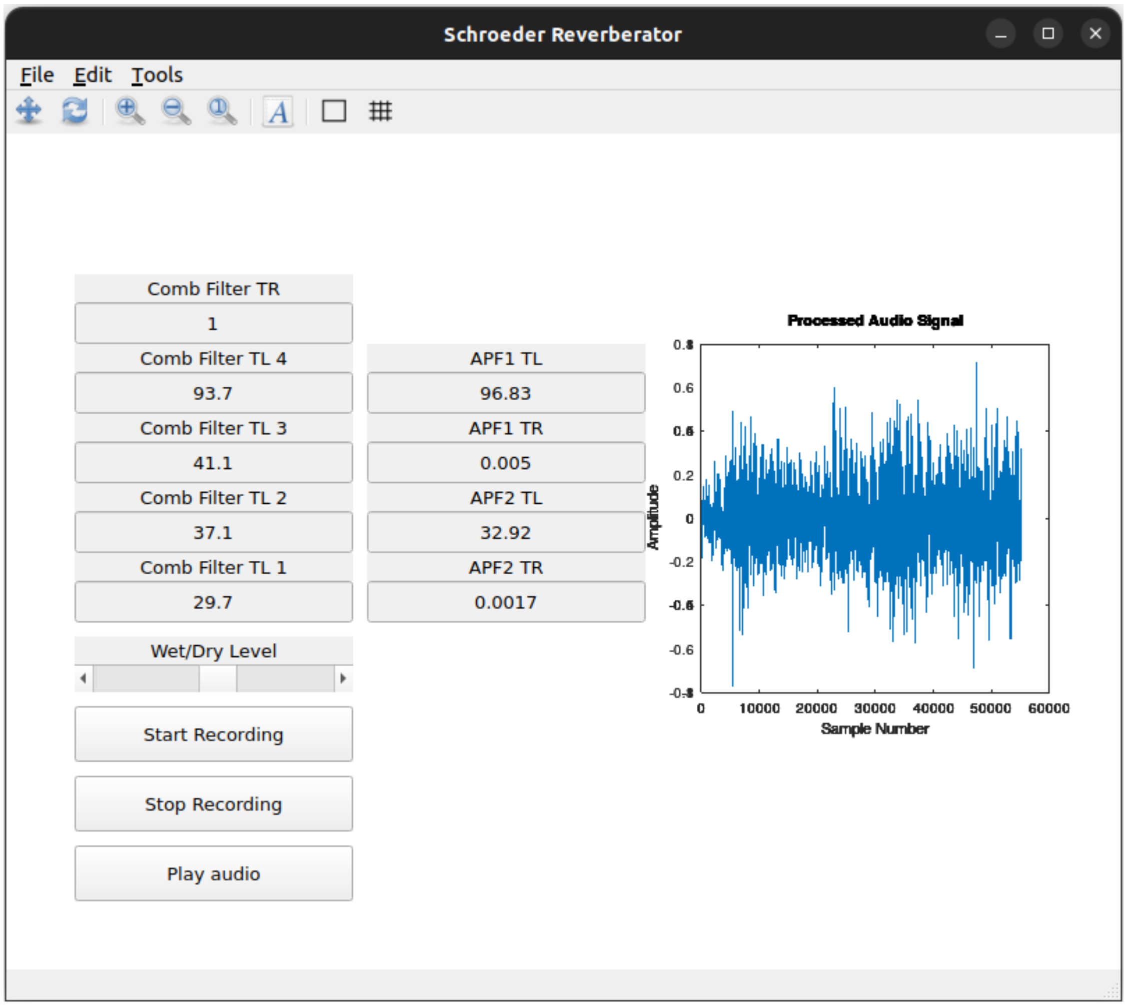
Schroeder Reverberator

Matlab/Octave Project Work

Audergon Colin
V0.1.0



Comb Filter TR	1
Comb Filter TL 4	93.7
Comb Filter TL 3	41.1
Comb Filter TL 2	37.1
Comb Filter TL 1	29.7
APF1 TL	96.83
APF1 TR	0.005
APF2 TL	32.92
APF2 TR	0.0017

Bern University of Applied Sciences
Engineering and Computer Science
Electrical Engineering and Information Technology

1 Introduction

Over the last decades, a large portion of music producing shifterd from hardware devices to software. The majority of music producer now uses DAWs (Digital Audio Workstations),in which they find a variety of virtual instruments, as well as virtual audio effects. The initial goal of the project was to implement a real time reverberation, this approach was deprecated once it appears that octave wasn't able to handle two threads at the same time.It was then decided to write an algorythm processing a full recording, and to wrap it with a graphical user interface with user tunable parameters.

In the current state the reverb performs well, it has what can be call a " metalic" sound. The computation time necessary to apply the algorythm is reasonable. And the user interface is usable.

2 Methods

2.1 Tools:

Software: *Octave v.6.40*
Operating system: *Ubuntu 22.04*

2.2 Reverb algorythm design overview

A schroeder reverberator is a unit based on a set of filters: a first layer of N comb filters, in parrallel, followed by two all pass filters. The filters configuration is shown in figure [1].

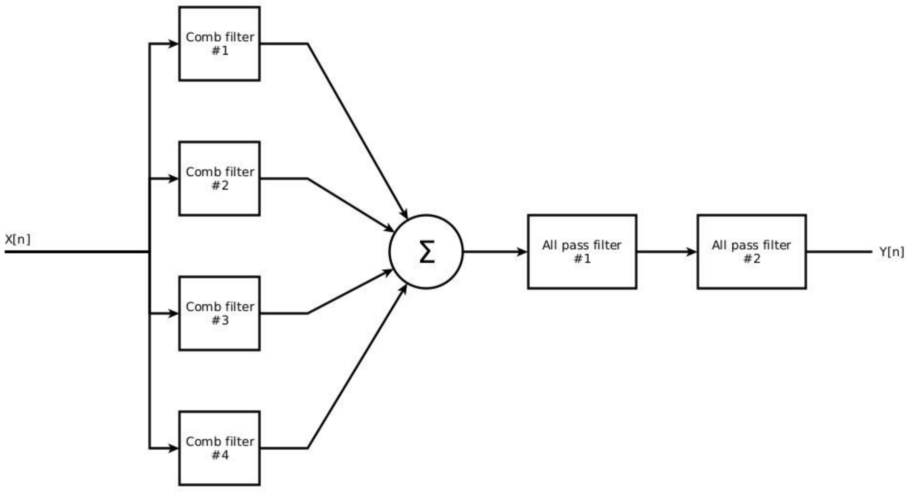


Figure 1: reverb algorithm configuration

2.3 Comb filter design:

A comb filter is a filter using a delay of N elements to add a delayed version of the signal to itself [1]. The filter structure can be obserberved below on the figure [3].

The figure [2] shows the theoretical impulse response of the comb filter. As one can see, it takes N index for the input pulse to be repeated at at the output, multipled by a gain below 1. The time between the repetition is the "loop time".
 Once the output pulse reaches -60dB, it is considered as silent. The time it takes between the impulse to the silence is known as T60, or the reverb time. In this design, all the comb filter share the same reverb time.

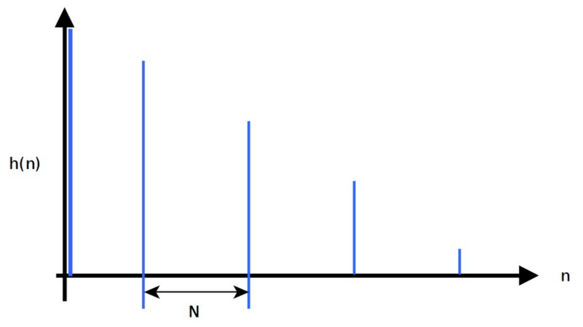


Figure 2: comb filter impulse response

To design the comb filters, one must know the relationship between all the parameters. Starting with the design flow, one can extract the difference equation, see figure [3].

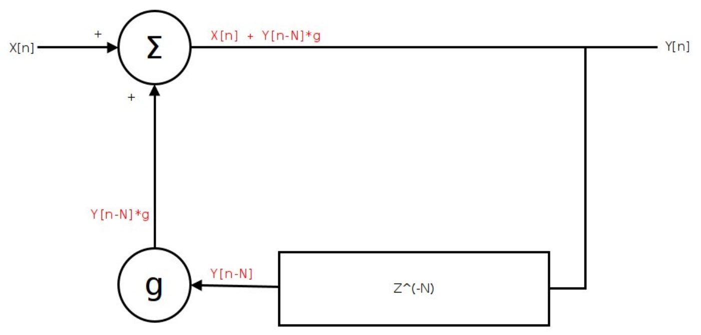


Figure 3: comb filter block diagram

As one can see, Y[n] can be described as:

$$Y[n]=X[n]+Y[n-N]\cdot g$$

By rearreanging the equation to have all the Y terms on the left side:

$$Y[n]-Y[n-N]\cdot g=X[n]$$

Using z as the delay operator, the equation can be rewritten as:

$$(1-g\cdot z^{(-N)})\cdot Y(z)=X(z)$$

Thus the transfer function:

$$\frac{1}{1-g\cdot z^{(-N)}}=\frac{Y(z)}{X(z)}$$

The loop time and reverb time are linked to the gain of the feedback loop. As one knows the amplitude at the reverb time beeing-60dB, it is a good entry point to the filter dimensionning.

$$T_{reverb}=-60\,dB=20\cdot\log_{10}(|x|)$$
$$x=10^{\left(\frac{T_{reverb}}{20}\right)}=0.001$$

With a known sampling frequency (fs) and loop time (T_L) one can now compute the required delay line size.

$$N = \lceil T_L \cdot fs \rceil$$

As one knows the reverb time (T_{reverb}), the feedback gain of the filter (g) can be computed.

$$g = 0.001^{\frac{N}{fs \cdot T_{Reverb}}}$$

In the final implementation, these computation will be automated, and each filter will be contained in a class, allowing for easier access to their propreties.

2.4 All pass filter design

The all pass filter impulse response is taken for [2] can be described as:

$$\frac{Y(z)}{X(z)} = \frac{-g + z^{-N}}{1 - g \cdot z^{-N}}$$

The delay line length (N) can be computed the same manner as for the comb filters, given the sampling frequency, the loop time and the reverb time. Once again, this filter was implemented in a class for simpler use.

2.5 Audio scaling

A best practice when working with digital signals, and particuraly audio signals, is to used normalized signal. With audio, it is also a good idea to remove the dc offset that might be present. This can be done in octave by deviding the signal by the maximum value of the signal and removing the mean.
This was implemented in a function, shown in the following code snippet.

```
function y = scaleAudio(x)
  y = x/max(x) - mean(x);
  endfunction
```

2.6 Effect level control

As one might not want the signal to be completely reverberated, but rather a mix between an unaffected signal and a processed signal, a function implementing a mixer was written. Its implementation in octave can be found below.

```
function y = audioDryWet(audioDry,audioWet,mixLevelWet)
  % Makes sure mix level is normalized
    if mixLevelWet < 0 || mixLevelWet > 1
      error('mixLevelWet should be between 0 and 1');
  end
    % Ensure the audio signals have the same length
  if length(audioDry) ~= length(audioWet)
      error('audioDry and audioWet must have the same length');
  end

  y = (1 - mixLevelWet) * audioDry + mixLevelWet * audioWet;
  endfunction
```

3 Results

3.1 Comb filter design

The comb filter implementation was tested with the code snippet bellow, the resulting impulse response was plotted and can be observed in the figure[4].

```
% Comb filter test script
Fs = 44100;
cmf1_reverb_time = 2;
tl = 0.00575 *1000;
cmf1=combFilter(tl,cmf1_reverb_time,Fs);
% Create a sample input signal
pulse_length = 1*Fs
pulse = zeros(1,pulse_length);
pulse(1) = 1;
cmf1 = cmf1.filterOutput(pulse);
pulse_out = cmf1.content;
scaled_pulse_out = pulse_out/max(pulse_out)-mean(pulse_out);
% Plot the input and output signals
subplot(3, 1, 1);
stem(pulse);
title('Input Signal');
xlabel('Sample');
ylabel('Amplitude');

subplot(3, 1, 2);
stem(pulse_out);
title('Output Signal');
xlabel('Sample');
ylabel('Amplitude');

subplot(3, 1, 3);
stem(scaled_pulse_out);
title('Scaled Output Signal');
xlabel('Sample');
ylabel('Amplitude');
```

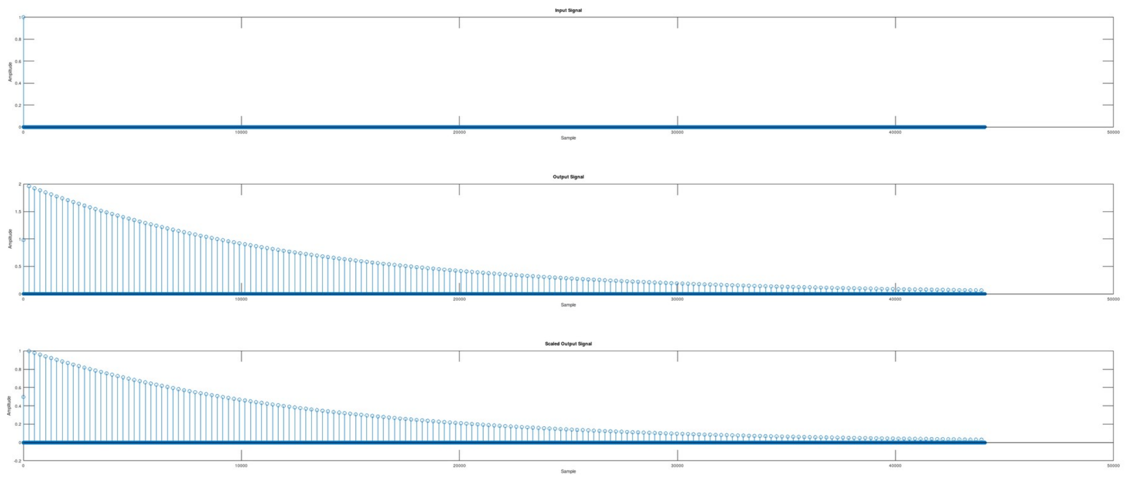


Figure 4: comb filter impulse response

The figure [5] shows the first two spikes visible at the output of the filter.

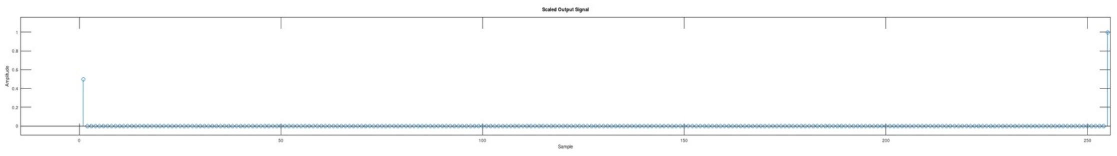


Figure 5: comb filter impulse response, two fisrt spikes

For a reverb time (T_{Reverb})of 2 seconds, a loop time (T_L)of 5.75 ms with a sampling frequency (fs) of 44.1kHz. One would compute:

$$N = \lceil T_L \cdot fs \rceil = \lceil \frac{5.75\,s}{1000} \cdot 44100\,Hz \rceil = \lceil 253.575 \rceil = 254$$

Matrices are indexed at 1 in ocatve, where the first pulse is situated. The second pulse is at the index 255. Thus, the length of the delay line is 254.

In the built in command line of octave, one can type the name of the filter followed by a dot and the propreties of the class desired. In this case, by entering the following “cmf1.gain”, octave returns: 0.9803.

by adding the following lines to the test script, one can test the gain value.

```
disp("difference between pulse 2 and 3:")

p1 = scaled_pulse_out(1,255 + cmf1.delayLineSize)
p2 = scaled_pulse_out(1,255 + 2*cmf1.delayLineSize)

disp("gain:")
disp(p2/p1 )
```

Resulting in:

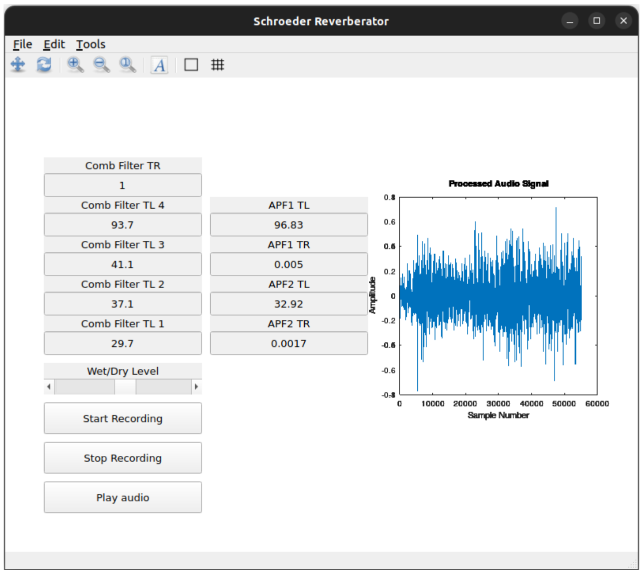
```
difference between pulse 1 and 2:
p1 = 0.9781
p2 = 0.9588
gain:
0.9803
```

3.2 All pass filter design

A script similar to the test script for the comb filter has been used to test the all pass filters. Yelding similar results.

3.3 User interface design

The reverb is implemented with a graphical user interface, shown in figure [6]. This allows for a simplified operation without the need of command prompt. An audio recorder is built in so the user can record and playback affected audio. The reverb algorythm parameters, such as the filters reverb time and loop time can be edited with the corresponding text box. A slider controlls the wet/dry ratio. Finally, the affected audio is ploted on a graph, for visual reference.



Comb Filter TR	1
Comb Filter TL 4	93.7
Comb Filter TL 3	41.1
Comb Filter TL 2	37.1
Comb Filter TL 1	29.7
APF1 TL	96.83
APF1 TR	0.005
APF2 TL	32.92
APF2 TR	0.0017

Figure 6: Graphical user interface

4 Discussion

In this project, a shroeder reverbarator reverb algorithm was implemented using octave. The algorithm performs well and has a distinct sound. For the most part the project went according to plan, except for the real time implentation. Indeed, it wasn't possible to implement a real time version of the system, as octave does not support dual threading. Which is necessary in this case.

In futur iteration of the project, a revamp of the graphical user interface could be beneficial. One might also want to try different approaches to tackle the real time operation. Maybe using an external script to start two distinct octave scripts, with a thread syncronisation via a file and/or sockets. Otherwise, implementing the algorythm in C++ could yeld similar, or better results, which is a route that will be taken in the form of another project.

Colin Audergons
Fribourg, 3 juin 2024.

5 Sources

[1] Wikipedia Contributors, "Comb filter," *Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/wiki/Comb_filter (accessed Jun. 2, 2024).

[2] J. O. Smith III, "Schroeder Reverberators," *CCRMA, Stanford University*, https://ccrma.stanford.edu/~jos/pasp/Schroeder_Reverberators.htm (accessed Jun. 2, 2024).