freeptr(cself->params);


	delete self;

}




} // cVerb::
```

[14]Gen Export of Delay Module

```
#include "cDelay_gendsp.h"


namespace cDelay_gendsp {



// global noise generator

Noise noise;

static const int GENLIB_LOOPCOUNT_BAIL = 100000;



// The State struct contains all the state and procedures for the gendsp
kernel

typedef struct State {

	CommonState __commonstate;

	Delay m_delay_10;

	Delay m_delay_9;

	Delay m_delay_8;

	Delay m_delay_7;

	int __exception;

	int vectorsize;

	t_sample m_damp_11;

	t_sample m_damp_12;

	t_sample m_feeder_13;

	t_sample m_x_6;

	t_sample m_x_4;

	t_sample samplerate;

	t_sample m_x_5;
	t_sample m_y_1;
	t_sample m_y_3;
	t_sample m_y_2;
	t_sample m_damp_14;
	// re-initialize all member variables;
	inline void reset(t_param __sr, int __vs) {
		__exception = 0;
		vectorsize = __vs;
		samplerate = __sr;
		m_y_1 = 0;
		m_y_2 = 0;
		m_y_3 = 0;
		m_x_4 = 0;
		m_x_5 = 0;
		m_x_6 = 0;
		m_delay_7.reset("m_delay_7", 44100);
		m_delay_8.reset("m_delay_8", 44100);
		m_delay_9.reset("m_delay_9", 44100);
		m_delay_10.reset("m_delay_10", 44100);
		m_damp_11 = 0.25;
		m_damp_12 = 0.25;
		m_feeder_13 = 1;
		m_damp_14 = 0.25;
		genlib_reset_complete(this);

	};
	// the signal processing routine;
	inline int perform(t_sample ** __ins, t_sample ** __outs, int __n)
{
		vectorsize = __n;
```

```
        const t_sample * __in1 = __ins[0];
        const t_sample * __in2 = __ins[1];
        const t_sample * __in3 = __ins[2];
        const t_sample * __in4 = __ins[3];
        t_sample * __out1 = __outs[0];
        t_sample * __out2 = __outs[1];
        if (__exception) {
            return __exception;

        } else if (( (__in1 == 0) || (__in2 == 0) || (__in3 == 0) ||
(__in4 == 0) || (__out1 == 0) || (__out2 == 0) )) {
            __exception = GENLIB_ERR_NULL_BUFFER;
            return __exception;

        };
        t_sample clamp_245 = ((m_damp_12 <= 0) ? 0 : ((m_damp_12 >= 1)
? 1 : m_damp_12));
        t_sample clamp_231 = ((m_damp_14 <= 0) ? 0 : ((m_damp_14 >= 1)
? 1 : m_damp_14));
        t_sample clamp_230 = ((m_damp_11 <= 0) ? 0 : ((m_damp_11 >= 1)
? 1 : m_damp_11));
        t_sample choice_15 = int(m_feeder_13);
        t_sample choice_16 = int(m_feeder_13);
        t_sample choice_17 = int(m_feeder_13);
        t_sample choice_18 = int(m_feeder_13);
        // the main sample loop;
        while ((__n--)) {
            const t_sample in1 = (*(__in1++));
            const t_sample in2 = (*(__in2++));
            const t_sample in3 = (*(__in3++));
            const t_sample in4 = (*(__in4++));
```

```
                    t_sample clamp_246 = ((m_x_4 <= -1) ? -1 : ((m_x_4 >= 1)
? 1 : m_x_4));
                    t_sample out1 = clamp_246;
                    t_sample clamp_244 = ((m_y_1 <= -1) ? -1 : ((m_y_1 >= 1)
? 1 : m_y_1));
                    t_sample out2 = clamp_244;
                    t_sample gate_223 = (((choice_15 >= 1) && (choice_15 <
2)) ? m_y_2 : 0);
                    t_sample gate_224 = (((choice_15 >= 2) && (choice_15 <
3)) ? m_y_2 : 0);
                    t_sample gate_225 = (((choice_15 >= 3) && (choice_15 <
4)) ? m_y_2 : 0);
                    t_sample gate_226 = ((choice_15 >= 4) ? m_y_2 : 0);
                    t_sample gate_219 = (((choice_16 >= 1) && (choice_16 <
2)) ? m_x_6 : 0);
                    t_sample gate_220 = (((choice_16 >= 2) && (choice_16 <
3)) ? m_x_6 : 0);
                    t_sample gate_221 = (((choice_16 >= 3) && (choice_16 <
4)) ? m_x_6 : 0);
                    t_sample gate_222 = ((choice_16 >= 4) ? m_x_6 : 0);
                    t_sample gate_215 = (((choice_17 >= 1) && (choice_17 <
2)) ? m_x_5 : 0);
                    t_sample gate_216 = (((choice_17 >= 2) && (choice_17 <
3)) ? m_x_5 : 0);
                    t_sample gate_217 = (((choice_17 >= 3) && (choice_17 <
4)) ? m_x_5 : 0);
                    t_sample gate_218 = ((choice_17 >= 4) ? m_x_5 : 0);
                    t_sample gate_211 = (((choice_18 >= 1) && (choice_18 <
2)) ? m_y_3 : 0);
                    t_sample gate_212 = (((choice_18 >= 2) && (choice_18 <
3)) ? m_y_3 : 0);
                    t_sample gate_213 = (((choice_18 >= 3) && (choice_18 <
4)) ? m_y_3 : 0);
                    t_sample gate_214 = ((choice_18 >= 4) ? m_y_3 : 0);
                    t_sample tap_251 = m_delay_10.read_linear(in3);
```

```
                t_sample fold_236 = fold(tap_251, -1, 1);
                t_sample  mix_366  =  (fold_236  +  (clamp_245  *  (m_x_6  -
fold_236)));
                t_sample mix_249 = mix_366;
                t_sample tap_242 = m_delay_9.read_linear(in3);
                t_sample fold_234 = fold(tap_242, -1, 1);
                t_sample  mix_367  =  (fold_234  +  (clamp_245  *  (m_x_5  -
fold_234)));
                t_sample mix_240 = mix_367;
                t_sample mul_228 = ((mix_240 + mix_249) * 0.5);
                t_sample  mix_368  =  (mul_228  +  (clamp_230  *  (m_x_4  -
mul_228)));
                t_sample mix_232 = mix_368;
                t_sample tap_239 = m_delay_8.read_linear(in4);
                t_sample fold_233 = fold(tap_239, -1, 1);
                t_sample  mix_369  =  (fold_233  +  (clamp_231  *  (m_y_3  -
fold_233)));
                t_sample mix_237 = mix_369;
                t_sample tap_248 = m_delay_7.read_linear(in4);
                t_sample fold_235 = fold(tap_248, -1, 1);
                t_sample  mix_370  =  (fold_235  +  (clamp_231  *  (m_y_2  -
fold_235)));
                t_sample mix_243 = mix_370;
                t_sample mul_227 = ((mix_237 + mix_243) * 0.5);
                t_sample  mix_371  =  (mul_227  +  (clamp_230  *  (m_y_1  -
mul_227)));
                t_sample mix_229 = mix_371;
                t_sample x1_next_252 = mix_249;
                t_sample x2_next_253 = mix_240;
                t_sample x3_next_254 = mix_232;
                t_sample y2_next_255 = mix_237;
                t_sample y1_next_256 = mix_243;
                t_sample y3_next_257 = mix_229;

                m_delay_10.write(((((gate_213 + gate_217) + gate_219) +
in1) + gate_224));

                m_delay_9.write(((((gate_212 + gate_215) + gate_221) +
gate_226) + in2));

                m_delay_8.write(((((gate_211 + gate_216) + gate_222) +
gate_225) + in2));

                m_delay_7.write(((((gate_214 + gate_218) + gate_220) +
in1) + gate_223));

                m_x_6 = x1_next_252;

                m_x_5 = x2_next_253;

                m_x_4 = x3_next_254;

                m_y_3 = y2_next_255;

                m_y_2 = y1_next_256;

                m_y_1 = y3_next_257;

                m_delay_7.step();

                m_delay_8.step();

                m_delay_9.step();

                m_delay_10.step();

                // assign results to output buffer;

                (*(__out1++)) = out1;

                (*(__out2++)) = out2;


            };

            return __exception;


        };

        inline void set_damp3(t_param _value) {

            m_damp_11 = (_value < 0 ? 0 : (_value > 1 ? 1 : _value));

        };

        inline void set_damp1(t_param _value) {
            m_damp_12 = (_value < 0 ? 0 : (_value > 1 ? 1 : _value));
        };
        inline void set_feeder(t_param _value) {
            m_feeder_13 = (_value < 0 ? 0 : (_value > 1 ? 1 : _value));
        };
        inline void set_damp2(t_param _value) {
            m_damp_14 = (_value < 0 ? 0 : (_value > 1 ? 1 : _value));
        };

} State;



///
///    Configuration for the genlib API
///

/// Number of signal inputs and outputs

int gen_kernel_numins = 4;
int gen_kernel_numouts = 2;

int num_inputs() { return gen_kernel_numins; }
int num_outputs() { return gen_kernel_numouts; }
int num_params() { return 4; }

/// Assistive lables for the signal inputs and outputs

const char * gen_kernel_innames[] = { "in1", "in2", "in3", "in4" };
const char * gen_kernel_outnames[] = { "out1", "out2" };
```

```
/// Invoke the signal process of a State object


int perform(CommonState *cself, t_sample **ins, long numins, t_sample
**outs, long numouts, long n) {
	State * self = (State *)cself;
	return self->perform(ins, outs, n);
}


/// Reset all parameters and stateful operators of a State object


void reset(CommonState *cself) {
	State * self = (State *)cself;
	self->reset(cself->sr, cself->vs);
}


/// Set a parameter of a State object


void setparameter(CommonState *cself, long index, t_param value, void
*ref) {
	State * self = (State *)cself;
	switch (index) {
		case 0: self->set_damp1(value); break;
		case 1: self->set_damp2(value); break;
		case 2: self->set_damp3(value); break;
		case 3: self->set_feeder(value); break;


		default: break;
	}
}
```

```
/// Get the value of a parameter of a State object

void getparameter(CommonState *cself, long index, t_param *value) {
      State *self = (State *)cself;
      switch (index) {
            case 0: *value = self->m_damp_12; break;
            case 1: *value = self->m_damp_14; break;
            case 2: *value = self->m_damp_11; break;
            case 3: *value = self->m_feeder_13; break;

            default: break;
      }
}

/// Get the name of a parameter of a State object

const char *getparametername(CommonState *cself, long index) {
      if (index >= 0 && index < cself->numparams) {
            return cself->params[index].name;
      }
      return 0;
}

/// Get the minimum value of a parameter of a State object

t_param getparametermin(CommonState *cself, long index) {
      if (index >= 0 && index < cself->numparams) {
            return cself->params[index].outputmin;
      }
      return 0;
}

/// Get the maximum value of a parameter of a State object

t_param getparametermax(CommonState *cself, long index) {
      if (index >= 0 && index < cself->numparams) {
            return cself->params[index].outputmax;
      }
      return 0;
}

/// Get parameter of a State object has a minimum and maximum value

char getparameterhasminmax(CommonState *cself, long index) {
      if (index >= 0 && index < cself->numparams) {
            return cself->params[index].hasminmax;
      }
      return 0;
}

/// Get the units of a parameter of a State object

const char *getparameterunits(CommonState *cself, long index) {
      if (index >= 0 && index < cself->numparams) {
            return cself->params[index].units;
      }
      return 0;
```

```
}


/// Get the size of the state of all parameters of a State object


size_t getstatesize(CommonState *cself) {
	return genlib_getstatesize(cself, &getparameter);
}


/// Get the state of all parameters of a State object


short getstate(CommonState *cself, char *state) {
	return genlib_getstate(cself, state, &getparameter);
}


/// set the state of all parameters of a State object


short setstate(CommonState *cself, const char *state) {
	return genlib_setstate(cself, state, &setparameter);
}


///  Allocate  and  configure  a  new  State  object  and  it's  internal
CommonState:


void * create(t_param sr, long vs) {
	State *self = new State;
	self->reset(sr, vs);
	ParamInfo *pi;
	self->__commonstate.inputnames = gen_kernel_innames;
	self->__commonstate.outputnames = gen_kernel_outnames;
	self->__commonstate.numins = gen_kernel_numins;
```

```
      self->__commonstate.numouts = gen_kernel_numouts;

      self->__commonstate.sr = sr;

      self->__commonstate.vs = vs;

      self->__commonstate.params = (ParamInfo *)genlib_sysmem_newptr(4 *
sizeof(ParamInfo));

      self->__commonstate.numparams = 4;

      // initialize parameter 0 ("m_damp_12")

      pi = self->__commonstate.params + 0;

      pi->name = "damp1";

      pi->paramtype = GENLIB_PARAMTYPE_FLOAT;

      pi->defaultvalue = self->m_damp_12;

      pi->defaultref = 0;

      pi->hasinputminmax = false;

      pi->inputmin = 0;

      pi->inputmax = 1;

      pi->hasminmax = true;

      pi->outputmin = 0;

      pi->outputmax = 1;

      pi->exp = 0;

      pi->units = "";       // no units defined

      // initialize parameter 1 ("m_damp_14")

      pi = self->__commonstate.params + 1;

      pi->name = "damp2";

      pi->paramtype = GENLIB_PARAMTYPE_FLOAT;

      pi->defaultvalue = self->m_damp_14;

      pi->defaultref = 0;

      pi->hasinputminmax = false;

      pi->inputmin = 0;

      pi->inputmax = 1;

      pi->hasminmax = true;
```

```
pi->outputmin = 0;

pi->outputmax = 1;

pi->exp = 0;

pi->units = "";       // no units defined

// initialize parameter 2 ("m_damp_11")

pi = self->__commonstate.params + 2;

pi->name = "damp3";

pi->paramtype = GENLIB_PARAMTYPE_FLOAT;

pi->defaultvalue = self->m_damp_11;

pi->defaultref = 0;

pi->hasinputminmax = false;

pi->inputmin = 0;

pi->inputmax = 1;

pi->hasminmax = true;

pi->outputmin = 0;

pi->outputmax = 1;

pi->exp = 0;

pi->units = "";       // no units defined

// initialize parameter 3 ("m_feeder_13")

pi = self->__commonstate.params + 3;

pi->name = "feeder";

pi->paramtype = GENLIB_PARAMTYPE_FLOAT;

pi->defaultvalue = self->m_feeder_13;

pi->defaultref = 0;

pi->hasinputminmax = false;

pi->inputmin = 0;

pi->inputmax = 1;

pi->hasminmax = true;

pi->outputmin = 0;
```

```
    pi->outputmax = 1;
    pi->exp = 0;
    pi->units = "";       // no units defined

    return self;
}

/// Release all resources and memory used by a State object:

void destroy(CommonState *cself) {
    State * self = (State *)cself;
    genlib_sysmem_freeptr(cself->params);

    delete self;
}

} // cDelay_gendsp::
```