Virtual Analog Simulation and Extensions of Plate Reverberation

Master Thesis

Silvin Willemsen - SMC171032

Aalborg University
Electronics and IT

AALBORG UNIVERSITY
STUDENT REPORT

Electronics and IT
Aalborg University
http://www.aau.dk

Title:
Virtual Analog Simulation and Extensions of Plate Reverberation

Theme:
Virtual Analog Simulation

Project Period:
Fall 2016 - Spring 2017

Project Group:
SMC171032

Participant(s):
Silvin Willemsen

Supervisor(s):
Stefania Serafin
Jesper Rindom Jensen

Copies: 1

Page Numbers: 100

Date of Completion:
May 21, 2017

Abstract:

In the 50s and 60s, steel plates were popularly used as a technique to add reverberation to sound. As a plate reverb itself is quite bulky (2×1 m) and requires lots of maintenance, a digital implementation would be desirable. Currently, the available (digital) plug-ins rely solely on recorded impulse responses or simple delay networks. Virtual Analog (VA) simulations, on the other hand, rely on a model of the analog effect they are simulating, resulting in a sound and 'feel' the of the classical analog effect. In this project, a VA simulation of plate reverberation is presented where parameters like the positions of the in- and outputs and the dimensions of the plate can be changed while sound goes through. This results in a unique flanging and pitch bend effect, respectively, which has not yet been achieved by the current state of the art. This has been implemented into a real-time plugin of which the output sound has been reported to be enjoyable (4/5).

Contents

Preface

This thesis was made as a completion of the Sound and Music Computing master's programme at Aalborg University. It describes a Virtual Analog simulation and extensions of plate reverberation and the process leading up to the result. The project has been done in collaboration with the French hard- and software developing company Arturia and was supervised by Stefania Serafin and Jesper Rindom Jensen at Aalborg University.

The topic of Virtual Analog simulations has much intrigued me since the semester leading up to this project. It uses current day technology to simulate classical effects that many musicians still love nowadays. Additionally, the simulations make it possible for otherwise fixed parameters to be changed, creating sounds never heard before. To me, this fusion of classical and modern in a musical context is something truly amazing and has motivated me greatly during this project.

This report is structured as follows: Chapter 1 gives an introduction on the topics of Virtual Analog simulations and plate reverberation. Also, the goals that were set for this project will be stated. Chapter 2 then gives an overview of the physics of a thin metal plate. This needs to be understood in order to reach the aforementioned goals. Chapter 3 describes the state of the art of both implementation approaches as well as plugins in digital plate reverberation after which Chapter 4 compares these different approaches by describing experiments that have been done based on these. At the end of this chapter, the approaches are compared and the most suitable one is chosen to be used for this project. Chapter 5 describes an algorithm created based on the chosen implementation approach and includes extensions that have not yet been achieved by the current state of the art. Chapter 6 presents a real-time plugin based on this algorithm. Then, Chapter 7 presents an evaluation on both the output sound of the implementation as well as the plugin. Chapter 8 discusses the audible results of the implementation. Lastly, Chapter 9 concludes and presents possible future work.

Throughout the course of this project I have published a paper at the *14th Sound* *and Music Computing Conference* [1]. This paper can be found in Appendix A. First of all, I would like to thank Stefania Serafin and Jesper Rindom Jensen for supervising me during the course of this project and putting in a lot of time to read through both this report and the published conference paper. Secondly, I would like to thank Stefano d'Angelo for being Arturias contact person and guiding me through the first phases of the project.

Aalborg University, May 21, 2017

Silvin Willemsen
<swille15@student.aau.dk>

Chapter 1

Introduction

A great number of digital audio effects is currently available to musicians and producers. Many sounds we could not even imagine a few years ago we can now create using current DSP technology. However, despite this immense amount of options, there is still a great desire for the sound of the classical analog effects that made their first appearance in the late 40s and characterised music from the 50s and 60s onwards.

1.1 Virtual Analog

Even though, generally, 'regular' digital (later referred to as digital) sound effects 'do the job', they do not have the 'feel' that the old analog effects had – something greatly desired by many musicians. Contrary to digital effect simulations, Virtual Analog (VA) simulations rely on a model of the analog effect they are simulating [2]. A great advantage of VA simulations over the original systems is that they do not age and thus do not require time consuming maintenance. Also, when digitalised they are easily accessible, mostly simpler to use and can be made much cheaper than their analog counterparts. Naturally, the sound of a used analog audio effect can have its charm, but if desired, this can be modelled into the simulation. Also, VA simulations make it possible for parameters like room size, material properties, etc. to be changed, which is physically impossible or very hard to do. This can result in unique sounds that can only be created using VA simulations.

1.2 Plate Reverberation

Analog audio effects have been around for almost a century. They especially flourished in the 50s and 60s as new techniques of manipulating audio signals were developed [3]. A popular reverberation technique at the time was plate reverberation. A plate reverb utilises a small speaker (actuator) attached to a big steel plate to make it vibrate, and several pickups to pick up the sound after it has propagated through the plate (see Figure 1.1). Several different plate reverbs made it to the market, the most popular being the *Elektro-Mess-Technik*'s EMT140 used in the Abbey Road Studios and undoubtedly leaving a mark on music in the aforementioned years. In fact, it was the only reverb used on Pink Floyd's Dark Side of the Moon [4].

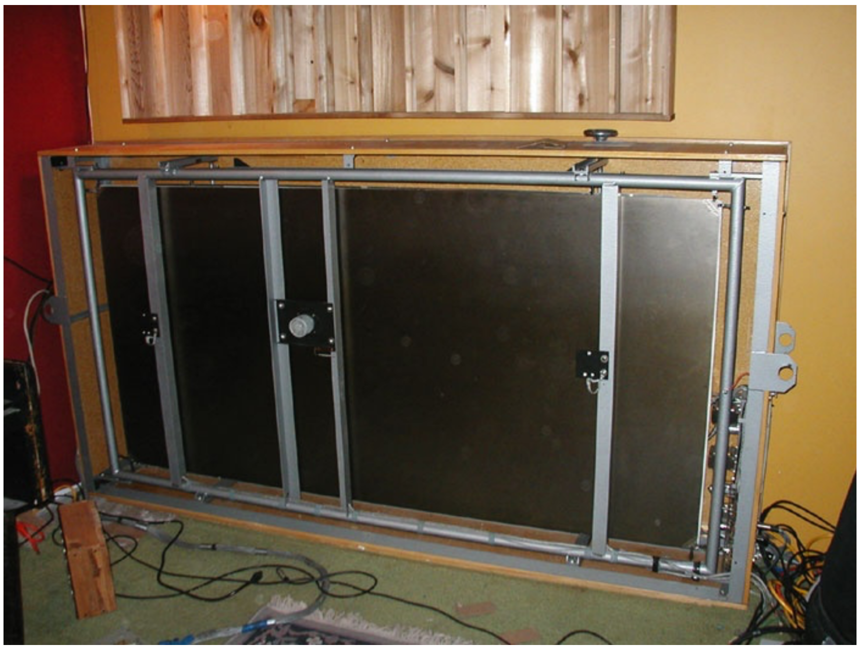


Figure 1.1: "An original hardware Plate Reverb." (Source and caption: [5])

1.3 Project Goals

A big issue of using an actual plate reverb is the sheer size and weight of it. The plate is about 2×1 m big and weighs (together with the rest of the installation) roughly 270 kg [6], hence, a digital implementation of it would be desirable. However, in order to sound as close to the physical plate reverb as possible, the implementation should be a simulation of the real thing: a VA simulation. As digital and VA implementations of plate reverberation already exist, research needs to be done on which implementations approaches exist and which one has the best speed versus quality tradeoff. The first goal of this project is therefore:

To find the implementation approach with the best speed versus quality tradeoff for a VA simulation of Plate Reverberation.

Currently available VA plugins (as will be described in Chapter 3) do not use the full potential of VA simulations. As said above, VA simulations make it possible to manipulate parameters that are physically impossible to change. Parameters like, for instance, pickup positions and sheet-size can be made dynamic, i.e., changed while sound is going through the plate. Hence, the second goal of this project is:

To explore the possibilities that VA simulations pose on Plate Reverberation and extend on already existing models accordingly.

Lastly, to apply this all, the model and extensions to it must be implemented. This should be made real-time if possible. The last goal of this project is thus:

To create a (preferably) real-time VA simulation of Plate Reverberation.

Chapter 2

Plate Physics

VA simulations rely on a model of the effect they are simulating. In order to create a simulation for plate reverberation, the physics of it first needs to be understood. This chapter describes the physics of plate reverberation. First, the Kirchhoff-Love model, one that mathematically describes vibration in thin metal plates, will be explained. Thereafter, different aspects of plate reverberation will be explored, including frequency dispersion, different kinds of damping mechanisms and boundary conditions.

2.1 Kirchhoff-Love Model

The Kirchhoff-Love model mathematically describes stresses and deformations in thin plates subjected to external forces. The partial differential equation for an isotropic plate (including damping) is [7]:

$$\frac{\partial^2 u}{\partial t^2} = -\kappa^2 \nabla^4 u - c\frac{\partial u}{\partial t} + f(x_{\text{in}}, y_{\text{in}}, t). \tag{2.1}$$

Here, $u = u(x, y, t)$, a state variable that describes the transverse plate deflection and is defined for $x \in [0, L_x]$, $y \in [0, L_y]$ – where L_x and L_y are the horizontal and vertical plate dimensions, respectively – and $t \geq 0$, c is a loss parameter, ∇^4 is the biharmonic operator, $f(x_{\text{in}}, y_{\text{in}}, t)$ is the input signal at input source location $(x_{\text{in}}, y_{\text{in}})$ at time instance t and finally, κ^2 can be referred to as the stiffness parameter:

$$\kappa^2 = \frac{Eh^2}{12\rho(1-\nu^2)}, \tag{2.2}$$

where E, h, ρ, and ν are the Young's modulus, plate thickness, plate density and Poisson's ratio, respectively.

Note: the third (z) dimension of the plate can be ignored as the Kirchhoff-Love model *assumes this dimension to be "much smaller than the other two" [8].*

Equation (2.1) can be seen analogous to a mass-spring-damper system written as the solution to the acceleration of u over x and y. In the first term, κ^2 can be seen as a spring constant and therefore has a negative, restoring effect on the acceleration. It also makes sense that, for instance, if the plate is thicker – which increases the value of κ^2 – there will be a bigger restoring force.

The biharmonic operator is defined as:

$$\nabla^4 = \frac{\partial^4}{\partial x^4} + 2\frac{\partial^4}{\partial x^2 \partial y^2} + \frac{\partial^4}{\partial y^4},$$

and can be multiplied onto a variable with two spatial dimensions, in this case the state variable u. The higher the resulting value of this at a certain point of the plate, the higher the restoring force at that point will be.

The second term of the equation can be seen as a damper and therefore has a negative effect on the acceleration as well. The bigger the loss parameter c or the velocity of u at a certain point, the bigger the negative effect on the acceleration at that point will be.

The last term is the input at a specified location. It makes sense that this is the only positive term, as it is the only term that adds energy to the system.

2.2 Frequency Dispersion

One feature that characterises the sound of a plate reverb is the fact that higher frequencies propagate faster through a metal plate than lower frequencies [7, 9]. This phenomenon is called frequency dispersion. The dispersion relation is defined as:

$$\omega^2 \simeq \gamma^4 \kappa^2, \tag{2.3}$$

where ω is the angular frequency and γ the wavenumber. The group and phase velocities are defined as:

$$c_{\text{ph}} = \sqrt{\kappa\omega} \qquad c_{\text{gr}} = 2\sqrt{\kappa\omega}. \tag{2.4}$$

Using the properties of the EMT140 (steel, 0.5 mm thick) and a frequency range of 20-20,000 kHz, group velocities can vary between ca. 20-620 m/s. In regular (or room) reverberation all frequencies travel at the same speed making frequency dispersion one of the main differences between plate and room reverberation.

2.3 Damping

In [7] and [9], the authors state that in plate reverbs, three different kinds of damping occur: thermoelastic damping, radiation damping and damping induced by a porous medium. All three will be shortly described in this section.

Thermoelastic Damping

Thermoelastic damping occurs in materials with a high thermal conductivity. It is an internal damping mechanism that damps different frequencies at different strengths according to the following formula [9]:

$$\alpha_{\text{th}}(\omega) = \frac{\omega}{2}\eta(\omega) \approx \frac{\omega^2 R_1 C_1}{2(\omega^2 h^2 + C_1^2/h^2)}, \tag{2.5}$$

where R_1 and C_1 are terms that contain material dependent constants. In the case of the EMT140 plate reverb these are [7]:

$$R_1 = 4.94 \cdot 10^{-3} \qquad C_1 = 2.98 \cdot 10^{-4} \text{ rad m}^2/\text{s}.$$

A graph showing the thermoelastic damping for the EMT140 plate can be seen in Figure 2.1:

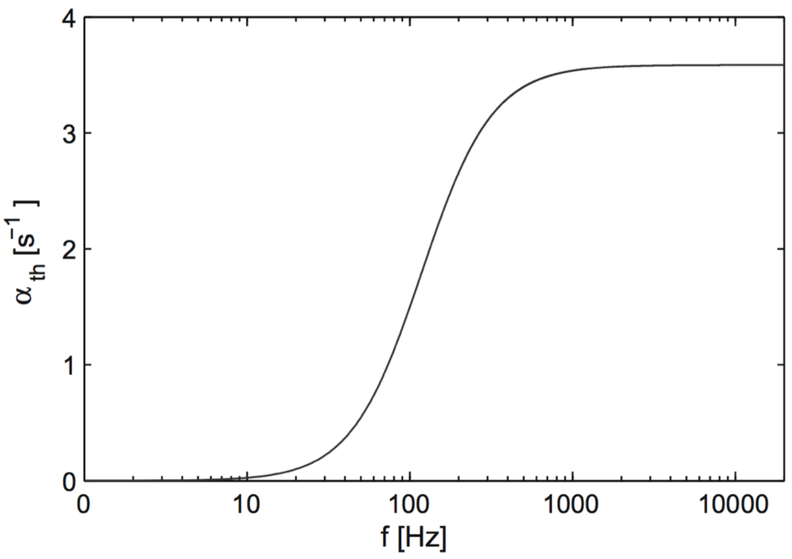


Figure 2.1: "Thermoelastic damping for the EMT140 plate. (...)" (Source image and caption: [9])

Radiation damping

Radiation damping happens when vibration is converted to acoustic energy. This happens according to the following formula [7, 9]:

$$\alpha_{\text{rad}} = \frac{1}{4\pi^2}\frac{c_a\rho_a}{\rho h}\frac{2(L_x + L_y)}{L_x L_y}\frac{c_a}{f_c}g(\psi), \tag{2.6}$$

$$g(\psi) = \frac{(1-\psi^2)\ln[(1+\psi)/(1-\psi)] + 2\psi}{(1-\psi^2)^{3/2}}, \text{ where } \psi = \sqrt{\frac{f}{f_c}}$$

Here, ρ_a and c_a are the density of air and speed of sound in air respectively and f_c is the critical frequency which can be calculated using:

$$f_c = \frac{c_a^2}{2\pi\kappa}.$$

A graph showing the predicted radiation damping can be seen in Figure 2.2:

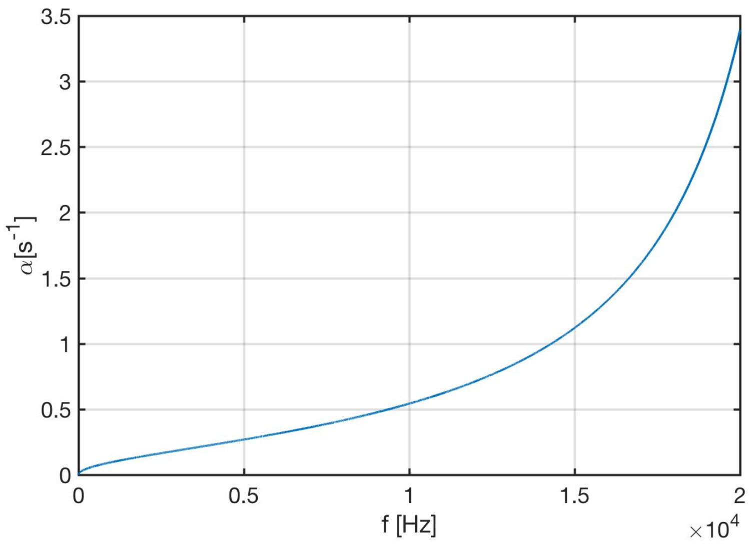


Figure 2.2: Predicted radiation damping using EMT140 properties.

Damping induced by Porous Medium

Plate reverberators contain a porous plate positioned behind the metal plate. The distance between the two entities can be set creating a difference in low-frequency decay [7, 9]. The damping plate position changes this decay between roughly 0.6 – 5.5 seconds [10] and can be manipulated on the interface of a real plate reverb like the EMT140.

A graph showing the damping induced by a porous medium can be seen in Figure 2.3.

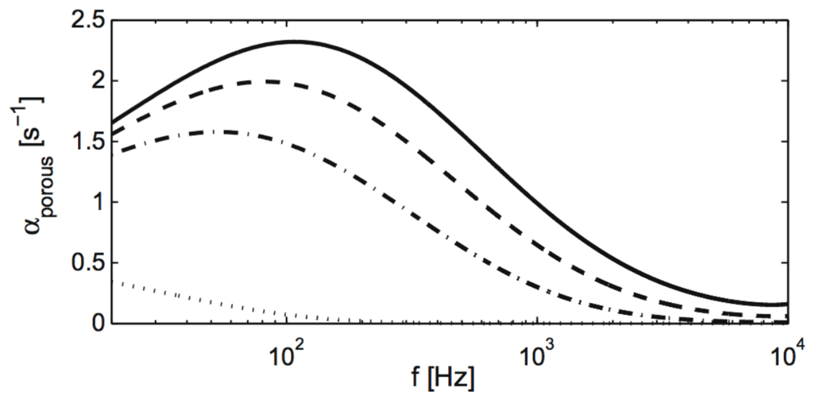


Figure 2.3: "Damping factor induced by the porous plate computed with Cummings model for different distances between the plates: d = 65.8 mm (dotted); 17.3 mm (dash-dotted), 13.2 mm (dashed); 11.0 mm (solid)." (Source image and caption: [9])

2.4 Boundary Conditions

The state of the edges of a plate are referred to as the boundary conditions. The authors state in [11] that a plate can have three different boundary conditions: free, clamped or simply supported (hinged). Every combination will have a unique impulse response and will thus sound different. This report will use the case where all sides are simply supported as this is the case with a plate reverb. This means that, recalling eq. (2.1), state variable $u = 0$ at $x = 0$, $x = L_x$, $y = 0$ and $y = L_y$.

Chapter 3

State Of The Art

This chapter is divided into two sections. First, the state of the art (SOTA) regarding digital implementation approaches is described, then the SOTA regarding digital and VA plate reverb plugins is described.

3.1 Implementation Approaches

3.1.1 Convolution Reverb

Convolution reverbs use the impulse response of, for instance, a room and convolve this with an input signal to get a reverberated output. In order to get a convolution plate reverb, the impulse response of an actual plate reverb needs to be recorded. Depending on the length of the impulse response, the computational power needed can vary a lot. Impulse responses of several seconds for example make for a very computationally intensive and inefficient implementation [12].

The advantage of using convolution for digital plate reverberation is that the audio output very closely resembles the audio output from the actual plate reverb. However, there is not a lot of flexibility in this implementation as the impulse response is recorded and hence static. All parameters, including plate properties, pickup positions and speaker positions are fixed.

3.1.2 Feedback Delay Networks

As devised by Jean-Marc Jot in [13] and used as an implementation approach for creating a digital plate reverb by Jonathan Abel in [10], Feedback Delay Networks (FDNs) are one way of digitally modelling a plate reverb. FDN structures are a nice solution to the digital reverberation problem. They can be represented as a set of digital delay lines whose inputs and outputs are connected by a feedback matrix [14] (see Figure 3.1). FDNs provide an way to efficiently parameterise dynamic control of the reverberators and can thus be a VA solution to the problem.

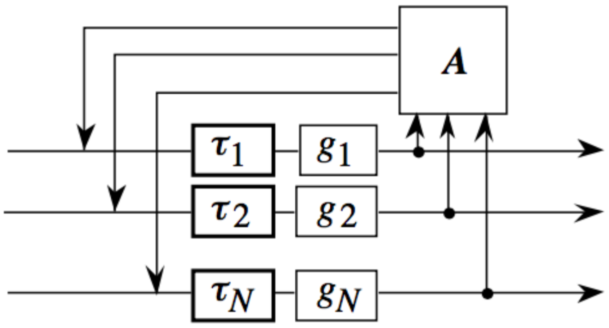


Figure 3.1: "Basic Feedback Delay Network." (Source image and caption: [14])

In his implementation, Abel uses a hybrid reverberator structure that is a combination of a convolutional impulse response $c(t)$ and an FDN reverberator dependent on damper setting θ with impulse response $d(t;\theta)$ running in parallel. This creates impulse response $h(t;\theta)$ being:

$$h(t;\theta) = c(t) + d(t;\theta). \tag{3.1}$$

An example of this, visualised using waveforms, can also be seen in Figure 3.2:

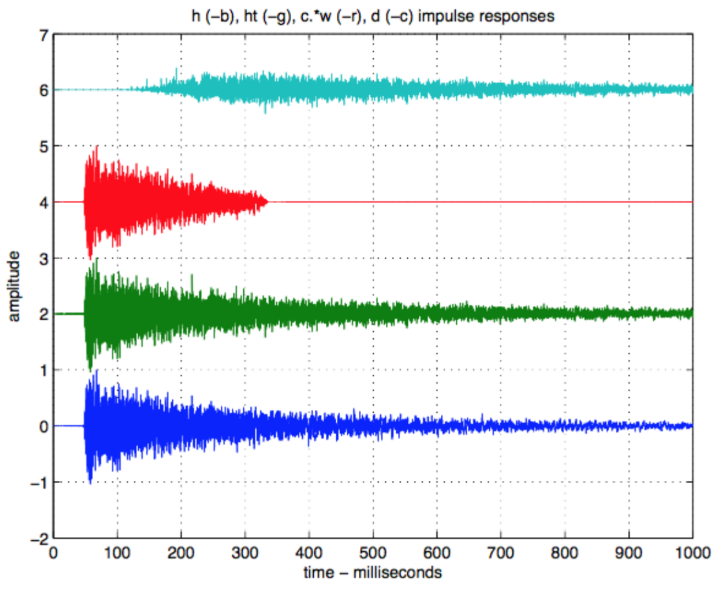


Figure 3.2: "Example Hybrid Structure Responses. From top to bottom: FDN d(t); windowed convolution c(t); complete hybrid response y(t); measured response h(t)." (Source image and caption: [10])

3.1.3 Finite-Difference Schemes

Finite-difference schemes are used for solving differential equations (such as the Kirchhoff-Love model) by approximating them with difference equations. In [9], Arcas says that finite-difference schemes can be used to verify physical models. What these schemes basically do, is to subdivide a large problem or model into smaller parts. In the case of a plate reverb it subdivides the plate into a grid where for plate state $u(x,y,t)$: $x = lX$, $y = mX$ and $t = nT$ for $l = 0, 1, ..., \frac{L_x}{X}, m = 0, 1, ..., \frac{L_y}{X}$ and $n \in \mathbb{N}^0$. Here X is the spacing between adjacent points and $T = 1/f_\text{s}$ is the time step. Naturally, the smaller spacing X and the higher sample rate f_s, the more detailed the approximation.

An implementation using finite-difference schemes is shown in [15]. Solving the formula for simply supported boundary conditions gives:

$$\begin{aligned}(1+\sigma T)u_{l,m}^{n+1} =& 2u_{l,m}^{n} - (1-\sigma T)u_{l,m}^{n-1} - \mu^2\left(u_{l+2,m}^{n} + u_{l-2,m}^{n} + u_{l,m+2}^{n} + u_{l,m-2}^{n}\right) \\ &- 2\mu^2\left(u_{l+1,m+1}^{n} + u_{l+1,m-1}^{n} + u_{l-1,m+1}^{n} + u_{l-1,m-1}^{n}\right) \\ &+ 8\mu^2\left(u_{l+1,m}^{n} + u_{l-1,m}^{n} + u_{l,m+1}^{n} + u_{l,m-1}^{n}\right) - 20\mu^2 u_{l,m}^{n} \\ &+ \frac{T^2}{\rho H X^2}\delta_{l_i,m_i} f^n. \end{aligned} \tag{3.2}$$

where $\sigma \geq 0$ is a loss parameter and $\mu = \kappa T/X^2 \leq 1/4$ for stability. In the above equation, the state u at position (l,m) at the next time instance is computed using multiple points on the grid at and around (l,m) at the current time instance, one point at (l,m) at the previous time instance and the input signal at point (l_i, m_i).

As can be seen in [16] the use of finite-difference schemes is very computationally demanding. With their implementation and a gridsize of 26×34 it took 22.6 seconds to output 1 second of sound at a sample rate of 44,100 Hz.

3.1.4 Modal Description

The state u, as seen in the Kirchhoff-Love equation (2.1) can be modelled as being a summation of a number of different modes [17]:

$$u = \sum_{m=1}^{M}\sum_{n=1}^{N} q_{mn}\Phi_{mn}(x,y), \tag{3.3}$$

where q_{mn} is the unknown 'amplitude' of mode (m,n) (m being the mode over the horizontal axis and n over the vertical axis of the plate) and $\Phi_{mn}(x,y)$ is a modal shape defined over $x \in [0, L_x]$ and $y \in [0, L_y]$. Apart from the chosen time step ($1/f_\text{s}$) the computational speed depends on the number of modes (M and N) being calculated instead of the gridsize as described in the previous section. In theory, the number of modes are infinite, but as can be seen in further on in this section, there exists a stability condition that limits this amount.

For a rectangular plate with sides L_x and L_y, using simply supported boundary conditions, $\Phi_{mn}(x,y)$ be calculated [17]:

$$\Phi_{mn}(x,y) = \frac{4}{L_x L_y} \sin\frac{m\pi x}{L_x} \sin\frac{n\pi y}{L_y}, \quad (m,n) \in \mathbb{Z}^+. \tag{3.4}$$

With eq. 3.3 substituted into eq. 2.1 and the derivative operators then substituted with difference operators:

$$\frac{\partial^2 q_{mn}}{\partial t^2} \Rightarrow \frac{1}{k^2}\left[q_{mn}^{t+1} - 2q_{mn}^{t} + q_{mn}^{t-1}\right], \quad \frac{\partial q_{mn}}{\partial t} \Rightarrow \frac{1}{2k}\left[q_{mn}^{t+1} - q_{mn}^{t-1}\right],$$

q_{mn} can be found using the following update equation [17]:

$$\left(\frac{1}{k^2} + \frac{c_{mn}}{\rho h k}\right) q_{mn}^{t+1} = \left(\frac{2}{k^2} - \omega_{mn}^2\right) q_{mn}^{t} + \left(\frac{c_{mn}}{\rho h k} - \frac{1}{k^2}\right) q_{mn}^{t-1} + \frac{\Phi_{mn}(x_p, y_p)}{\rho h} P^t, \tag{3.5}$$

where P^t is the input signal at time instance t at specified input location (x_p, y_p), k is the chosen time step $1/f_s$ and c_{mn} is a loss coefficient that can be set per mode. This gives a lot of control over the frequency content of the output sound. The solution to both the loss coefficients and the eigenfrequencies ω_{mn} can be found in Chapter 5.

The output can be retrieved at a specified point by multiplying the q_{mn} that was found in eq. (3.5) with a specified output point in Φ_{mn} [17]:

$$u_{\text{out}} = \sum_{m=1}^{M} \sum_{n=1}^{N} q_{mn} \Phi_{mn}(x_{\text{out}}, y_{\text{out}}). \tag{3.6}$$

Resonator Filters

The total system described above can also be seen as an addition of many frequency dependent filters. If in eq. (3.5) we let:

$$A_{mn} = \frac{1}{k^2} + \frac{c_{mn}}{\rho h k}, \quad B_{mn} = \frac{2}{k^2} - \omega_{mn}^2, \quad C_{mn} = \frac{c_{mn}}{\rho h k} - \frac{1}{k^2},$$

$$\Phi_{mn}(x_p, y_p) = \Phi_{\text{in}_{mn}}, \quad \Phi_{mn}(x_{\text{out}}, y_{\text{out}}) = \Phi_{\text{out}_{mn}},$$

the transfer function of the total filter will be:

$$H(z) = \sum_{m=1}^{M} \sum_{n=1}^{N} \frac{\frac{\Phi_{\text{in}_{mn}} \cdot \Phi_{\text{out}_{mn}}}{\rho h} z^{-1}}{A_{mn} + B_{mn} z^{-1} + C_{mn} z^{-2}}. \tag{3.7}$$

If we normalise with respect to A_{mn}, the transfer function will look like this:

$$H(z) = \sum_{m=1}^{M} \sum_{n=1}^{N} \frac{\frac{\Phi_{\text{in}mn} \cdot \Phi_{\text{out}mn}}{\rho h A_{mn}} z^{-1}}{1 + \frac{B_{mn}}{A_{mn}} z^{-1} + \frac{C_{mn}}{A_{mn}} z^{-2}}. \tag{3.8}$$

A block diagram representation of the system can be seen in Figure 3.3:

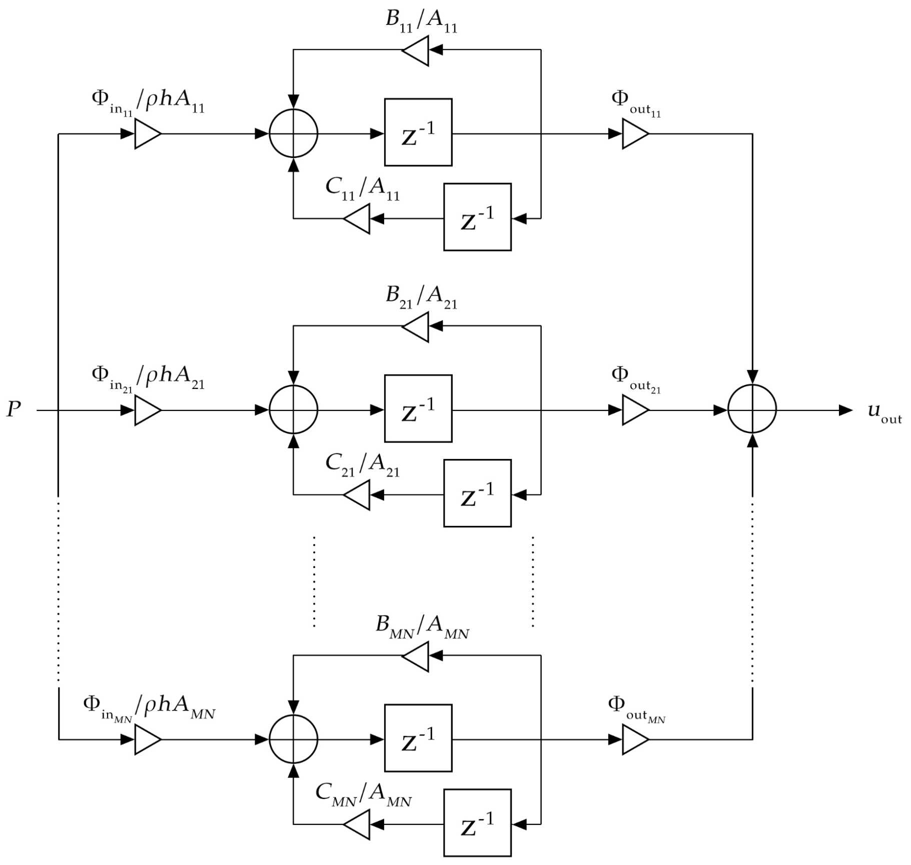


Figure 3.3: Block diagram representation of the update equation for all modes.

Essentially $H(z)$ is a collection of resonator filters that individually filter the signal according to the eigenfrequency of mode (m, n) and are added together in the end. The transfer function of a resonator filter (in general form) is [18]:

$$H_{\text{res}}(z) = \frac{G}{1 + a_1 z^{-1} + a_2 z^{-2}}, \tag{3.9}$$

where G is the gain of the filter and:

$$a_1 = -2R\cos(\omega_0) \quad \text{and} \quad a_2 = R^2.$$

Here, $0 < R < 1$ (for stability of the filter) and ω_0 is the resonant frequency (i.e., the location of the peak). In the range $\infty < \omega_0 < \infty$ it can be shown that the following condition should hold:

$$-2 < a_1 < 2. \tag{3.10}$$

If this is not true, R must be bigger than 1 making the filter unstable. The above condition can thus be regarded as a stability condition for the filter.

If we compare eqs. (3.8) and (3.9), we can see that:

$$G_{mn} = \frac{\Phi_{\text{in}_{mn}} \cdot \Phi_{\text{out}_{mn}}}{\rho h A_{mn}} z^{-1}, \quad a_1 = -\frac{B_{mn}}{A_{mn}}, \quad a_2 = -\frac{C_{mn}}{Amn}.$$

Notes: $\frac{B_{mn}}{A_{mn}}$ and $\frac{C_{mn}}{B_{mn}}$ are on the right side of the update equation, so their sign must be inverted. Moreover, G_{mn} is multiplied by z^{-1} which is not the case in the general transfer function. This does not change the frequency response of the filter, it merely applies a phase shift on the output signal.

From this, it be seen that the gain G_{mn} of each individual filter greatly depends on the input and output positions on the plate. If we look at C_{mn} and A_{mn} we find that their terms $\frac{1}{k^2}$ are a lot bigger than $\frac{c_{mn}}{\rho hk}$ making the latter a lot less influential for the general equation than the former. It can be found that for all modes:

$$-\frac{C}{A} \lesssim -1 \quad \Rightarrow \quad R = \sqrt{\frac{C}{A}} \gtrsim 1.$$

Also, $-\frac{B}{A}$ can be substituted in condition (3.10) resulting in:

$$-2 < -\frac{B}{A} < 2.$$

Filling in B and A, getting rid of the minus sign (as this will not change anything to the condition) and replacing $\frac{1}{k^2}$ with f_s^2 we get:

$$-2 < \frac{2f_s^2 - \omega^2}{f_s^2 + \frac{c}{\rho hk}} < 2.$$

As said before, the factor $\frac{c}{\rho hk}$ does not influence the equation much. Ignoring this term results in:

$$-2 < 2 - \frac{\omega^2}{f_s^2} < 2.$$

If we assume the eigenfrequencies ω to be real valued, the second term in the equation will always be negative. If we also assume ω to be non-zero, the equation will always result in a value lower than 2. This means that only following condition is left:

$$\begin{aligned} 2 - \frac{\omega^2}{f_s^2} &> -2, \\ -\omega^2 &> -4f_s^2, \\ \omega &< 2f_s. \end{aligned} \tag{3.11}$$

Equation (3.11) can be seen as the stability condition for the algorithm. (Also seen in [17])

3.2 Plugins

In this section, the SOTA regarding digital/VA plugins is described. Most plugins described could be downloaded and were informally evaluated (by the author) using Digital Audio Workstation *Garageband* [19].

3.2.1 Abbey Road Reverb Plates Plugin

In 2016, Audio Plugin company Waves introduced the Abbey Road Reverb Plates plugin which models the plate reverbs from the Abbey Road studios (see Figure 3.4) [20]. The Graphical User Interface (GUI) allows for the user to adjust the following parameters:

- Damper amount (0-10)
- Plate type (4 different kinds)
- Input/Output level (-Inf – 18 dB)
- Treble amount (-20 – +20 dB)
- Bass Cut (0 – 4)
- Predelay (0–500 ms)
- Dry/Wet (%)
- Drive (%)
- Analog (%)

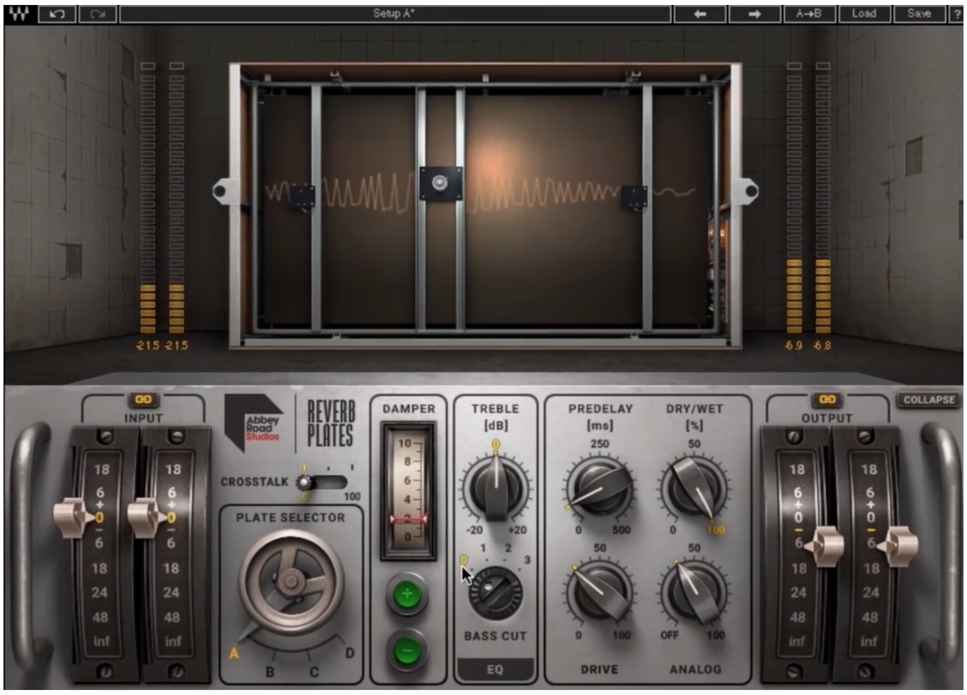


Figure 3.4: Abbey Road Plate Reverb plugin.

Evaluation

A trial version of the plugin could freely be downloaded from the Waves audio website. This evaluation is based on that.

The GUI works really well and nice sounds can be created with it. Unfortunately the only plate properties available are the ones derived from the plate reverbs used at the Abbey Road Studios making it not possible to change parameters like plate size and positioning of the input speaker and pickups. Of course, this is what needed to be achieved with this plugin, yet it decreases freedom which can be achieved using a VA implementation approach.

3.2.2 U-audio EMT140 Plugin

In 2010, Universal Audio introduced the EMT140 Classic Plate Reverberator Plug-In (see Figure 3.5) [21]. It is a convolution reverb [22] and the parameters that can be changed are:

- Input Filter
- Plate type (3 different kinds)
- Reverberation time (0–5.5 sec)

- More equalising and panning settings.

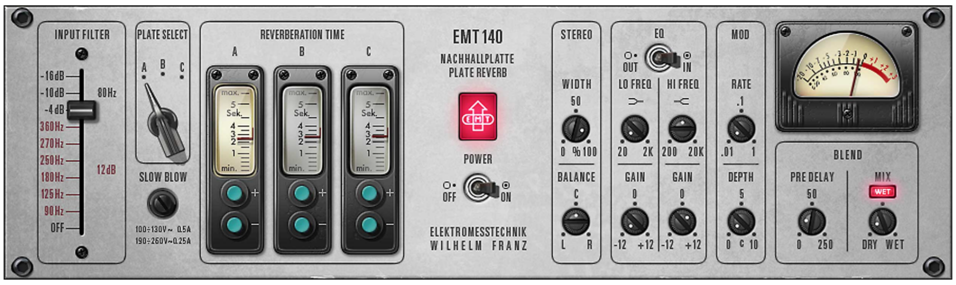


Figure 3.5: U-audio EMT140 plugin.

Evaluation

Although, no trial version could be found of this plugin, a demo video [4] could be used to evaluate the product.

In the video, the functionality of the plate reverb plugin is shown. Just as with the Waves Abbey Road plugin the freedom of plate properties is limited. In this plugin it is limited to three different plates formerly found at studio 'The Plant' in Sausalito, CA [4]. Also, in the opinion of the author, the difference between the dry signal and the signal after processing does not change that much. This could be due to the choice of sound in the video, which already contained some reverb if heard correctly.

The looks of the GUI have been chosen to match the original interface of the EMT140 which is a nice addition to the plugin.

3.2.3 PA1 Dynamic Plate Reverb

A very interesting solution found is the PA1 Dynamic Plate Reverb plugin made by Craig Webb (see Figure 3.6) [23]. It uses the Kirchhoff-Love equation to calculate the plate's modes of vibration and subsequently use that to calculate output at a given position from an input at another given position. Essentially, it adopts the modal description explained in Section 3.1.4, making it a VA plugin. With this plugin it is possible to change the following parameters:

- Material (Steel, Gold, Silver, Titanium, Aluminium)
- Plate size: changed by dragging the top-right corner (1.50×1.00 m – 3.00×2.00 m)
- Stereo pickup positions (anywhere on the plate)
- Tension of the plate (0 – 2000 N)

- Plate gain (0.5 – 2.0)
- Move input: which will move the input across the plate at a certain speed creating a flanging/vibrato effect (0–7 m/s)
- Stereo width (%)
- Pre-delay (2–200 ms)
- Dry/Wet (%)
- Frequency dependent decay of 8 different octave bands: changed by dragging dots resembling different bands (62.5, 125, 250, 500, 1000, 2000, 4000 and 8000 Hz) up and down (0.3–10 sec).

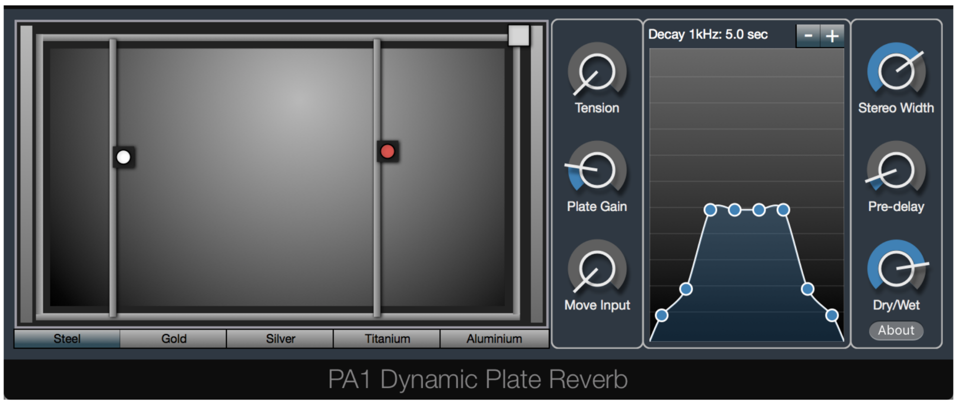


Figure 3.6: PA1 Dynamic Plate Reverb plugin.

Evaluation

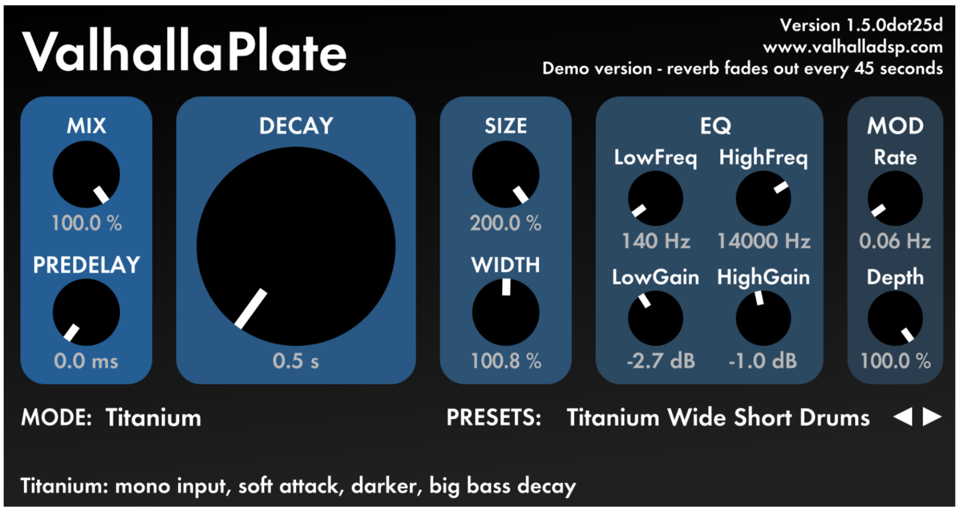
A beta version of the plugin could be downloaded for free. This evaluation is based on that.

This plugin has a large set of possibilities and sounds natural. The interface is clear and being able to visually change the pickup positions and the size of the plate is a nice touch to the plugin. Especially interesting are the parameters that are hard/impossible to change in a physical plate reverb. These are material, plate size, movement of the input, pickup position, tension of the plate and frequency dependent decay.

3.2.4 ValhallaPlate

In 2015, ValhallaDSP made a plugin called ValhallaPlate (see Figure 3.7) [24]. The parameters that could be changed by the plugin are:

- Mix (%)
- Predelay (0–500 ms)
- Decay (0.5–30 s)
- Size (0–200%)
- Width (0–200%)
- Equalisation settings
- Modulation rate (0.05–5Hz)
- Modulation depth (%)
- Mode (Option to change the material: Chrome, Steel, Cobalt, Brass, Aluminum, Copper, Unobtanium, Adamantium, Titanium, Osmium, Radium and Lithium)
- Presets


ValhallaPlate
Version 1.5.0dot25d
www.valhalladsp.com
Demo version - reverb fades out every 45 seconds

MIX 100.0 %
PREDELAY 0.0 ms
DECAY 0.5 s
SIZE 200.0 %
WIDTH 100.8 %
EQ LowFreq 140 Hz HighFreq 14000 Hz LowGain -2.7 dB HighGain -1.0 dB
MOD Rate 0.06 Hz Depth 100.0 %

MODE: Titanium
PRESETS: Titanium Wide Short Drums

Titanium: mono input, soft attack, darker, big bass decay

Figure 3.7: ValhallaPlate plugin.

Evaluation

A trial version of the plugin could freely be downloaded from the ValhallaDSP audio website. This evaluation is based on that.

ValhallaPlate is a very nice sounding reverb. It is a visually minimalistic and easy to use plugin but sounds very powerful. As the website states, it is based on the physical principles of plates (probably making it another VA solution). Also, the amount of materials that can be chosen are high. Another great power of the ValhallaPlate is the abundance of presets to which the user can also add their own creations.

3.2.5 CSR Plate Reverb

In 2011, IK Multimedia released a plugin called the CSR Plate Reverb (see Figure 3.8) [25]. The plugin is divided into an *easy* and an *advanced* panel where the former is a boiled down version of the latter. The parameters that could be changed in the easy-panel are:

- Mix (%)
- Diffusion (%)
- Reverb Time (0.105 – 23.56 s)
- Low Time (0.2–4.0x)
- High Frequency cutoff (20–20,000 Hz)
- High Frequency Damping (0.0–24.0dB (HICUT))
- Equalisation settings
- Modulation rate (0.05–5 Hz)
- Modulation depth (%)

The advanced panel is divided into parameters regarding the following topics (and their parameters):

- Input/Output (In Level, Out Level, Mix, In Image, Out Image)
- Time (Reverb time, Low time, Crossover, High Frequency Cutoff, High Frequency Damping Pre Delay)
- Reverb (Size, Diffusion, Buildup Disperse Modulation)
- Color (Low Cutoff Frequency, Low Cutoff Gain, High Cutoff Frequency, High Cutoff Gain)
- Reflections (Time Left, Level Left, Time Right, Level Right)
- Echo (Time Left, Feed Left, Time Right, Feed Right)

Lastly, there is a MOD-panel where LFOs and even envelopes can be applied to specific parameters.

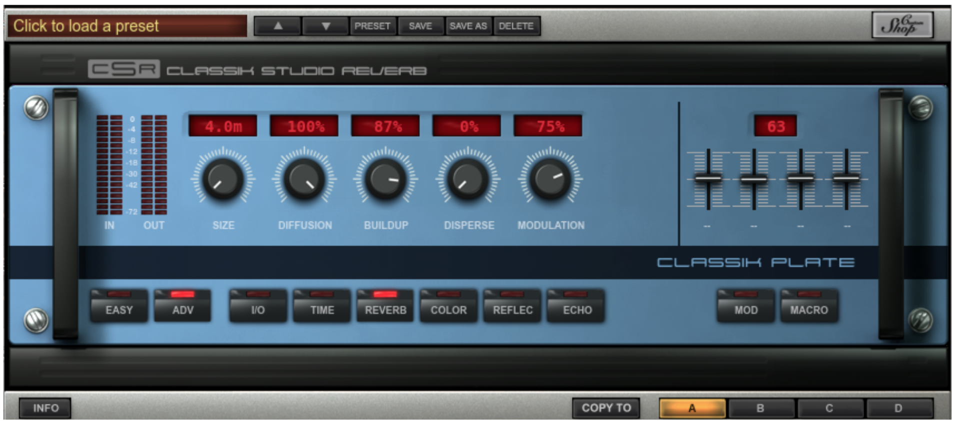


Figure 3.8: CSR Plate Reverb plugin.

Evaluation

A trial version of the plugin could freely be downloaded from the IK Multimedia audio website. This evaluation is based on that.

The power of the CSR Plate Reverb plugin is the large number of possibilities. The user can fine-tune the plate reverb exactly the way they want to and even apply LFOs and envelopes to the different effects. If the user is not an expert in this field, the easy-panel makes simple parameter tweaking possible.

Despite the possibilities, he reverb itself does not sound really natural. There are also no possibilities for changing the material or direct size of the plate (probably making this a convolution reverb).

3.2.6 Discussion Plugins

Considering all the above, the PA1 Dynamic Plate Reverb was found most interesting plugin. It was the most natural sounding and closer to a VA implementation than the others. It has been concluded that an implementation solution for this project would – in a certain sense – extend upon this plugin.

Even though the plugin is a perfect example of an already existing VA solution to the problem, not all possibilities that VA simulations pose on plate reverberation have been exploited. In the PA1 plugin, one can change the plate dimensions, but this does not affect the sound real-time in any way. When the program changes to the new dimensions, it can even be heard in the output sound. What the plugin does contain is the possibility to move the input source horizontally with a speed that the user can set. It does not, however, include moving outputs, let alone moving them in different patterns over the plate. In Chapter 5 these possibilities will be described and experimented with.

Chapter 4

Comparison of Different Implementation Approaches

This chapter compares different implementation approaches that Chapter 3 describes. All explorations have been carried out in the MATLAB environment on a MacBook Pro with a 2,2 GHz Intel Core i7 processor. At the end of the chapter, the explorations are discussed and a conclusion is given on what implementation approach would be most suitable for this project.

4.1 Finite-Difference Schemes

Below, the implementation of the finite-difference schemes explained in Section 3.1.3 is described.

For implementing a finite-difference scheme, the one found in [15] was used. Here, the authors implement a finite-difference scheme using several matrices containing the material properties and source position of the sound. The output would be a vector containing all values of one row of plate values. For this exploration, the scheme found in eq. (3.2) was implemented. Also, a visualisation of the plate was made (see Figure 4.1).

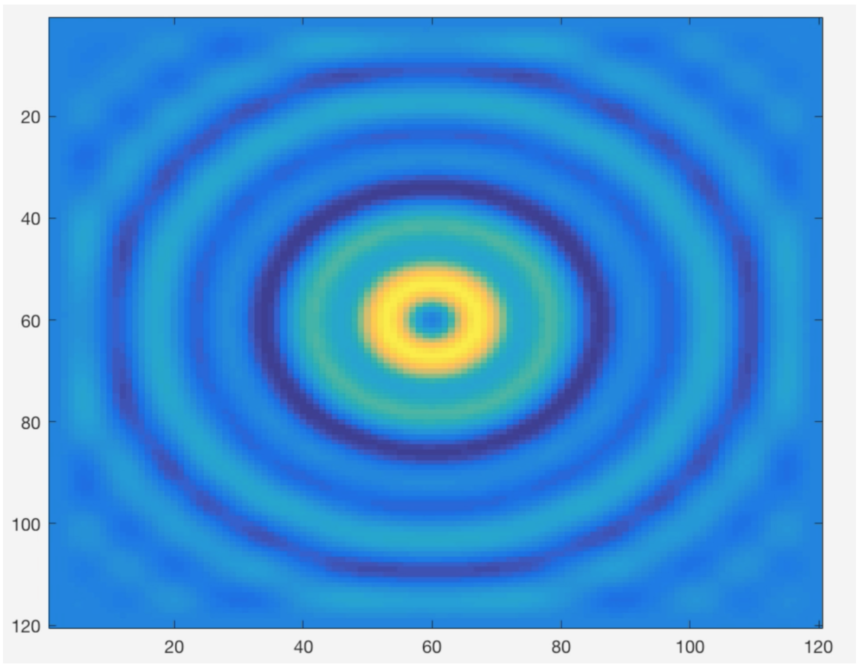


Figure 4.1: Visualisation of Finite-Difference Scheme (source in centre).

4.2 Reflection Model

A way to think about the plate model was to use the edges of the plate as mirrors for the input source (see Figure 4.2 for an illustration of this). An algorithm would be created that would include all reflected sources in a coordinate system. The sources would be located at points:

$$(\pm x_p + 2x \cdot L_x, \pm y_p + 2y \cdot L_y), \quad (x, y) \in \mathbb{Z}, \tag{4.1}$$

where (x_p, y_p) is the input source position. It would then be calculated how far each reflected source would be away from the output position(s). The delay and damping would be calculated for each individual source accordingly. The distances from the sources to the output would get closer and closer together the further the source is, differentiating between early and late reflections.

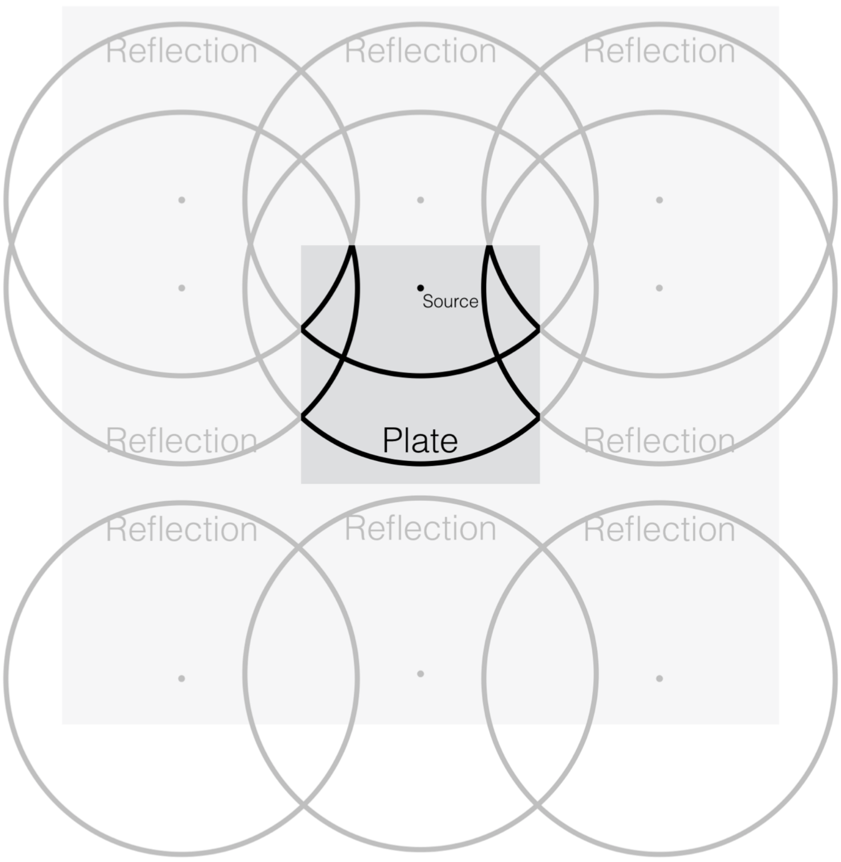


Figure 4.2: An illustration of the reflection model.

4.3 Modal Description

The modal description explained in Section 3.1.4 will be shortly described below. A more detailed exploration of this implementation approach can be found in Chapter 5.

As explained before, a modal description of a plate, decomposes it into several modes that each can be activated differently over time (recall eq. (3.3)). The amount that each mode is activated (q_{mn}), depends, recalling eq. (3.5), on the previous state of the plate, the input position of the actuator on the plate and the input signal itself. Retrieving the output happens by filling in q_{mn} for all modes into eq. (3.6) and setting a specific output position. All this means that only three points on the entire plate are needed to compute a stereo output using one input on the plate. It has been found that roughly 14 seconds were needed in order to compute 9.2 seconds of reverberated sound.

Next to that, it has been found very easy to change the parameters the plate depends on using this model.

4.4 Discussion and Conclusion

It was chosen not to investigate convolution, as it requires a recording of the impulse response (of which the physical parameters can not be changed) and is therefore not a VA solution. It could, however, be twisted into a VA solution when it is generated using physical parameters. Then in a hybrid structure as described by Abel in [10] it could be used and potentially become a VA solution. This is something that could be investigated in future work.

The finite-difference schemes worked quite well. The output is a sound that could clearly be heard as being reverberated. However, compared to some state of the art plugins, the output sounds really metallic and has a lot of high frequency content. Next to that, the implementation is not very fast. For a sound of 0.1 seconds and a grid of 100×100 points it takes almost 4 seconds to process. Every point on the grid of the plate has to be computed for every time instance in order for the scheme to work. For the eventual implementation (preferably) only input and output points need to be computed.

The reflection model was initially thought to drastically improve computational time as it would only take coordinates of reflected sources as input, but this was underestimated. Especially the fact that horizontal reflections would also be reflected vertically and vice versa. Also, as can be seen in Section 2.2, some frequencies travel at speeds higher than 600m/s. For a response of, for instance, only 5 seconds, the algorithm would need to include all reflections in a radius of more than 3000 meters from the source. Using a plate of 2×1m this would mean that the algorithm would need to account for at least 14,137,067 reflected sources[1]! If the distances to both outputs would be calculated, this number would be even twice as large. This would be (way) too much data to result in a computationally efficient model.

The implementation approach that was eventually chosen is the modal description. Compared to the aforementioned approaches, it only requires three points on the plate to fully generate a stereo output. Looking at the computational times of the different approaches, it can be seen that this results in a significantly faster approach. Next to that, as mentioned above, parameters are very easy to change using this model. This is promising for eventually implementing extensions stated in the second project goal in Chapter 1.

[1]Calculation: $A \cdot S_\text{d} = \pi \cdot 3000^2 \cdot \frac{1}{2}$ where A is the area of the circle in m^2 and S_d is the source density in 1/m^2.

Chapter 5

Implementation of a VA Plate Reverb based on a Modal Description

This chapter describes an implementation based on the theory found in Section 3.1.4. It gives a detailed explanation of the created implementation with the help of code snippets for the more complex parts of the algorithm. The raw code can be found in Appendix B.1.

5.1 Initialisation

At the start of the algorithm, many parameters need to be initialised. For testing purposes, it has been chosen to use the properties of the EMT140 plate reverb in the algorithm (which can obviously be changed). Therefore, the plate width and height are set to: $L_x = 2$ m, $L_y = 1$ m respectively. If we recall eq. (2.2) where stiffness parameter κ^2 is calculated, we need the Young's modulus, the thickness of the plate, the density of the material and Poisson's ratio. For steel, with a thickness of 0.5 mm, these parameters are: $E = 2 \cdot 10^{11}$ N/m^2, $h = 0.0005$ m, $\rho = 7850$ kg/m^2 and $\nu = 0.3$ respectively. Using `kSquared = (E*h^2)/(12*rho*(1-v^2))`, κ^2 is calculated. The input and left and right outputs on the EMT140 are at the following positions [9]: $(x_p, y_p) = [0.4L_x, 0.415L_y]$, $(x_l, y_l) = [0.1L_x, 0.45L_y]$ and $(x_r, y_r) = [0.85L_x, 0.45L_y]$. Furthermore, the air density and speed of sound (in air) are needed to calculate the radiation damping α_{rad} in eq. (2.6). These are set to $\rho_{\text{a}} = 1.225$ kg/m^2 and $c_{\text{a}} = 343$ m/s respectively. Finally a sample rate $f_{\text{s}} = 44{,}100$ Hz is used, and a sound file is chosen as input. This file will be zero-padded (in this case, 5 seconds or 220,500 samples) it will leave room for a long reverberation time in the output signal.

5.2 Calculate Eigenfrequencies

The (angular) eigenfrequencies ω_{mn} can be calculated using the following equation [26]:

$$\omega_{mn} = \kappa\pi^2\left(\frac{m^2}{L_x^2} + \frac{n^2}{L_y^2}\right). \tag{5.1}$$

Recalling condition (3.11), we can see that this poses a limit on the total number of modes in (3.5) and a creates the dependency:

$$\omega_{M(n)m}, \omega_{mN(m)} < 2f_s, \tag{5.2}$$

where $m \in [1, M(n)]$ and $n \in [1, N(m)]$. All the eigenfrequencies that satisfy this condition will be used in the algorithm (see Figure 5.1 as an illustration for this). Using a sample rate of 44,100 Hz the highest stable eigenfrequency will be $\frac{2\cdot 44{,}100\text{ Hz}}{2\pi} = 14{,}037$ Hz.

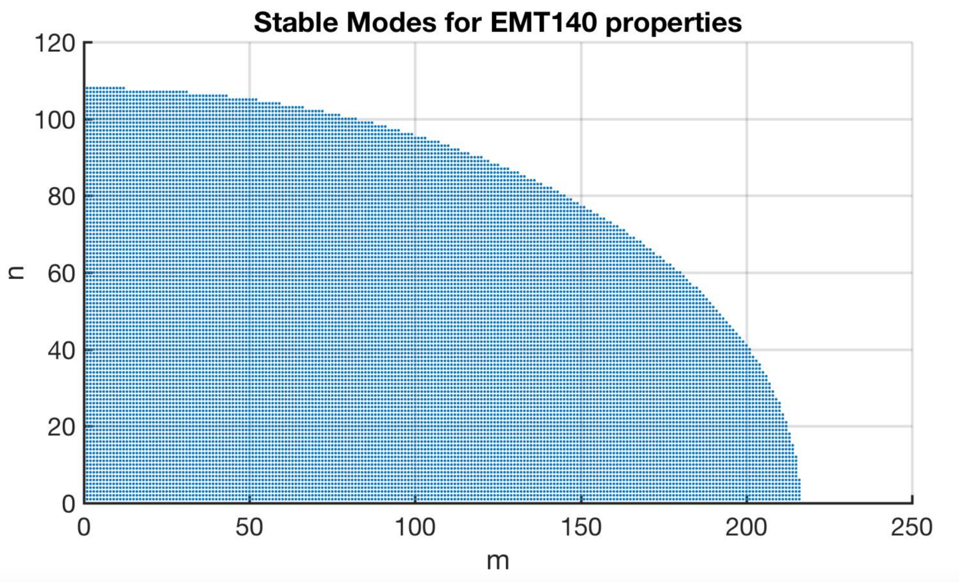


Figure 5.1: Visualisation of stable modes using EMT140 properties. Each dot represents a single mode (m, n) that, when inserted into eq. (5.2), satisfies the condition $\omega_{mn} < 2f_s$.

In the algorithm, an `omega` matrix is created where the first column contains all eigenfrequencies (ω) and the second and third column the corresponding horizontal (m) and vertical (n) modes respectively. In the code below, the two latter are are defined as `m1` and `m2` respectively.

```
%% Create Eigenfrequencies
disp('Create Omega')
val = 0;
m = 1;
m1 = 1;
m2 = 1;
omega = zeros(100000,3); %set to zeros for program speed
while val < fs*2 %check for stability
    val = ((m1/Lx)^2 + (m2/Ly)^2)*sqrt(kSquared)*pi^2; %calculate ...
        eigenfrequency
    %m1 is fixed, increase m2 until above stability condition
    if val < fs*2 %double check for stability
        omega(m,1) = val; %first column: eigenfrequency
        omega(m,2) = m1; %second column: horizontal mode
        omega(m,3) = m2; %third column: vertical mode
        m2 = m2 + 1;
        m = m+1;
    else
        if m1 == 1
            stablem2 = m2-1; %set highest stable vertical mode
        end
        m2 = 1; %reset m2
        m1 = m1 + 1; %increment m1
        val = ((m1/Lx)^2 + (m2/Ly)^2)*sqrt(kSquared)*pi^2; %check ...
            if (m1,1) > 2fs
        if val > fs*2
            stablem1 = m1 - 1; %set highest stable horizontal mode
            break; %if (m1,1)> 2fs , break out of the loop
        end
    end
end
omega = omega(1:m-1,:); %get rid of zeros
```

5.3 Calculate In- and Output Vectors

In the algorithm, three vectors are created, Φ_{in}, Φ_{outL} and Φ_{outR} (`phiIn`, `phiOutL` and `phiOutR`) are calculated using eq. (3.4). The modes corresponding to the eigenfrequencies calculated in the previous section (`omega(:,2)` and `omega(:,3)`) will be inserted as m and n in this equation. In the code below, the values of `in`, `outL` and `outR` are the relative positions of the in- and outputs on the plate and are normalised back to the plate dimensions in the calculation below.

```
%% Create PhiIn and -Out
M = length(omega(:,1));
phiIn = zeros(M,1);
phiOutL = zeros(M,1);
phiOutR = zeros(M,1);
```

```
for m = 1:M
    phiIn(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*in(1))/Lx)*...
        sin((omega(m,3)*pi*in(2))/Ly);
    phiOutL(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*outL(1))/Lx)*...
        sin((omega(m,3)*pi*outL(2))/Ly);
    phiOutR(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*outR(1))/Lx)*...
        sin((omega(m,3)*pi*outR(2))/Ly);
end
```

5.4 Calculate Loss Coefficients

In [17], the authors set loss coefficients per frequency band (also see Section 3.2.3). Here, the thermoelastic and radiation damping mechanisms described in Section 2.3 have been implemented to ultimately create a loss coefficient for each individual eigenfrequency. It was chosen to leave out the damping induced by a porous medium, as it can be ignored when set to the furthest possible distance and is thus no internal damping mechanism in a thin metal plate. In Figure 5.2 the results can be seen.

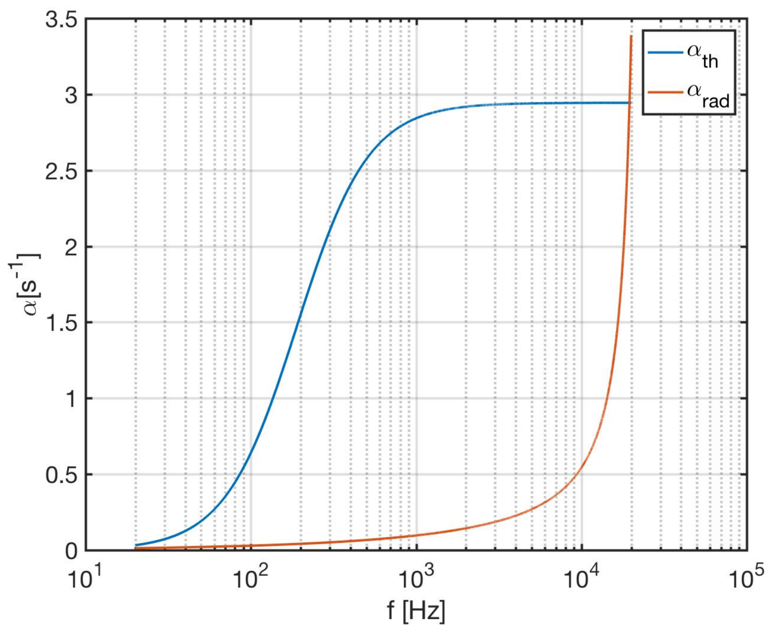


Figure 5.2: Calculated thermoelastic and radiation damping for EMT140 properties.

The loss coefficients are then calculated in the following way [9, 17]:

$$c_{mn} = \frac{12\ln(10)}{T_{60}} \quad \text{where} \quad T_{60} = \frac{3\ln(10)}{\alpha_{\text{tot}}} \quad \text{gives} \quad c_{mn} = 4\alpha_{tot}. \tag{5.3}$$

Here, α_{tot} is the addition of all damping factors.

5.5 The Main Loop

Now, with everything initialised, the reverberated output can be calculated using update equation (3.5). In the code below, the vectors `factorBdA`, `factorCdA` and `factorIndA` are $\frac{B}{A}$, $\frac{C}{A}$ and $\frac{\Phi_{\text{in}}}{\rho h A}$ found in Section 3.1.4.

As q^{t+1} is a state in the future, the entire equation is shifted back one sample. This causes P^t to become P^{t-1} in the update equation. Because at the very first time instance, there is no 'previous' input signal, P^0 is set to 0.

```
%% Main loop
disp('Loop');
output = zeros(2,length(input)); %initialise output
output2 = zeros(2,length(input)); %initialise normalised output

for t = 1:length(input)
    %the update equation
    if t == 1 %there is no value for t-1 at t=1, so set input to 0
        qNext =(factorBdA.*qNow+factorCdA.*qPrev+factorIndA.*0);
    else
        qNext ...
            =(factorBdA.*qNow+factorCdA.*qPrev+factorIndA.*input(t-1));
    end

    %fill in the output at sample t
    output(1,t) = qNext'*phiOutL;
    output(2,t) = qNext'*phiOutR;

    %update qVectors
    qPrev = qNow;
    qNow = qNext;
end
```

After the loop is done, a normalised version of the output is created:

```
% Normalise Output
output2 = zeros(2,length(output));
for i = 1:2
    output2(i,:) = output(i,:)/max(abs(output(i,:)));
end
```

5.6 Extensions: Dynamic Outputs

The PA1 Dynamic Plate Reverb Plugin (as described in Section 3.2.3) has the functionality of moving the input from left to right over the plate to create a flanging effect. Here, not only both outputs (not inputs) will be able to move, but they will be able to move in circular and even Lissajous patterns. This happens according to the following formulas:

$$x_{\text{out}}(t) = R_x L_x \sin\left(S_x \cdot \frac{2\pi t}{f_s} + \theta_x\right) + 0.5, \tag{5.4}$$

$$y_{\text{out}}(t) = R_y L_y \sin\left(S_y \cdot \frac{2\pi t}{f_s} + \theta_y\right) + 0.5. \tag{5.5}$$

Here, $-0.5 < R_x < 0.5$ and $-0.5 < R_y < 0.5$ when multiplied with L_x and L_y respectively, determine the horizontal and vertical maxima of the output pattern. If one of these has a value of zero, the outputs will move in a linear fashion. The shape of the pattern is determined by the horizontal and vertical speeds S_x, S_y and the horizontal and vertical phase shifts θ_x and θ_y respectively. For example if $S_x = S_y = 1$, $\theta_x = 0$ and $\theta_y = 0.5\pi$ the outputs will follow an elliptical pattern. The patterns created for different values of S_x, S_y and θ_x, θ_y can be seen in [27]. If either S_x or S_y has a value of zero, the outputs will again move in a linear fashion. Note that, though not implemented, it could be perfectly possible to set different values for the aforementioned parameters for the left output channel and the right.

All possible output positions are precomputed in order to improve computational time. A value of 10,000 coordinates/meter is chosen as this has been found to be a good tradeoff between speed and quality of the output sound. The centres of the patterns have been set to the middle of the plate. In the algorithm, the code in Section 5.3 is changed to:

```matlab
%% Create PhiIn
M = length(omega(:,1));
phiIn = zeros(M,1);
for m = 1:M
    phiIn(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*in(1))/Lx)*...
        sin((omega(m,3)*pi*in(2))/Ly);
end


%% Create PhiOut
phiOutL = zeros(M,1);
phiOutR = zeros(M,1);
disp('Create PhiOut')
```

```
if flanging == true %if true, precompute phiOutL and R for all ...
    possible output positions
    % Set up Moving Outputs
    disp('Set up Moving Outputs')
    %set shape extremes
    Rx = 0.4;
    Ry = 0.4;

    xPoints = 2*Rx*Lx;
    yPoints = 2*Ry*Ly;

    outputPointsX = 0:1/(10000*xPoints):1;
    outputPointsY = 0:1/(10000*yPoints):1;
    outputPointsX = (outputPointsX-0.5)*xPoints+0.5;
    outputPointsY = (outputPointsY-0.5)*yPoints+0.5;

    %set x and y speeds
    Sx = 4;
    Sy = 3;

    %set thetax and y
    thetax = 0;
    thetay = 0.5*pi;

    %Create possible output positions
    circX = outputPointsX(ceil(length(outputPointsX)*...
        (Rx*sin(Sx*2*pi*(1:2/max([Lx Ly]):fs)/fs + thetax)+0.5)));
    circY = outputPointsY(ceil(length(outputPointsY)*...
        (Ry*sin(Sy*2*pi*(1:2/max([Lx Ly]):fs)/fs + thetay)+0.5)));

    %Set speeds for left and right output
    Lspeed = 50;
    Rspeed = 30;
    phiOutLPre = zeros(M,length(circX));
    phiOutRPre = zeros(M,length(circX));
    for t = 1:length(circX)
        for m = 1:M
            phiOutLPre(m,t) = ...
                (4/(Lx*Ly))*sin((omega(m,2)*pi*circX(t))/Lx)*...
                sin((omega(m,3)*pi*circY(t))/Ly);
            phiOutRPre(m,t) = ...
                (4/(Lx*Ly))*sin((omega(m,2)*pi*circX(t))/Lx)*...
                sin((omega(m,3)*pi*circY(t))/Ly);
        end
    end
else %if false, create PhiOutL and R for their set output positions
    for m = 1:M
        phiOutL(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*outL(1))/Lx)*...
            sin((omega(m,3)*pi*outL(2))/Ly);
        phiOutR(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*outR(1))/Lx)*...
            sin((omega(m,3)*pi*outR(2))/Ly);
    end
end
```

In the loop, the dynamic outputs are accounted for by adding the code below before line 14 in the main loop (see previous section). Even though the left and right output follow the same pattern, they can be set to go at different speeds using the `Lspeed` and `Rspeed` variables.

```
%% Flanging
if flanging == true
    phiOutL = phiOutLPre(:,floor(mod(t/Lspeed,length(circX))+1));
    phiOutR = phiOutRPre(:,floor(mod(t/Rspeed,length(circX))+1));
end
```

5.7 Extensions: Changing Plate Dimensions

The fact that VA simulations are extremely flexible, poses a lot of interesting opportunities for model manipulation. The most interesting parameters to manipulate are those that are physically 'fixed', such as the dimensions of a steel plate. Generally speaking, changing the dimensions of a steel plate is physically impossible to do. The challenge was thus to imagine how the sound would change if it would be possible. Looking at the variables that are dependent on the horizontal (L_x) and vertical (L_y) plate dimensions, it can be seen that changing these would change the eigenfrequencies, the modal shapes and radiation damping. As can be seen in eq. (5.1) the eigenfrequencies lower as L_x and L_y grow. The thickness of the plate can also be changed. This changes the stiffness factor κ^2 in eq. (2.2) which then changes the eigenfrequencies again. It can be shown that a decrease in thickness will lower the eigenfrequencies. In the algorithm, all this is accounted for by introducing new variables `Lxnew`, `Lynew` and `hNew` which are initially set to the original values of their fixed counterparts. Below, some examples can be seen (where `start = ...` `44100` samples) that, when added to the loop, will change the dimensions of the plate over time:

```
%change dynamic variables over time
Lxnew = Lx*(1+(t-start)/44100); %change in width
Lynew = Ly*(1-(t-start)/(4*fs)); %change in height
hNew = h*(1-(t-start)/(4*fs)); %change in thickness
kSquared = (E*hNew^2)/(12*rho*(1-v^2)); %update stiffness factor
```

These variables are then used in eq. (5.1) and change the eigenfrequencies accordingly:

```matlab
%change eigenfrequency values according to Lxnew and Lynew
for i = 1:length(omega(:,1))
    omega(i,1)=(((omega(i,2)*pi)/Lxnew)^2 + ...
        ((omega(i,3)*pi)/Lynew)^2)*sqrt(kSquared);
end
```

Given the stability condition (5.2), the number of eigenfrequencies that can be accounted for also grows as the plate dimensions grow. In the algorithm, this is implemented by adding zero values to q^{t+1}, q^{t} and q^{t-1} (the *q-vectors*) in eq. (3.5), where the combinations mn were non-existent in the previous update. In the algorithm the highest stable modes are found using the following code:

```matlab
% calculate highest stable modes
m1check = 1;
m1Prev = stablem1;
m2Prev = stablem2;
stableValm1 = 0;
stableValm2 = 0;

% calculate highest stable modes
m1Var = 0;
while m1Var == 0
    stableValm1 = ((m1check*pi/Lxnew)^2 + ...
        (1*pi/Lynew)^2)*sqrt(kSquared);
    if stableValm1 > fs*2
        m1Var = 1;
    else
        m1check = m1check+1;
    end
end
stablem1 = m1check-1;
m2check = 1;
m2Var = 0;
while m2Var == 0
    stableValm2 = ((1*pi/Lxnew)^2 + ...
        (m2check*pi/Lynew)^2)*sqrt(kSquared);
    if stableValm2 > fs*2
        m2Var = 1;
    else
        m2check = m2check+1;
    end
end
stablem2 = m2check-1;
```

If the plate decreases in size, the values of the *q-vectors* for which the combinations mn do not satisfy the stability condition anymore will be removed. The code for both cases can be seen below. As this code is computationally heavy, it will only be checked every 100th sample. This number has been chosen as this has been found to be a good tradeoff between speed and quality of the output sound.

```
%% Check for unstable eigenfrequencies (happens when decreasing Lx ...
    or Ly and increasing h)
tooHighVect = find(omega(:,1) > fs*2);
if ~isempty(find(omega(:,1) > fs*2,1))
    % delete if omega > 2fs
    omega(tooHighVect,:) = [];
    qNext(tooHighVect,:)= [];
    qNow(tooHighVect,:) = [];
    qPrev(tooHighVect,:) = [];
else
    %% Add eigenfrequencies that are now stable (happens when ...
        increasing Lx or Ly and decreasing h)
    if m1Prev < stablem1 || m2Prev < stablem2
    % if highest stable mode has changed create new omega matrix
        omegaPrev = sortrows(omega,[2,3]); %set for comparison later
        omega = zeros(100000,3);
        val = 0;
        m = 1;
        m1 = 1;
        m2 = 1;
        while val < fs*2
            val = ((m1*pi/Lxnew)^2 + ...
                (m2*pi/Lynew)^2)*sqrt(kSquaredNew);
            if val < fs*2
                omega(m,1) = val;
                omega(m,2) = m1;
                omega(m,3) = m2;
                m2 = m2 + 1;
                m = m+1;
            else
                m2 = 1;
                m1 = m1 + 1;
                val = ((m1*pi/Lxnew)^2 + ...
                    (m2*pi/Lynew)^2)*sqrt(kSquaredNew);
                if val > fs*2
                    break;
                end
            end
        end
        omega = omega(1:m-1,:);
        %% OPTION ONE: stable horizontal mode increased
        if m1Prev < stablem1
            index = 0;
            for j = 1:length(omega)
                if j-index < length(omegaPrev)
                    if omegaPrev(j-index,2) ~= omega(j,2)
                        %insert 0's at right locations
                        qNext = [qNext(1:j-1,1); 0; qNext(j:end,1)];
                        qNow = [qNow(1:j-1,1); 0; qNow(j:end,1)];
                        qPrev = [qPrev(1:j-1,1); 0; qPrev(j:end,1)];
                        index = index + 1;
                    end
                else
                    %insert 0's at the end
                    qNext = [qNext ; ...
                        zeros(length(omega)-length(qNext),1)];
                    qNow = [qNow ; ...
                        zeros(length(omega)-length(qNow),1)];
                    qPrev = [qPrev ; ...
                        zeros(length(omega)-length(qPrev),1)];
                end
            end
        else
            %% OPTION TWO: stable vertical mode increased
            if m2Prev < stablem2 % Just adding zeros instead of ...
                this smart-adding gives metallic sounds
                index = 0;
                for j = 1:length(omega)
                    if j-index < length(omegaPrev)
                        %insert 0's at right locations
                        if omegaPrev(j-index,2) ~= omega(j,2)
                            qNext = [qNext(1:j-1,1); 0; ...
                                qNext(j:end,1)];
                            qNow = [qNow(1:j-1,1); 0; qNow(j:end,1)];
                            qPrev = [qPrev(1:j-1,1); 0; ...
                                qPrev(j:end,1)];
                            index = index + 1;
                        end
                    else
                        %insert 0's at the end
                        qNext = [qNext ; ...
                            zeros(length(omega)-length(qNext),1)];
                        qNow = [qNow ; ...
                            zeros(length(omega)-length(qNow),1)];
                        qPrev = [qPrev ; ...
                            zeros(length(omega)-length(qPrev),1)];
                    end
                end
            end
        end
    end
end
```

Because the above code is not called every sample (but every 100th), it needs to be called when the 'stretching' stops. If it does not do this, the update equation will not rely on the current (changed) parameters and can become unstable. Therefore, it runs through it one extra time when it is done.

Lastly, the different kinds of damping – the thermoelastic damping α_{th} in eq.

(2.5) and the radiation damping α_{rad} in eq. (2.6) and with that A, B and C in Section 3.1.4 – are also updated according to the new dimensions.

5.8 Extra: MATLABs Filter Function

As an alternative to the update equation, the input can be filtered with $H(z)$ found in the *Resonator Filter* part of Section 3.1.4. The input will be filtered using the coefficients of every individual mode. All these 'outputs' will ultimately be added together to create an identical output to the update equation. In the code this looks like:

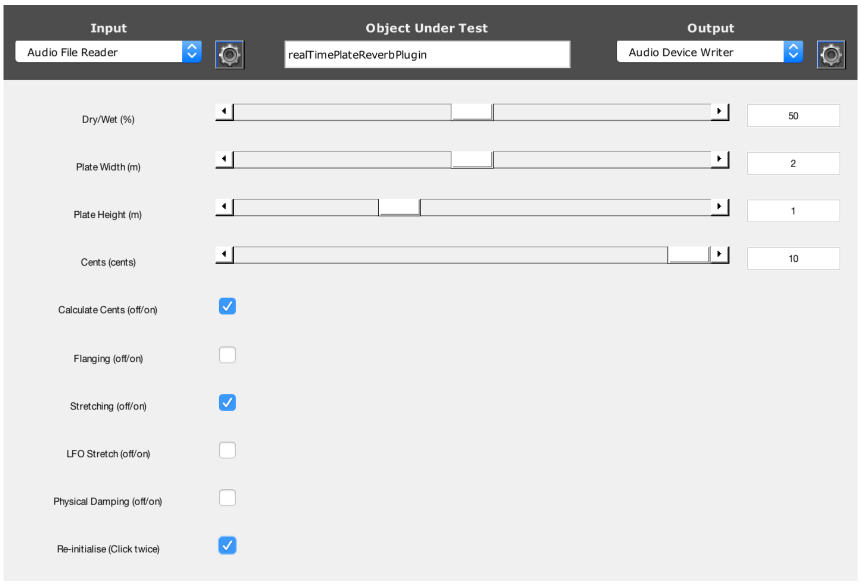
```
%initialise filter coefficients
b = [0 0];
a = [0 0 0];
for mode = 1:length(omega(:,1))
    b = [0 factorIn(mode)];
    a = [1 -factorBdA(mode) -factorCdA(mode)];
    output(1,:) = output(1,:) + phiOutL(mode).*filter(b,a,input)';
    output(2,:) = output(2,:) + phiOutR(mode).*filter(b,a,input)';
end
```

This method is a lot slower than the update equation: it takes roughly 3 minutes to process 9.2 seconds of sound.

Chapter 6

Real-Time Plugin

This chapter describes the real-time plugin that was made based on the implementation described in the previous chapter. As the algorithm was still computationally heavy, some adaptations have been made to it. These will also be described in this chapter. The plugin was developed using the Audio System Toolbox in the MATLAB environment and its layout can be seen in Figure 6.1. It has been tested using a MacBook Pro containing a 2,2GHz Intel Core i7 processor. Its raw code can be found in Appendix B.2.



Input	Object Under Test	Output
Audio File Reader	realTimePlateReverbPlugin	Audio Device Writer

Dry/Wet (%)	50
Plate Width (m)	2
Plate Height (m)	1
Cents (cents)	10
Calculate Cents (off/on)	✓
Flanging (off/on)	☐
Stretching (off/on)	✓
LFO Stretch (off/on)	☐
Physical Damping (off/on)	☐
Re-initialise (Click twice)	✓

Figure 6.1: Plugin Layout.

6.1 Controls and Layout

The main controls of the plugin are:

- The *Dry/Wet*-ness of the signal ranging from 0 - 100%. This simply adds the input to the reverberated output in the following way: $(1 - DW) \cdot \text{input} + DW \cdot \text{output}$ where DW is the *Dry/Wet*-value ranging from 0–1.
- The *Plate Width* and *Height* ranging from 1.0 – 3.0 m and 0.5 – 2.0 m, respectively.
- The amount of *Cents* in between each eigenfrequency ranging from 0.01 to 10.0 cents (see Section 6.2.3 for a detailed explanation on this).

Next to that, there are switches which turn off/on:

- The calculation of *Cents* (Calculate Cents)
- Movement of the pickups over the plate (Flanging)
- The option of hearing the change in plate dimensions in real-time (Stretching)
- An LFO that stretches the plate over time (LFO Stretch)
- The thermoelastic and radiation damping (Physical Damping). (see Chapter 8 for a detailed explanation on this matter) *Note: when turned off, T_{60} is set to 4 seconds for all frequencies.*

The sixth button (Reinitialise) can be used to reset the algorithm if, for example the *Plate Width* or *Height* have been changed (also see Section 6.4. As continuously updating the eigenfrequencies causes artefacts in the sound, it has been chosen to only update these when this button is pressed.

Note: It would, however, be possible to include a library with many different settings for the plate and load these instead of calculating every change at the spot. This has not yet been investigated and could be implemented in future work.

6.2 Computational Speed

As stated in Section 3.1.4, the computational speed depends on the total number of eigenfrequencies ($M(n)$ and $N(m)$) being accounted for in the algorithm. A correlation of 0.996 has been found between this and program speed, making decreasing this amount the main focus. This number has been found by using MATLABs tic-toc function. The number of modes would be set and the loop would be run roughly 50 times. The average time was taken and linked to the number of modes. After a few different settings of modes, the correlation was calculated.

As stated in Section 4.3, it takes roughly 14 seconds to compute 9.2 seconds of sound. The total number of eigenfrequencies accounted for in the algorithm using EMT140 properties is 18,218. Below, a few ways are described to decrease this number, without greatly affecting the (perceived) output sound. They have been added to the algorithm right after the eigenfrequencies are calculated. All computational times have been retrieved by using MATLABs tic-toc function in the algorithm.

6.2.1 Ignore Unactivated Modes

The input position (x_p, y_p) greatly determines which modes are activated and which are not. If we, for example, let $x_p = 0.25L_x$, it can be shown that in eq. (3.4), the term $\sin\left(\frac{m\pi x_p}{L_x}\right) = 0$ for $m = 4, 8, 12...$ etc., i.e., multiples of 4. This makes the eventual state variable u in eq. (3.3) 0 at all times for those specific modes. (It also makes sense when looking at eq. (3.9), where if $\Phi(x_p, y_p) = 0$ for a certain mode, the gain G of that mode will also equal 0.) The reason for this fact, is that these modes have a node at this specific input position and are thus not activated.

The way that this is implemented in the algorithm is by first calculating how many times $\frac{x_p}{L_x}$ and $\frac{y_p}{L_y}$ need to be multiplied until they are integers. Then these values x_{int} and y_{int} can be used to find the unactivated modes. All eigenfrequencies with modes that satisfy:

$$\text{mod }(m, x_{\text{int}}) = 0 \qquad || \qquad \text{mod }(n, y_{\text{int}}) = 0, \tag{6.1}$$

can be ignored. Using this condition and the input position used by the EMT140 $(0.4Lx_, 0.415L_y)$, the total number of eigenfrequencies can be reduced to 14,614 (see Figure 6.2).

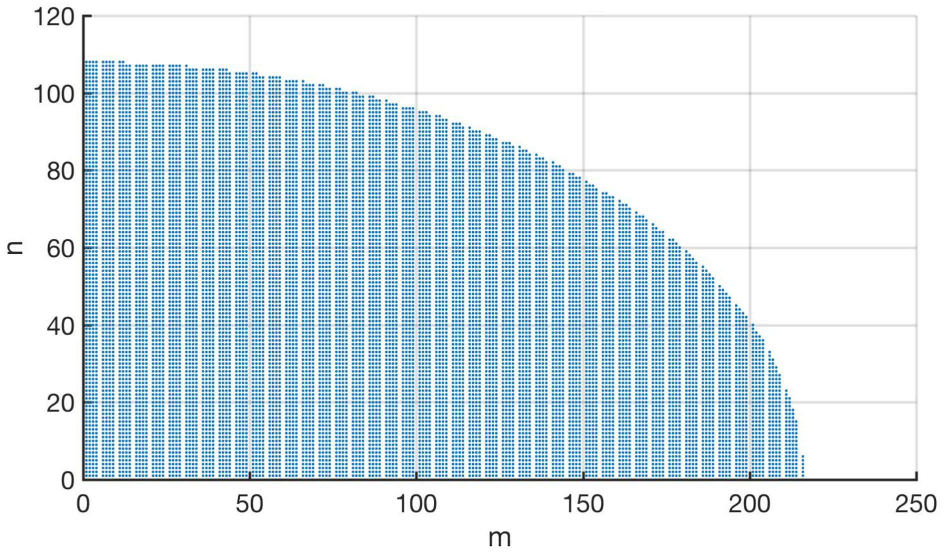


Figure 6.2: Visualisation of modes accounted for where unactivated modes are ignored (using EMT140 properties).

Now, the algorithm only needs roughly 12 seconds to compute 9.2 seconds of sound. Using an input position of $(0.5L_x, 0.5L_y)$ deletes almost three quarters of the total the number of modes, resulting in a total amount of 4,593 (see Figure 6.3).

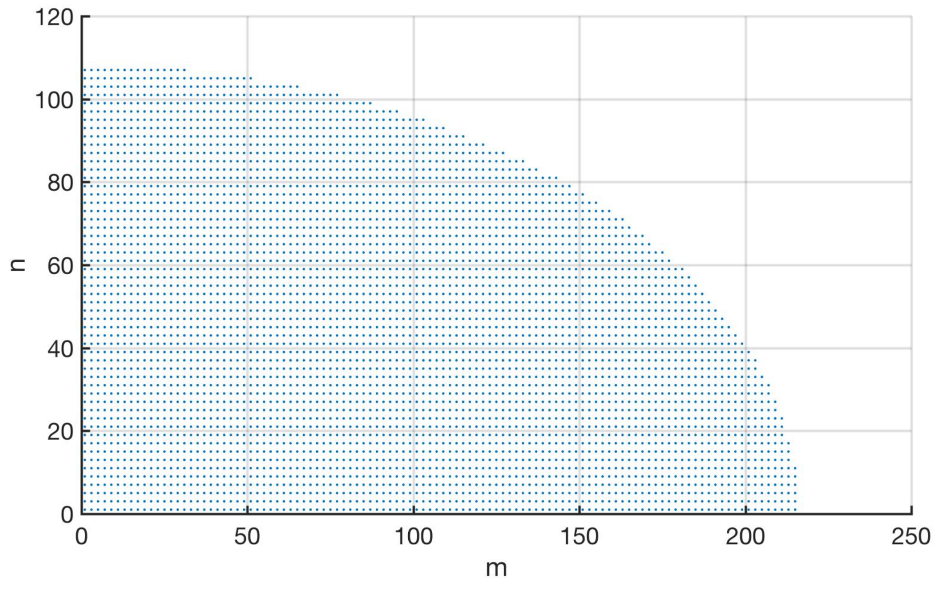


Figure 6.3: Visualisation of modes accounted for where unactivated modes are ignored (using an input at $(0.5L_x, 0.5L_y)$).

This is because every even mode in both the x and y-direction can be ignored as they have a node at this position. This special case reduces the computational time to about 5.7 seconds. Below, the implementation in the algorithm can be seen.

```
%% Delete Neclegible Modes From Input
if delModes == true
    disp('Delete neclegible modes from input')
    i1 = 1;
    answ1 = p(1);
    %multiply relative position x with integers until the answer ...
        becomes an integer
    while rem(answ1,1) ~= 0
        i1 = i1+1;
        answ1 = p(1)*i1;
    end
    %multiply relative position y with integers until the answer ...
        becomes an integer
    i2 = 1;
    answ2 = p(2);
    while rem(answ2,1) ~= 0
        i2 = i2+1;
        answ2 = p(2)*i2;
    end
    %discard the eigenfrequencies accordingly
    n = 1;
    while n <= length(omega(:,1))
        if mod(omega(n,2), i1) == 0 || mod(omega(n,3), i2) == 0
            omega(n,:) = [];
        else
            n = n + 1;
        end
    end
end
```

The same can be done for the output positions. However, because there are multiple outputs they need to be located on the same nodes, i.e., they must share either their vertical or their horizontal position, or a multiple of this. In other words, most of the time, modes for a single output can not be discarded as the other output might need it. On top of that, when the flanging effect is turned on, the nodes that the outputs It has thus been decided not to discard modes based on output position.

6.2.2 Remove Dependency

What the authors do in [17] is to only include the eigenfrequencies $m \in [1, M]$ and $n \in [1, N]$ where M and N are fixed, as opposed to the dependency seen in condition (5.2). As the authors do not make it clear how they find M and N, their values were chosen so to maximise the total number of eigenfrequencies. This was done by multiplying m with n for the eigenfrequencies that satisfy the stability condition (5.2), selecting m and n that maximise this answer and setting M and N to these values. All eigenfrequencies that have higher mode-values than these are then discarded. See the code below.

```
%% Remove dependency
if square == true
    highMode = omega(:,2).*omega(:,3);
    indexFound = find(highMode==max(highMode(:))); %find highest ...
        possible MN
    i = 1;
    maxM1 = omega(indexFound(1),2);
    maxM2 = omega(indexFound(1),3);
    while i <= length(omega(:,1))
        % if an eigenfrequency has a mode-value higher than the ...
            maximum, discard
        if omega(i,2) > maxM1 || omega(i,3) > maxM2
            omega(i,:) = [];
        else
            i = i + 1;
        end
    end
end
```

This action reduces the number eigenfrequencies to 11,628 and the computational time to 9.8 seconds for 9.2 seconds of sound. If we look at what this action actually does, it can be seen that it removes modes with a high-frequency. Using a linearly swept sine (20 – 20,000Hz) of 10 seconds as an input and comparing the dependency-removed output with the original output proves the aforementioned statement (see Figure 6.4).

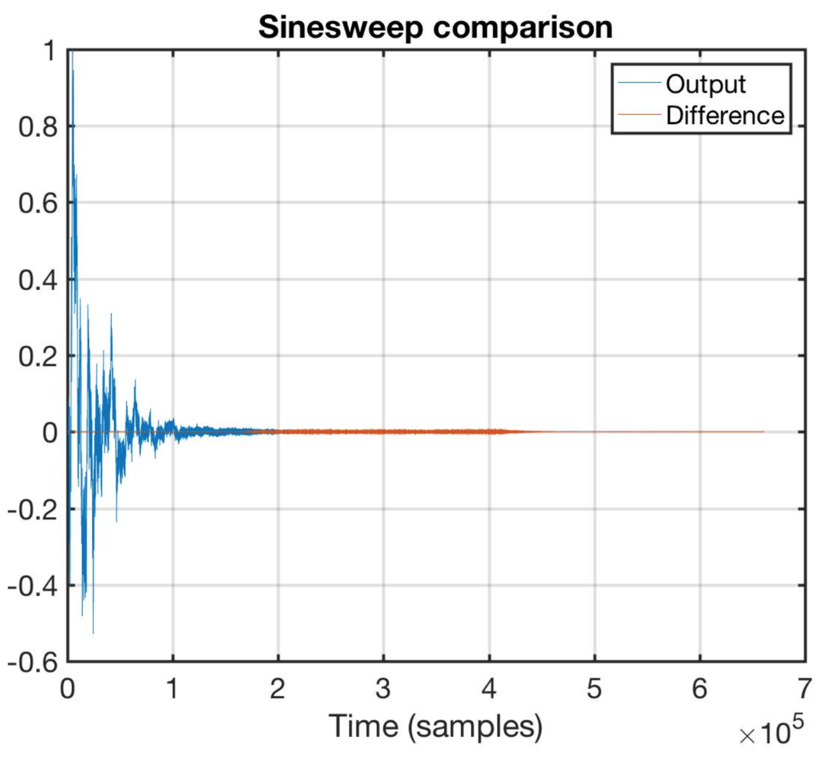


Figure 6.4: The original output and its difference with the dependency-removed output. The input is a linearly swept sine (20 – 20,000 Hz) with a length of 10 seconds.

6.2.3 Cents

What the authors explain in [17] is the possibility of removing eigenfrequencies that are close together and thus not perceptually important. Here, the following is proposed:

$$d = \left(\sqrt[12]{2}^{C/100} - 1 \right) \cdot f_c, \tag{6.2}$$

where C is an arbitrary amount in cents and f_c is the current eigenfrequency (in Hz) starting with the lowest eigenfrequency accounted for in the algorithm, which can be calculated using $f = \frac{\omega}{2\pi}$. The algorithm will then discard the eigenfrequencies if:

$$f_i - f_c < d,$$

otherwise:

$$f_c = f_i.$$

When C is put to 0.1 cents, the total number of eigenfrequencies accounted for (using the EMT140 properties) decreases from 18,218 to 7,932 (see Figure 6.5), which already makes a big difference in computational time: roughly 8.3 seconds to compute 9.2 seconds of sound.

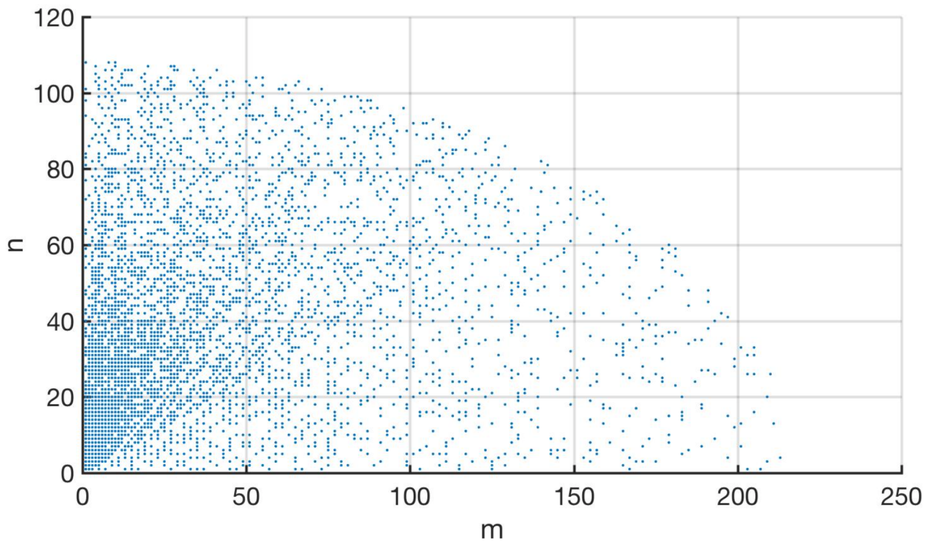


Figure 6.5: Visualisation of modes accounted for where $C = 1$ cent.

Below, the implementation of the above can be seen. Note, that the `omega` matrix first needs to be sorted according to the eigenfrequencies in order for it to work. At the end, it will be sorted back according to its modes.

```
%% Calculate Cents
if calcCent == true
    disp('Calculate Cents')
    n = 1;
    C = 0.1;
    omegaPrev = 0; %set omega_i
    omega = sortrows(omega,1); %sort according to eigenfrequencies
    ncent = ...
        nthroot(2,12)^(C/100)*(omega(1,1)/(2*pi))-omega(1,1)/(2*pi); ...
        %set d in Hz
    while n < length(omega(:,1))
        %if f_c-f_i < d, discard eigenfrequency
        if omega(n,1)/(2*pi) - omegaPrev/(2*pi) < ncent
            omega(n,:) = [];
        else
            omegaPrev = omega(n,1); %otherwise set next f_i
            n = n+1;
            % and calculate next d
            ncent = nthroot(2,12)^(C/100)*(omega(n,1)/(2*pi))...
                -omega(n,1)/(2*pi);
            end
        end
        omega = sortrows(omega,[2,3]); %sort according to modes
    end
```

If all of the above is implemented and the initial values of the sliders are set to: *Dry/Wet*: 100%, *Plate Width*: 2 m and *Height*: 1 m, putting *Cents* to 1 will give no audible artefacts using a buffer size of 512 samples. In total, 3,018 eigenfrequencies will be accounted for in the algorithm (see Figure 6.6).

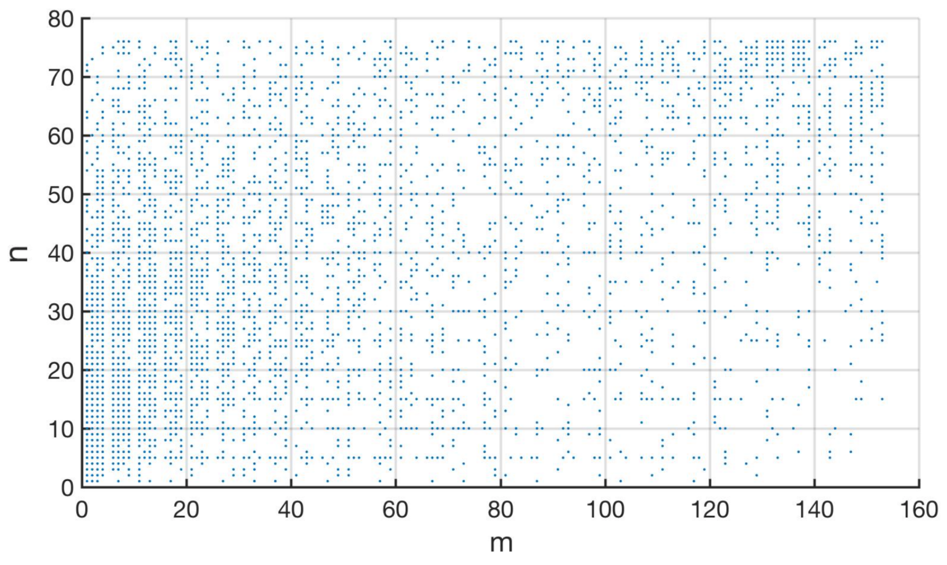


Figure 6.6: Visualisation of modes accounted for with all mode-reduction techniques applied ($C = 1$ cent).

6.3 Dynamic Outputs

As can be seen in the previous chapter, in order to improve the computational speed when the outputs are moving, the modal shape $\Phi(x_{\text{out}}, y_{\text{out}})$ is evaluated for different values of t: $t \in [1, f_s]$ with steps of $2/max([L_x\, L_y])$, before the main loop and is then selected (instead of evaluated) at the appropriate times in the update equation. This step size was found to be a good balance between speed and having minimal artefacts in the sound if the speed is not set to be too high.

6.4 Changing Plate Dimensions

In the algorithm, the eigenfrequencies and *q-vectors* would be recalculated according to the plate dimensions (see Section 5.7). Unfortunately, to do this continuously is too computationally expensive and will cause artefacts in the sound. Therefore, it has been decided to keep the same number of eigenfrequencies until reinitialisation of the plugin. The values of the individual eigenfrequencies, however, will be updated to maintain the stretching effect.

6.4.1 Smooth Stretching

If the dimensions of the plate are changed fast, it sounds like the plate is stretched 'stepwise' which creates an undesired sound. In order to resolve this, an addition has been made to the algorithm to smooth out the slider changes. For both the x and y-dimension if:

$$[D_\text{s} \cdot \phi]/\phi > [D_\text{c} \cdot \phi]/\phi,$$

then:

$$D_\text{s} = D_\text{s} - 1/\phi,$$

and if:

$$[D_\text{s} \cdot \phi]/\phi < [D_\text{c} \cdot \phi]/\phi,$$

then:

$$D_\text{s} = D_\text{s} + 1/\phi,$$

where D_s is the smoothed out dimension that will be used in the algorithm, D_c is the current dimension, i.e., the current slider position and ϕ is a variable dependent on the program speed. The 'round'-function is used to convert the analog values from the slider to values with a specified step size.

If the dimensions are changed slowly, the above method has the opposite effect. This is why the smoothing only occurs when the changes are big, i.e., bigger than $1/\phi$ meters. As said before, there is an extremely high correlation between the total number of eigenfrequencies and the program speed. It has thus been chosen to make ϕ linearly dependent to the number of eigenfrequencies. It has been found that $\phi = \frac{E_\text{tot}}{20}$ (where E_tot is the total number of eigenfrequencies accounted for) is a nice tradeoff between speed of D_s reaching D_c quickly when the slider is changed fast and eliminating the 'stepwise stretching' sound. See the code below for the implementation.

```
M = plugin.lengthOmega;
sS = round(M/20); %set speed dependent variable
%check for change larger than threshold
if abs(plugin.Lx - plugin.Lxpre) > 1/sS
```

```
    plugin.smoothLx = true;
else
    plugin.smoothLx = false;
end
if abs(plugin.Ly - plugin.Lypre) > 1/sS
    plugin.smoothLy = true;
else
    plugin.smoothLy = false;
end

%smooth stretching
if plugin.smoothLx == true
    if round(LxSmoothUse*sS)/sS > round(plugin.Lx*sS)/sS
        LxSmoothUse = LxSmoothUse - 1/sS;
    else
        if round(LxSmoothUse*sS)/sS < round(plugin.Lx*sS)/sS
            LxSmoothUse = LxSmoothUse + 1/sS;
        end
    end
else
    LxSmoothUse = plugin.Lx;
end
if plugin.smoothLy == true
    if round(LySmoothUse*sS)/sS > round(plugin.Ly*sS)/sS
        LySmoothUse = LySmoothUse - 1/sS;
    else
        if round(LySmoothUse*sS)/sS < round(plugin.Ly*sS)/sS
            LySmoothUse = LySmoothUse + 1/sS;
        end
    end
else
    LySmoothUse = plugin.Ly;
end

%set previous value the value used in the plugin
plugin.Lxpre = LxSmoothUse;
plugin.Lypre = LySmoothUse;
```

Chapter 7

Evaluation

This chapter describes three different types of evaluations that have been done over the course of this project. First of all, to find out how much the number of modes could be reduced (to increase computational speed) before a difference would be heard in the output sound, a MUSHRA test was carried out. Secondly, a questionnaire was made (and filled out by the same participants of the MUSHRA test) especially to evaluate whether the output sound was enjoyed. Lastly, the plugin was informally evaluated by some music producers to find out if they would use the implementation for their own projects. All three evaluations and their results are described below after which they are discussed and concluded upon.

7.1 MUSHRA test

MUSHRA stands for MUltiple Stimuli with Hidden Reference and Anchor and is used to test subjective audio quality (also see [28]). The number of modes that the algorithm accounts for was reduced according to eq. (6.2). In this formula, C was set to 0.1, 0.5, 1, 2, 5, 10 and 20 cents, resulting in an output that became of less and less quality. These values have been chosen based on either: 1) being close to the predicted value where subjects would perceive a difference between reference and mode reduced reference (i.e., the value that was looked for) or 2) being so much reduced in quality that it would act as an extra anchor point for the participants not to weigh small impairments too heavily. Next to this, the reference output (no mode reduction) was low-passed at 3.5kHz and used in the test as an anchor point. The audio files used were: 1) a short melody played by with a digital version of the Wurlitzer Classic instrument found in *Garageband* [19], 2) a recording of a ukulele playing a part the intro of Led Zeppelins *Stairway to Heaven*, 3) a part of Queens *We are the Champions* where vocals were isolated. These sounds were chosen as they belong to three different groups of sounds (digital keyboard, acoustic string instrument, vocals) that the plugin could be used with. The lengths of the audio files were 9.2, 12 and 15 seconds respectively. The audio files were processed in the algorithm and presented to the subject as a 100% wet output signal. The test was carried out using the software found in [29] (see Figure 7.1 for the GUI layout). For the test, a MacBook Pro with a 2.2GHz Intel Core i7 processor and a set of Beyerdynamic DT770 M headphones were used. The participants were allowed to change the volume before and during the test.

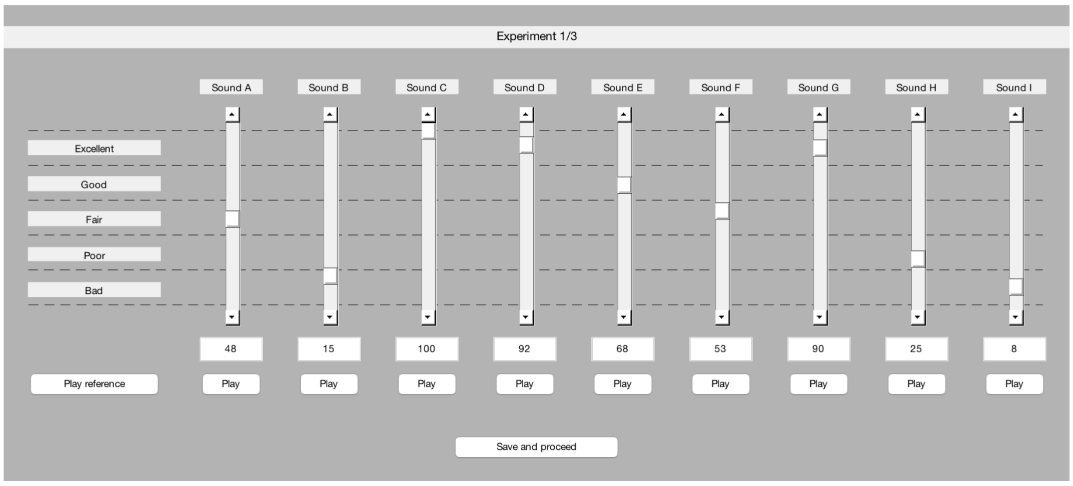


Figure 7.1: MUSHRA test GUI.

In total, nine people participated in the test, of which eight were considered either musicians (at least 2–3 years of experience) or experienced listeners (4 or more hours listening to music a day). The participants were between 19–32 years of age. The average rating per instrument per condition can be found in Figure 7.2. The anchor (the low-passed reference at 3.5kHz) was rated on average 95, 47 and 94 for the keyboard, ukulele and vocals respectively. In Figure 7.3 the means and 95% confidence intervals of the difference in rating between the reference and the mode reduced condition can be seen. For the detailed results, see Appendix C.1.

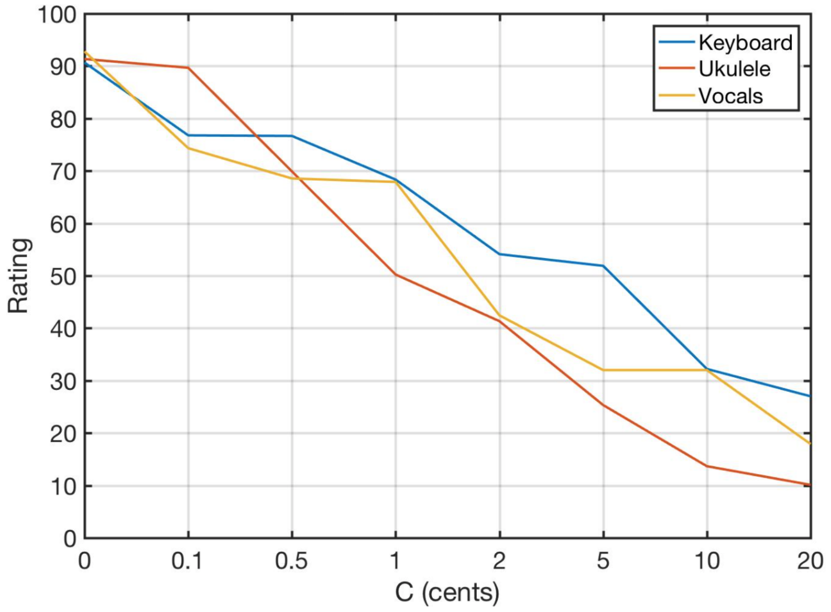


Figure 7.2: Average rating for the different sound files per mode reduced condition.

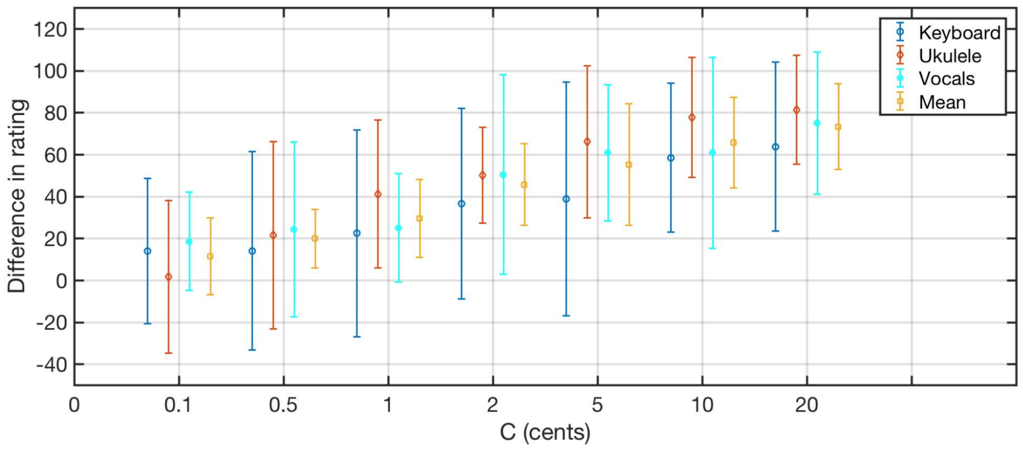


Figure 7.3: Difference in rating between reference and mode reduced condition (means and 95% confidence intervals).

7.2 Questionnaire

The same subjects participating in the MUSHRA test needed to fill out a questionnaire. The most important part of the questionnaire was to find out if the participants enjoyed the reverb effect when applied to the different audio files. A video would present both the unprocessed input sound and the processed (reference) sound (see [30]). Thereafter, the participants would answer the question: "Did you enjoy the sound of the reverberation effect in the following sounds: 1) Keyboard, 2) Ukulele, 3) Vocals" on a scale of 1-5 (no, not at all - no, not really - I'm indifferent - Yes, quite - Yes, very much). After this, the following was asked: "What is your general opinion on the reverberation effect? If you're a musician/producer, would you use it? For what kind of music/sounds? Any additional comments?" to retrieve extra information on the opinion of the participants.

For the keyboard the participants rated the reverberated output an average of 4.33, for the ukulele 4.56, and for the vocals 2.89 on the aforementioned scale (also see Figure 7.4. Most musicians and producers that participated in the test, would use the plate reverberation effect for their instruments/as a plugin. For the detailed results, see Appendix C.2.

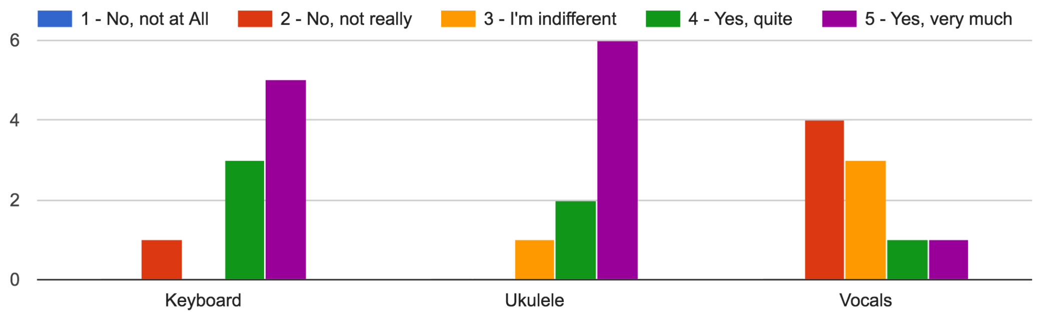


Figure 7.4: Histogram of ratings for the three different reverberated sounds.

7.3 Plugin

To evaluate the plugin in use, it was sent to four music producers personally known by the author. They were then asked to give their informal opinion on the sound of the plugin. The following instructions were given: "*Cents* is set to 10 at initialisation of the plugin. Changing this to a lower value will increase the quality of the reverb, but increases the chance of audible artefacts. To apply your change in *Cents* turn *Re-Initialise* off and on again." The lowest *Cents* value the producers could use before experiencing artefacts differed a lot between them. One could only set its value to 4 while another could set it to 1 (see Figure 7.5 for his detailed analysis).

The plugin was generally enjoyed by the people it was sent to, especially when applied to vocals, guitars and kicks. One producer said the output sound contained a lot of low and mid-frequency content and missed the 'highs'. He also said the reverb contained a lot of resonance, making it very 'dirty' and unique.

Changing the *Plate Width/Height* parameters have been considered a very interesting feature of the plugin.

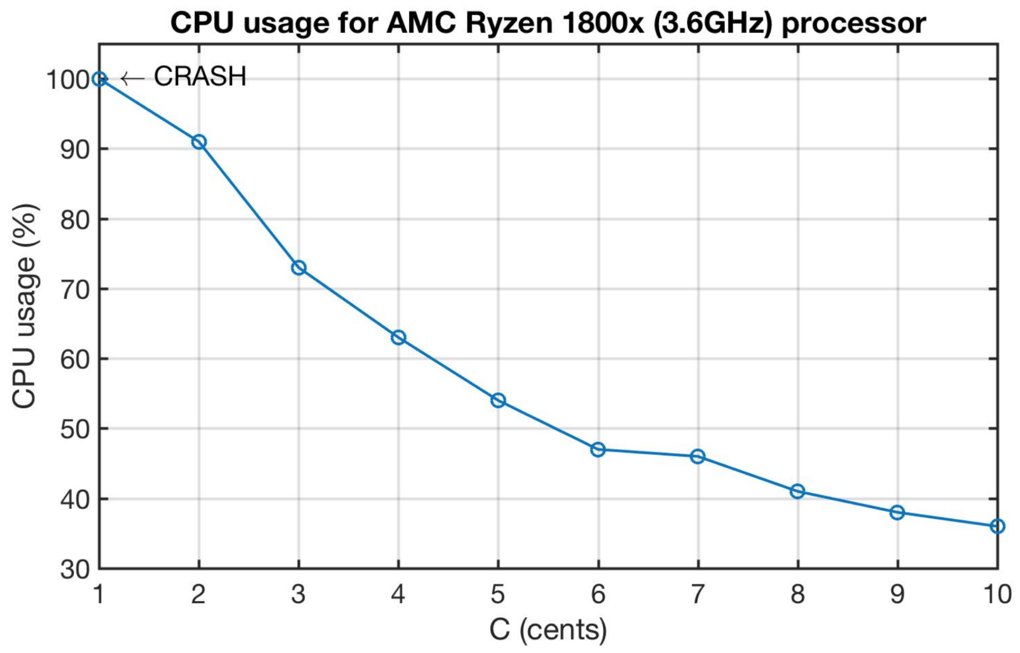


Figure 7.5: CPU load of one of the music producers.

7.4 Discussion and Conclusion

MUSHRA test

First of all, in Figure 7.2, it can be seen that (on average) the more the modes were reduced, the lower the participants would rate the quality of the audio file (as expected). An interesting thing that can be seen is that the ukulele got a lower rating more quickly than the other sounds as the modes decreased. This can be explained by the fact that this sound has more high-frequency content and the mode reduction algorithm has more influence on higher frequencies than lower. The rating for the low-passed anchor sound also implies that the ukulele sound is more affected by high-frequency reduction.

In Figure 7.3 it can be seen that the results of the MUSHRA test differ a lot between different instruments. For the ukulele it only takes 1 cent to be distinguishable from the reference 95% of the time, whereas for the vocals it takes 2 cents and for the keyboard even 5. The mean of the three instruments, however, implies that the difference between reference and mode-reduced reference can not be heard (with 95% certainty) when *Cents* is set to 0.1 but can be heard when *Cents* is set to 0.5. Because of this difference between the mean and the individual audio files it has been concluded that more subjects are needed to be able to say more about the data.

Questionnaire

The questionnaire proves that the output of the algorithm is generally enjoyed. As can be seen from the ratings, participants enjoyed the reverb more when applied to instruments rather than vocals. This might be caused by the wetness of the signal. When applied to vocals a too wet signal might be too "rich" or make it sound like someone is "singing in the shower" (see Appendix C.2).

Plugin

The difference in the lowest possible *Cents* value (before artefacts are heard) between the music producers is most probably due to the different processors they use. The participant who experienced artefacts at 1 cent, used an AMD Ryzen 1800x (3.6GHz) processor whereas the participant who could not go below 4 cents used a four-year-old Intel Core i5-4670 3.4GHz processor. In the end however, the artefacts only appear during real-time playback in the DAW. After exporting a sound file the artefacts will generally be gone [31].

The observation of the producer saying that the output sound missed high-frequency content, can most probably be explained by the fact that the mode reduction especially influences this part of the spectrum. He set the *Cents* parameter between 2–4 cents removing a lot of high-frequency modes, explaining his observation.

Some music producers stated that they would not use the plugin 100% wet, but only, for instance, 25–30%. This means that a low quality as a result of mode reduction would be masked by the dry signal in that case, i.e, the reverb would blend into the sound, rather than be a stand-alone sound, allowing for a higher mode reduction.

A point of general feedback was that the *Dry/Wet*-parameter was found to be "a bit too sluggish", which can be solved by having a separate gain for both in- and output sound. Next to that, one of the producers would have liked to see a decay parameter, which – according to the author – would not be hard to implement.

Chapter 8

Discussion

In this chapter, the (audible) results of the algorithm will be discussed. They have been informally evaluated by the author. Unfortunately, during the course of this project, there was no access to an actual plate reverb, so to in order to tell whether the algorithm was successful it was compared to the PA1 Dynamic Plate Reverb and ValhallaPlate: already existing VA plugins. When compared to these plugins, it was attempted to put their settings (plate size, decay values, etc.) as close to the values of the algorithm as possible. The outputs from the plugin and the algorithm were then compared. The reasons for any differences in sound output could only be speculated on, as the plugins were not open-source and internal parameters were thus hidden.

As there was no way to compare the extensions (moving the outputs and changing plate dimensions) to existing solutions, the output could only be evaluated based on what it was expected to sound like. The spectrograms of the input and a few output examples can be found in Figure 8.1.

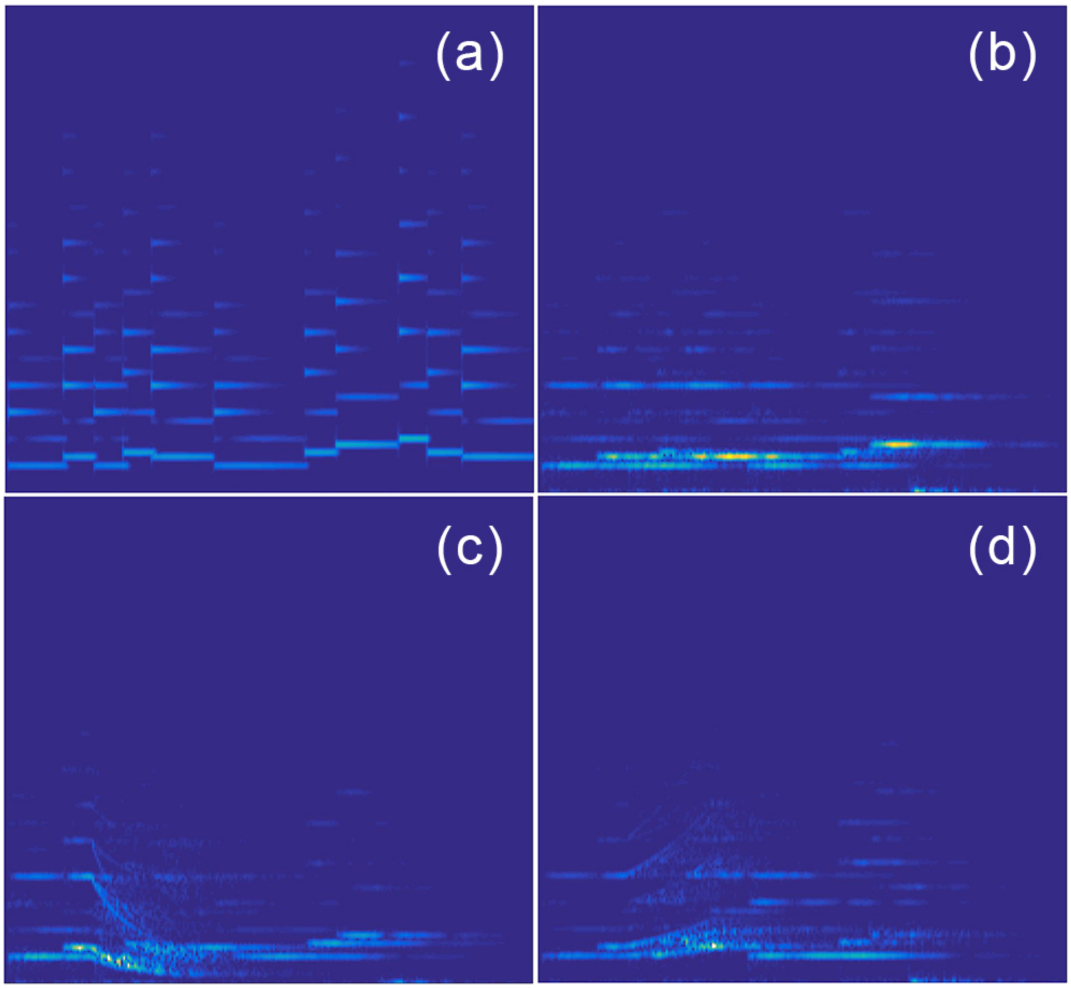


Figure 8.1: Specrtograms of input and different output examples (all normalised) using EMT140 properties. x-axis: Time (0–6 seconds), y-axis: Frequency (0–3000 Hz). (a): Dry input signal, (b): Left output signal $(0.1L_x, 0.45L_y)$, (c): Plate stretched from L_x = 2–4 m between 1–2 seconds., (d): Plate shrunk from L_x = 2–1 m between 1–2 seconds.

8.1 General Output

In general, the output sound is very natural, especially when combined with some dry input signal. Although no formal listening tests (focusing on this) have been carried out, the naturalness of the sound has been reported by the author. When compared to the PA1 Dynamic Plate Reverb and ValhallaPlate the output has more low-frequency content. This is probably because the damping factors that were included in the algorithm mostly attenuate the high-frequency content. As can be seen in Chapter 6, the option of turning off the physical damping factors was added. This creates a 'fuller' sound with more high frequency content. (For more on this, see Section 8.4)

8.2 Dynamic Outputs

Moving the outputs creates a result that sounds like a vibrato/flanging effect. This can be explained by the fact that the outputs 'travel away' or 'travel towards' sound that has already been travelling in the plate which changes the pitch of the output sound slightly. This could be seen analogous to the Doppler effect.

Letting the left and right output move at different speeds or in different patterns causes the sound to have arrive later/earlier in the left channel than the right channel at different time instances. Perceptually, it will sound like the reverb is moving from left to right at different speeds causing a very immersive stereo effect.

8.3 Changing Plate Dimensions

Increasing the size of the plate creates a pitch-bend effect: lower pitch when L_x and/or L_y are increased and higher when these are decreased. This can be explained by the fact that the eigenfrequencies are 'stretched/shortened' as the plate dimensions increase/decrease. The exact opposite happens when plate thickness is changed: if the plate gets thicker, the pitch goes up, and the other way around.

An interesting effect occurs when input and output are combined (not 100% wet signal) as the pitch bend effect only applies to the reverb, and does not influence the dry input signal.

8.4 Loss Coefficients

The current state of the art does not make use of loss coefficients dependent on physical parameters. The possibility to make some of these parameters dynamic was explored; the most interesting parameter being the air density ρ_a in eq. (2.6). When increasing this, the sound (especially the high frequency content) will die out sooner and has a 'muffled' sound; this can also be derived from the equation. Ultimately, it was decided against implementing this feature, as it did not add much to the sound; it only decreased the naturalness of the plate reverb.

Furthermore, in the real-time plugin, the option of turning the *Physical Damping* parameter off and on was included. It has been found that for a lot of different input sounds, the output gets more appealing when this damping is excluded from the algorithm. It generally gets more 'muffled' as there is a decrease in high-frequency content when including these damping mechanisms. The author assumes the creators of the The PA1 Dynamic Plate Reverb plugin also found this, as this plugin offers frequency dependent controls. It could also be that these creators saw the potential of VA simulations of offering different kinds of damping that are physically impossible, while maintaining the option of recreating the original sound. User-controlled damping in the low-frequency range is implemented in plate reverbs (damping induced by porous medium), but VA simulations allow to extend upon this by having more options regarding frequency spectrum. This is also the reason why the porous medium has been left out of the project. A parameter that controls the loss coefficients just like a porous medium would do in the real world could be implemented in future work.

Chapter 9

Conclusion and Future Work

In this report, a VA simulation of plate reverberation was presented. Different implementation techniques have been investigated of which one was chosen to be the most suitable for VA simulation of plate reverberation. Possible extensions to already existing models have then been explored and created. The aforementioned has all been implemented and transformed into a real-time plugin of which the output is found to be natural and enjoyable. Parameters like in- and output positions and plate dimensions have been made dynamic resulting in a very unique and interesting vibrato and pitch-bend effect (respectively), something which has not yet been achieved by the current state of the art. The above means that all of the project goals stated in Chapter 1 have been achieved.

In the future, I would like improve on the plugin, both in usability and functionality. First of all, parameters related to the LFO stretching of the plate (such as speed and amount) and the movement of the pickups (different patterns, speed, etc.) could be expanded upon and made available to the user. Next to that, I would like to explore the possibilities of improving computational speed in different ways than stated in this report. It could be investigated if the algorithm could be parallelised in some way. Alternatively, a library containing information about the plate at different settings could be created replacing the need to calculate some values every loop. I would then like to test the plugin (again) with musicians in order to test usability and whether the output sound is satisfactory. Furthermore, as stated in Chapter 8 it would be interesting to add the damping induced by a porous medium to the implementation. Lastly, I would like to explore the possibilities of changing other parameters of the plate using the presented algorithm as a basis. The shape and the structure of the plate, for example, would be very interesting to make dynamic. Moreover, different materials, such as gold and aluminium, or even non-metallic materials like glass, could be explored and result in some interesting timbres.

Bibliography

[1] S. Willemsen. "Virtual Analog Simulation and Extensions of Plate Reverberation". In: *Proceedings of the 14th Sound and Music Computing Conference* (2017).

[2] V. Välimäki. "Introduction to the Special Issue on Virtual Analog Audio Effects and Musical Instruments". In: *IEEE Transactions on Audio, Speech, and Language Processing* Vol. 18 (2010), pp. 713–714. URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5446591`.

[3] G. Grimes. *History of Guitar Effects*. 2011. URL: `http://electronics.howstuffworks.com/gadgets/audio-music/guitar-pedal1.htm`.

[4] Universal Audio. *EMT140 Plate Reverb Powered Plug-In for UAD-2 (video)*. 2010. URL: `https://www.youtube.com/watch?v=_KnMG_OW5c0`.

[5] Soundonsound. *Plate Reverb image*. 2007. URL: `http://www.soundonsound.com/techniques/mechanical-sfx`.

[6] J. Menhorn. *EMT140 Plate Reverb*. 2012. URL: `http://designingsound.org/2012/12/emt-140-plate-reverb/`.

[7] K. Arcas. "Physical Modelling and Measurements of Plate Reverberation". In: *19th International Congress on Acoustics Madrid* (2007). URL: `https://www.researchgate.net/publication/242117132`.

[8] Department of Aerospace Engineering Sciences. *Advanced Finite Element Methods (Course material)*. 2017. URL: `http://www.colorado.edu/engineering/CAS/courses.d/AFEM.d/AFEM.Ch20.d/AFEM.Ch20.pdf`.

[9] K. Arcas and A. Chainge. "On the quality of Plate Reverberation". In: *Applied Acoustics 71* (2010), pp. 147–156. URL: `http://www.sciencedirect.com/science/article/pii/S0003682X09001716`.

[10] J. S. Abel, D. P. Berners, and A. Greenblatt. "An Emulation of the EMT140 Plate Reverberator Using a Hybrid Reverberator Structure". In: *127th AES Convention* (2010).

[11] N. H. Fletcher and T. D. Rossing. *The Physics of Musical Instruments*. Second Edition. Springer, 1998.

[12] W. G. Gardner. "Efficient Convolution without Input-Output Delay". In: *Journal of the Audio Engineering Society* Vol. 43 (1995), pp. 127–136. URL: www.cs.ust.hk/mjg_lib/bibs/DPSu/DPSu.Files/Ga95.PDF.

[13] J.-M. Jot and A. Chaigne. "Digital delay networks for designing artificial reverberators". In: *AES 90th Convention* (1991). URL: http://www.bibsonomy.org/bibtex/2b9702a79fa8020f265101f5804a2d4f7/bovansnow.

[14] J.-M. Jot. "Efficient Models for Reverberation and Distance Rendering in Computer Music and Virtual Audio Reality". In: *Proc. 1997 Int. Computer Music Conf* (1997). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.8176&rep=rep1&type=pdf.

[15] V. Välimäki et al. *DAFX: Digital Audio Effects*. Second Edition. John Wiley & Sons Ltd., 2011, pp. 473–522. URL: http://onlinelibrary.wiley.com/doi/10.1002/9781119991298.ch12/summary.

[16] S. Bilbao and M. van Walstijn. "A finite difference scheme for plate synthesis". In: *Proceedings of the International Computer Music Conference* (2005). URL: http://www.research.ed.ac.uk/portal/en/publications/a-finite-difference-scheme-for-plate-synthesis(e212ac20-a87a-4137-99a5-f3ca6b47512e).html.

[17] C. J. Webb and M. Ducceschi. "Plate reverberation: Towards the development of a real-time physical model for the working musician". In: *Proceedings of the 22nd International Congress on Acoustics* (2016). URL: http://mdphys.org/PDF/icaPLA_2016.pdf.

[18] S. J. Orfanidis. *Introduction to Signal Processing*. Pearson Education, 2010, pp. 214–264. URL: http://www.ece.rutgers.edu/~orfanidi/intro2sp/orfanidis-i2sp.pdf.

[19] Apple Inc. *Garageband for Mac*. 2016. URL: http://www.apple.com/mac/garageband/.

[20] Waves Audio. *Abbey Road Reverb Plates*. 2016. URL: http://www.waves.com/plugins/abbey-road-reverb-plates.

[21] U-Audio. *U-Audio EMT140 classic plate reverberator*. 2010. URL: http://www.uaudio.com/store/reverbs/emt-140.html.

[22] M. Walker, D. Williamson, and P. White. *"Plug-in Folder" (Sound on Sound article)*. 2005. URL: http://www.soundonsound.com/node/4914494.

[23] C. Webb. *PA1 Dynamic Plate Reverb*. 2005. URL: http://www.physicalaudio.co.uk/PA1.html.

[24] ValhallaDSP. *ValhallaPlate*. 2015. URL: http://valhalladsp.com/shop/reverb/valhalla-plate/.

[25] IK Multimedia Production. *CSR Plate Reverb*. 2011. URL: http://www.ikmultimedia.com/products/trcsrplate/.

[26] G. Xie, D.J. Thompson, and C.J.C. Jones. "Mode count and modal density of structural systems: relationships with boundary conditions". In: *Journal of Sound and Vibration 274* (2004), pp. 621–651. URL: http://www.sciencedirect.com/science/article/pii/S0022460X03009507.

[27] D Reinfurt. *Lissajous Patterns*. 2014. URL: http://1000things.org/media/image/large/1lissajous.gif.

[28] The ITU Radiocommunication Assembly. *Method for the subjective assessment of intermediate quality level of coding systems*. 2003. URL: https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.1534-1-200301-S!!PDF-E.pdf.

[29] E. Vincent. *A Matlab interface for MUSHRA listening tests (version 1.0)*. 2005. URL: http://c4dm.eecs.qmul.ac.uk/downloads/#mushram.

[30] S. Willemsen. *MUSHRA In- and Outputs (video)*. 2017. URL: https://www.youtube.com/watch?v=3ZWIYljyQV.

[31] Image-Line. *Buffer underruns and maximizing FL Studio performance*. 2017. URL: https://www.image-line.com/support/FLHelp/html/app_underrun.htm.

Appendix A

Paper written for the 14th SMC conference

VIRTUAL ANALOG SIMULATION AND EXTENSIONS OF PLATE REVERBERATION

Silvin Willemsen
Aalborg University
swille15@student.aau.dk

Stefania Serafin
Dept. AD:MT
Aalborg University
sts@create.aau.dk

Jesper Rindom Jensen
Audio Analysis Lab, AD:MT
Aalborg University
jrj@create.aau.dk

ABSTRACT

In the 50s and 60s, steel plates were popularly used as a technique to add reverberation to sound. As a plate reverb itself is quite bulky and requires lots of maintenance, a digital implementation would be desirable. Currently, the available (digital) plugins rely solely on recorded impulse responses or simple delay networks. Virtual Analog (VA) simulations, on the other hand, rely on a model of the analog effect they are simulating, resulting in a sound and 'feel' the of the classical analog effect. In this paper, a VA simulation of plate reverberation is presented. Not only does this approach result in a very natural sounding reverb, it also poses many interesting opportunities that go beyond what is physically possible. Existing VA solutions, however, have limited control over dynamics of physical parameters. In this paper, we present a model where parameters like the positions of the in- and outputs and the dimensions of the plate can be changed while sound goes through. This results is in a unique flanging and pitch bend effect, respectively, which has not yet be achieved by the current state of the art.

1. INTRODUCTION

A great number of digital audio effects is currently available to musicians and producers. Many sounds we could not even imagine a few years ago, we can now create using current DSP technology. However, despite this immense amount of options, there is still a great desire for the sound of the classical analog effects that made their first appearance in the late 40s and characterised music from the 50s and 60s onwards. Even though, generally, digital sound effects 'do the job', they do not have the 'feel' that the old analog effects had - something greatly desired by many musicians. Contrary to digital effect simulations, Virtual Analog (VA) simulations rely on a model of the analog effect they are simulating [1]. A great advantage of VA simulations over the original systems is that they do not age and thus do not require time consuming maintenance. Also, when digitalised they are easily accessible, mostly simpler to use and can be made much cheaper than their analog counterparts. Naturally, the sound of a used analog audio effect can have its charm, but if desired, this can be modelled into the simulation. Also, VA simulations make it possible for parameters like room size, material properties, etc., to be changed, which is physically impossible or very hard to do. This can result in unique sounds that can only be created using VA simulations.

A popular reverberation technique used in the 50s and 60s was plate reverberation. A plate reverb utilises a small speaker (actuator) attached to a big steel plate to make it vibrate, and several pickups to pick up the sound after it has propagated through the plate. Several different plate reverbs made it to the market, the most popular being the EMT140 used in the Abbey Road Studios and undoubtedly leaving a mark on music in the aforementioned years. In fact, it was the only reverb used on Pink Floyd's Dark Side of the Moon [2]. A big issue of using an actual plate reverb is the sheer size and weight of it. The plate is 2x1m big and weighs (together with the rest of the installation) roughly 270kg [3], hence, a digital implementation of it would be desirable.

There are different ways of digitally implementing plate reverberation. Convolution is an effective approach, but not a VA one as it is not based on physical parameters. This means that flexibility is limited. Feedback Delay Networks (FDNs), as proposed by Jean-Marc Jot in [4], are also used as an implementation approach for creating a digital plate reverb by Jonathan Abel in [5]. FDNs are an efficient way of realising a VA solution to the plate reverberation problem. However, in his implementation, Abel uses a hybrid structure consisting of a convolutional part and an FDN-based part, making it not fully VA. Finite difference schemes as proposed in [6] and [7] are flexible, VA, but very computationally heavy solutions. Lastly, the vibrations of the plate can be decomposed into a series of plate modes - a modal description of the plate. It is both a VA and not computationally heavy approach and gives a lot of freedom in dynamic parameter manipulation.

Currently there are some VA plate reverb plugins available such as the *PA1 Dynamic Plate Reverb* [8] (that uses the aforementioned modal description) and the *Valhalla-Plate* [9]. These plugins, however, do not use the full potential of VA simulations. Parameters like for example pickup positions and sheet-size can be made dynamic, i.e., changed while sound is going through the plate. This will result in some interesting sounds and effects that can not be achieved with a physical plate reverb. In this paper we propose a VA simulation of plate reverberation that utilises these dynamic parameters - something that the aforementioned plugins have not implemented. Furthermore, different kinds of damping that occur in physical plate reverbs are taken into account to make the simulation sound even more natural.

This work is structured as follows: in Section 2 the physics of a thin metal plate will be explained, in Section 3 a numerical solution will be described, in Section 4 the results of this solution will be discussed and lastly in Section 5 we will conclude and discuss future works for this project.

2. PHYSICS OF A THIN METAL PLATE

In order to simulate a plate reverb, a model of the physics of the plate is needed. The Kirchhoff-Love model mathematically describes stresses and deformations in thin plates subjected to external forces. The partial differential equation for an isotropic plate (including damping) is [10]:

$$\frac{\partial^2 u}{\partial t^2} = -\kappa^2 \nabla^4 u - c\frac{\partial u}{\partial t} + f(x_{\text{in}}, y_{\text{in}}, t). \tag{1}$$

Here, $u = u(x, y, t)$, a state variable that describes the transverse plate deflection and is defined for $x \in [0, L_x]$, $y \in [0, L_y]$ and $t \geq 0$, c is a loss parameter, ∇^4 is the biharmonic operator, $f(x_{\text{in}}, y_{\text{in}}, t)$ is the input signal at input source location $(x_{\text{in}}, y_{\text{in}})$ at time instance t and finally, κ^2 can be referred to as the stiffness parameter:

$$\kappa^2 = \frac{Eh^2}{12\rho(1-v^2)}, \tag{2}$$

where E, h, ρ, and v are the Young's modulus, plate thickness, plate density and Poisson's ratio, respectively.

2.1 Frequency dispersion

One feature that characterises the sound of a plate reverb is the fact that higher frequencies propagate faster through a metal plate than lower frequencies [10, 11]. This phenomenon is called frequency dispersion. The dispersion relation is defined as:

$$\omega^2 \simeq \gamma^4 \kappa^2, \tag{3}$$

where ω is the angular frequency and γ the wavenumber. The group and phase velocities are defined as:

$$c_{\text{ph}} = \sqrt{\kappa\omega} \qquad c_{\text{gr}} = 2\sqrt{\kappa\omega}. \tag{4}$$

Using the properties of the EMT140 (steel, 0.5mm thick) and a frequency range of 20-20,000kHz, group velocities can vary between ca. 20-620m/s. In regular (or room) reverberation all frequencies travel at the same speed making frequency dispersion one of the main differences between plate and room reverberation.

2.2 Damping

In plate reverbs, three different kinds of damping occur: thermoelastic damping, radiation damping and damping induced by a porous medium [10, 11]. All three will be shortly described in this section.

2.2.1 Thermoelastic damping

Thermoelastic damping occurs in materials with a high thermal conductivity. It is an internal damping mechanism that damps different frequencies at different strengths according to the following equation [11]:

$$\alpha_{\text{th}}(\omega) = \frac{\omega}{2}\eta(\omega) \approx \frac{\omega^2 R_1 C_1}{2(\omega^2 h^2 + C_1^2/h^2)} \tag{5}$$

where R_1 and C_1 are material dependent constants. In the case of the EMT140 plate reverb, these are $R_1 = 4.94 \cdot 10^{-3}$ and $C_1 = 2.98 \cdot 10^{-4}$ [10].

2.2.2 Radiation damping

Radiation damping happens when vibration is converted to acoustic energy. This happens according to the following equation [10, 11]:

$$\alpha_{\text{rad}} = \frac{1}{4\pi^2}\frac{c_{\text{a}}\rho_{\text{a}}}{\rho h}\frac{2(L_x + L_y)}{L_x L_y}\frac{c_{\text{a}}}{f_{\text{c}}}g(\psi), \tag{6}$$

$$g(\psi) = \frac{(1-\psi^2)\ln[(1+\psi)/(1-\psi)] + 2\psi}{(1-\psi^2)^{3/2}},$$

where ρ_{a} and c_{a} are the density of air and speed of sound in air respectively and $\psi = \sqrt{\frac{f}{f_{\text{c}}}}$. Here, f_{c} is the critical frequency and can be calculated using $f_{\text{c}} = \frac{c_{\text{a}}^2}{2\pi\kappa}$.

2.2.3 Damping induced by porous medium

Plate reverberators contain a porous plate positioned behind the metal plate. The distance between the two entities can be set creating a difference in low-frequency decay [10, 11]. The damping plate position changes this decay between roughly 0.6 - 5.5 seconds [5] and can be manipulated on the interface of a real plate reverb like the EMT140.

2.3 Boundary conditions

The states of the edges of a plate are referred to as the boundary conditions. In [12], the authors state that a plate can have three different boundary conditions: free, clamped or simply supported (hinged). For a rectangular plate such as the plate reverb this means that there are 27 different possible combinations of boundary conditions. Every combination will have a unique impulse response and will thus sound different. In this work, we limit ourselves to all sides being simply supported. This means that state variable $u = 0$ at $x = 0$, $x = L_x$, $y = 0$ and $y = L_y$.

3. NUMERICAL SOLUTION

The implementation approach we chose to use is the modal description (see Section 1). In this section a numerical solution of this will be presented.

The state u, as seen in the Kirchhoff-Love equation (1) can be modelled as being a summation of a number of different modes [13]:

$$u = \sum_{m=1}^{M}\sum_{n=1}^{N} q_{mn}\Phi_{mn}(x, y), \tag{7}$$

where q_{mn} is the unknown amplitude of mode (m, n) (m being the mode over the horizontal axis and n over the vertical axis of the plate) and Φ_{mn} is a modal shape defined over $x \in [0, L_x]$ and $y \in [0, L_y]$. In this model, $M \cdot N$ is the total number of modes accounted for. In theory, this is infinite. Note that – apart from the chosen time step $(1/f_s)$ – the computational speed depends on M and N.

For a rectangular plate with sides L_x and L_y, using simply supported boundary conditions, $\Phi_{mn}(x, y)$ can be calculated [13]:

$$\Phi_{mn}(x, y) = \frac{4}{L_x L_y} \sin \frac{m\pi x}{L_x} \sin \frac{n\pi y}{L_y}, (m, n) \in \mathbf{Z}^+, \tag{8}$$

and q_{mn} can be found using the following update equation [13]:

$$A q_{mn}^{t+1} = B q_{mn}^{t} + C q_{mn}^{t-1} + \frac{\Phi_{mn}(x_p, y_p)}{\rho h} P^t, \tag{9}$$

where constants A, B and C can be described in the following way:

$$A = \frac{1}{k^2} + \frac{c_{mn}}{\rho h k}, \qquad B = \frac{2}{k^2} - \omega_{mn}^2,$$

$$C = \frac{c_{mn}}{\rho h k} - \frac{1}{k^2}.$$

Moreover, P^t is the input signal at time instance t at specified input location (x_p, y_p), k is the chosen timestep $1/f_s$ and c_{mn} is a loss coefficient that can be set per eigenfrequency which gives a lot of control over the frequency content of the output sound.

3.1 Eigenfrequencies

The (angular) eigenfrequencies ω_{mn} can be calculated using the following equation [14]:

$$\omega_{mn} = \kappa \pi^2 \left(\frac{m^2}{L_x^2} + \frac{n^2}{L_y^2} \right). \tag{10}$$

According to [13] and experimental observation, stability of the update equation (9) can only be assured if and only if:

$$\omega_{mn} < 2 f_s. \tag{11}$$

This poses a limit on the total number of modes in (7) and a creates the dependency:

$$\omega_{M(n)n}, \omega_{mN(m)} < 2 f_s, \tag{12}$$

where $m \in [1, M(n)]$ and $n \in [1, N(m)]$. All the eigenfrequencies that satisfy this condition will be used in the algorithm.

3.2 Loss coefficients

In [13], the authors set loss coefficients per frequency band. In our implementation, the different damping mechanisms described in Section 2.2 have been implemented to ultimately create a loss coefficient for each individual eigenfrequency. If we set the porous medium to the furthest distance possible, the damping induced by this can be ignored. In Figure 1 the results can be seen.

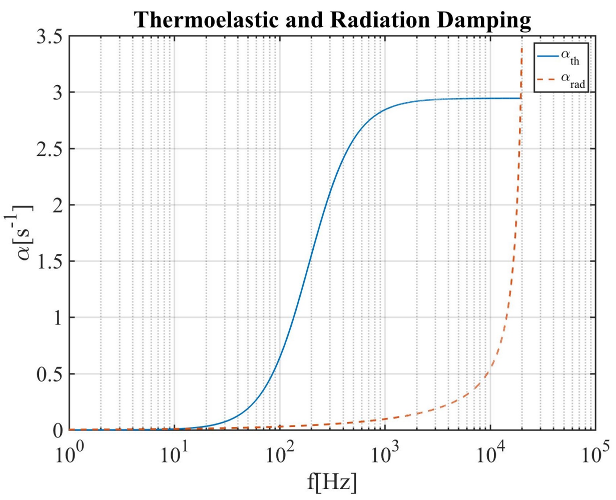


Figure 1. Thermoelastic (α_{th}) and Radiation damping (α_{rad}) for values based on the EMT140.

The loss coefficients are then calculated in the following way [11, 13]:

$$c_{mn} = \frac{12 \ln(10)}{T_{60}}, \quad \text{where} \quad T_{60} = \frac{3 \ln(10)}{\alpha_{\text{tot}}}, \tag{13}$$

$$\Rightarrow \quad c_{mn} = 4\alpha_{\text{tot}}.$$

Here, α_{tot} is simply the addition of all of damping factors.

3.3 Dynamic outputs

The output can be retrieved at a specified point by, in a certain sense, inverting (7):

$$u_{\text{out}} = \sum_{m=1}^{M} \sum_{n=1}^{N} q_{mn} \Phi_{mn}(x_{\text{out}}, y_{\text{out}}). \tag{14}$$

The *PA1 Dynamic Plate Reverb* (as described in Section 1) has the functionality of moving the input from left to right over the plate to create a flanging effect. In our implementation, not only both outputs (not inputs) are able to move, but they are able to move in elliptical and Lissajous patterns. This happens according to the following equations:

$$x_{\text{out}}(t) = R_x L_x \sin\left(S_x \cdot \frac{2\pi t}{f_s} \right) + 0.5, \tag{15}$$

$$y_{\text{out}}(t) = R_y L_y \sin\left(S_y \cdot \frac{2\pi t}{f_s} + \theta \right) + 0.5. \tag{16}$$

Here, $R_x, R_y \in [-0.5, 0.5]$ when multiplied with L_x and L_y respectively, determine the horizontal and vertical maxima of the output pattern. If one of these has a value of zero, the outputs will move in a linear shape. The shape of the pattern is determined by the horizontal and vertical speeds S_x, S_y and the phase shift θ. For example if

$S_x = S_y = 1$ and $\theta = 0.5\pi$ the outputs will follow an elliptical pattern. The patterns created for different values of S_x, S_y and θ can be seen in [15]. If either S_x or S_y has a value of zero, the outputs will again move in a linear fashion. Note, that it is perfectly possible to set different values for the aforementioned parameters for the left output channel and the right.

3.4 Changing plate dimensions

The fact that VA simulations are extremely flexible, poses a lot of interesting opportunities for model manipulation. We were interested in manipulating parameters that are physically 'fixed', such as the dimensions of a steel plate. Generally speaking, changing the dimensions of a steel plate is physically impossible to do. The challenge was thus to imagine how the sound would change if it would be possible. Looking at the variables that are dependent on the horizontal (L_x) and vertical (L_y) plate dimensions, we see that changing these would change the eigenfrequencies, the modal shapes and radiation damping. As can be seen in (10) the eigenfrequencies lower as L_x and L_y grow. Given the stability condition (11), the number of eigenfrequencies that can be accounted for ($M(n)$ and $N(m)$ in (7)) also grows as the plate dimensions grow. In the algorithm, this is implemented by adding zero values to q^{t+1}_{mn}, q^t_{mn} and q^{t-1}_{mn} (the q-vectors) in (9), where the combinations mn were non-existent in the previous update. If the plate decreases in size, the values of the q-vectors for which the combinations mn do not satisfy the stability condition anymore will be removed. Also in (7), the modal shapes change depending on L_x and L_y in (8). As long as the same number of modes is accounted for, the q-vectors will not be affected in (9) as Φ_{mn} changes. Lastly, the radiation damping α_{rad} and factors A, B and C in (9) will be updated as L_x and L_y change.

The thickness of the plate can also be changed. This changes the thermoelastic damping α_{th} in (5), the radiation damping α_{rad} in (6) and the stiffness factor κ^2 in (2) which then changes the eigenfrequencies again. It can be derived that a decrease in thickness will lower the eigenfrequencies.

4. RESULTS

In this section, the results of the implementation will be discussed. They have been informally evaluated by the authors and sound demos have been made available online [1]. Unfortunately, we did not have access to an actual plate reverb, so to in order to tell whether the implementation was successful we compared it to the *PA1 Dynamic Plate Reverb* and *ValhallaPlate*: already existing VA plugins. When compared to these plugins, we tried to put their settings (plate size, decay values, etc.) as close to the values of our implementation as possible. The outputs from the plugin and our implementation were then compared. The reasons for any differences in sound output we could only speculate on, as the plugins were not open-source and internal parameters were thus hidden.

[1] http://tinyurl.com/zwscbtl Full link: [16]

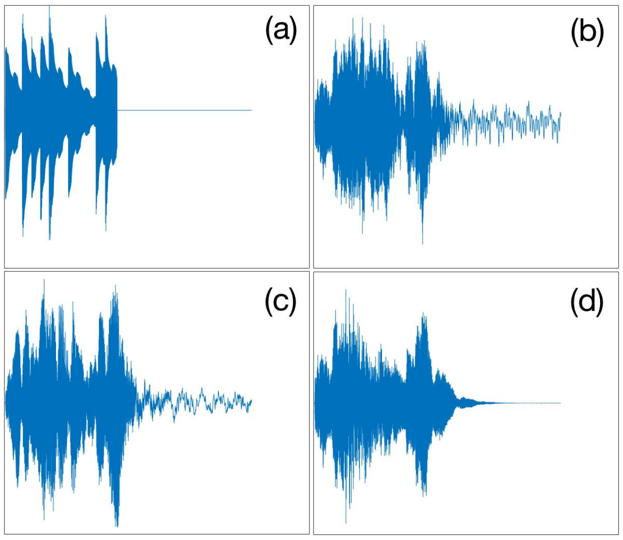


Figure 2. Plots of input and different output examples using EMT140 properties. Total length of outputs: 9.2 seconds. (a): Dry input signal, (b): Left output signal $(0.1L_x, 0.45L_y)$, (c): Left moving output signal ($R_x = R_y = 0.4, S_x = 6, S_y = 5, \theta = 0.5\pi$), (d): Plate stretched from $L_y = 1 - 2$m between $1 - 2$ seconds.

As we did not have any way of comparing our novel additions (moving the outputs and changing plate dimensions) to existing solutions, we could only evaluate the output based on what we expected it to sound like. The waveforms of the input and a few output examples can be found in Figure 2.

4.1 General output

In general, the output sound is very natural, especially when combined with some dry input signal. Although no formal listening tests have been carried out, the naturalness of the has been reported by the authors. When compared to the *PA1 Dynamic Plate Reverb* and *ValhallaPlate* the output has a little more low-frequency content. This is probably because the damping factors we included in our algorithm mostly attenuate the high-frequency content.

4.2 Moving outputs

Moving the outputs creates a result that sounds like a vibrato/flanging effect. This can be explained by the fact that the outputs 'travel away' or 'travel towards' sound that has already been travelling in the plate which changes the pitch of the output sound slightly.

Letting the left and right output move at different speeds or in different shapes causes the sound to arrive in the left channel and the right channel at different time instances. Perceptually, it will sound like the reverb is moving from left to right at different speeds causing a very immersive stereo effect.

4.3 Changing plate dimensions

Increasing the size of the plate creates a pitch-bend effect: lower pitch when L_x and/or L_y are increased and higher when these are decreased. This can be explained by the fact that the eigenfrequencies are 'stretched/shortened' as the plate dimensions increase/decrease. The exact opposite happens happens when plate thickness is changed: if the plate gets thicker, the pitch goes up, and the other way around. This can be explained by the increase in the stiffness factor in (2) when the thickness increases.

An interesting effect occurs when input and output are combined (not 100% wet signal) as the pitch bend effect only applies to the reverb, and does not influence the dry input signal.

4.4 Dynamic loss coefficients

To the best of our knowledge, the current state of the art does not make use of loss coefficients dependent on physical parameters. We explored the possibility to make some of these parameters dynamic. The most interesting parameter we explored is the air density (ρ_a) in Equation (6). When increasing this, the sound (especially the high frequency content) will die out sooner and has a 'muffled' sound and can also be derived from the equation. Ultimately, we decided against implementing this feature, as it did not add much to the sound; it only decreased the naturalness of the plate reverb.

4.5 Computational speed

As stated in before, the computational speed depends on the total number of eigenfrequencies ($M(n)$ and $N(m)$) being accounted for in the algorithm. We found a correlation of 0.996 between this number and program speed, making decreasing the total number of eigenfrequencies our main focus. What the authors explain in [13] is the possibility of removing the eigenfrequencies that are not perceptually important. In our algorithm we propose:

$$d = (\sqrt[12]{2}^{C/100} - 1) \cdot f_\mathrm{c}, \tag{17}$$

where C is an arbitrary amount in cents and f_c is the current eigenfrequency (in Hz) starting with the lowest eigenfrequency accounted for in the algorithm which can be calculated using $f = \frac{\omega}{2\pi}$. The algorithm will then discard the eigenfrequencies if:

$$f_i - f_\mathrm{c} < d,$$

otherwise:

$$f_\mathrm{c} = f_i.$$

When C is put to 0.1 cents, the total number of eigenfrequencies accounted for (using the EMT140 properties) decreases from 18,218 to 7,932, which makes a big difference in computational time. Now, our implementation only needs only needs roughly 7 seconds instead of 12 seconds to process 9.2 seconds of audio, proving that a real-time implementation would indeed be feasible. The algorithm has been tested using a MacBook Pro containing a 2,2GHz Intel Core i7 processor. The authors indeed report no significant audible difference after reducing the number of modes this way.

In order to improve computational time when the outputs are moving, $\Phi_{mn}(x_\mathrm{out}, y_\mathrm{out})$ is evaluated for different values of t: $t \in [1, f_\mathrm{s}]$ with steps of $4/max([L_x\, L_y])$, before the update equation (9) and is then selected (instead of evaluated) at the appropriate times in the update equation. We found this step-size to be a good balance between speed and having minimal artefacts in the sound if the speed is not set to be too high.

Changing the plate dimensions greatly influences the computational time as the eigenfrequencies need to be recalculated many time. To improve computational time, the eigenfrequencies and the modal shapes are reevaluated every 100th time-step, according to the current (new) dimensions of the plate. As with the previous chosen time-step, found this to be a good balance between speed and having minimal artefacts.

5. CONCLUSION AND FUTURE WORK

In this paper we presented a VA simulation of plate reverberation. The output of the implementation sounds natural and parameters like in- and output positions and plate dimensions have been made dynamic resulting in a very unique and interesting flanging and pitch-bend effect (respectively), something which has not yet been achieved by the current state of the art. Another novelty is that we included thermoelastic and radiation damping that influence every eigenfrequency independently.

In the future we would like to create a real-time plugin containing all that is presented in this paper. In order to do so, the algorithm needs to be optimised, especially computationally heavy processes like moving the pickups and changing the dimensions of the plate. When this is achieved, the plugin will be tested with musicians in order to test usability and whether the output sound is satisfactory. Furthermore, we would like to add damping induced by a porous medium to our implementation. Even though it has been ignored in this work, it is an important feature in the EMT140. Lastly, we would like to explore the possibilities of changing other parameters of the plate using the presented model as a basis. The shape and the structure of the plate, for example, would be very interesting to make dynamic. Moreover, different materials, such as gold and aluminium, or even non-metallic materials like glass, could be explored and result in some interesting timbres.

6. REFERENCES

[1] V. Välimäki, F. Fontana, J. O. Smith, and U. Zölzer, "Introduction to the special issue on virtual analog audio effects and musical instruments," in *IEEE Transactions on Audio, Speech, and Language Processing, VOL. 18, NO. 4*, New York, USA, 2010.

[2] U. Audio. (2010) EMT140 plate reverb powered plug-in for UAD-2 (video). [Online]. Available: https://www.youtube.com/watch?v=_KnMG_OW5c0

[3] J. Menhorn. (2012) EMT140 plate reverb. [Online]. Available: http://designingsound.org/2012/12/emt-140-plate-reverb/

[4] J. M. Jot and A. Chaigne, "Digital delay networks for designing artificial reverberators," in *Proc. 90th Convention Audio Eng. Soc, preprint 3030*, Paris, France, 1991.

[5] J. S. Abel, D. P. Berners, and A. Greenblatt, "An emulation of the EMT140 plate reverberator using a hybrid reverberator structure," in *127th AES Convention*, New York, USA, 2009.

[6] V. Välimäki, S. Bilbao, J. O. Smith, J. S. Abel, J. Pakarinen, and D. Berners, *DAFx: Digital Audio Effects Second Edition*. John Wiley and Sons Ltd., 2011.

[7] S. Bilbao and M. van Walstijn, "A finite difference scheme for plate synthesis," in *Proceedings of the International Computer Music Conference. pp. 119-122*, Barcelona, Spain, 2005, pp. 119–122.

[8] C. Webb. (2016) PA1 dynamic plate reverb. [Online]. Available: http://www.physicalaudio.co.uk/PA1.html

[9] ValhallaDSP. (2015) Valhallaplate. [Online]. Available: http://valhalladsp.com/shop/reverb/valhalla-plate/

[10] K. Arcas, "Physical modelling and measurements of plate reverberation," in *19th International Congress on Acoustics Madrid*, Madrid, Spain, 2007.

[11] K. Arcas and A. Chainge, "On the quality of plate reverberation," in *Applied Acoustics 71*, Palaiseau, France, 2010, pp. 147–156.

[12] N. H. Fletcher and T. D. Rossing, *The Physics of Musical Instruments Second Edition*. Springer, 1998.

[13] M. Ducceschi and C. J. Webb, "Plate reverberation: Towards the development of a real-time physical model for the working musician," in *Proc. of the 22nd International Congress on Acoustics*, Buenos Aires, Argentina, 2016.

[14] T. Irvine. (2013) Response of a rectangular plate to base exitation, revision e. [Online]. Available: http://www.vibrationdata.com/tutorials2/plate_base_excitation.pdf

[15] D. Reinfurt. (2014) Lissajous patterns. [Online]. Available: http://1000things.org/media/image/large/1lissajous.gif

[16] S. Willemsen. (2017) Sound files. [Online]. Available: https://www.dropbox.com/s/wy2ceyklx3wptu8/Sound%20Files%20VA%20Plate%20Reverb.zip?dl=0

Appendix B

MATLAB Code

B.1 Main Algorithm

```matlab
%% Reset Functions
%clear all;
clc
close all;

%% Set Global Variables
fs = 44100; %sample rate
ca = 343; %speed of sound in air
pa = 1.225; %air density

%% Set VA effects
phasing = false; %make pickups move or not
stretching = 0; %stretch or not (0 = false, 1 = true)
physDamp = false; %set physical damping off/on
decay = 4; %if physDamp = false, this decay value will be used for ...
    all frequencies

%% Set plate parameters
Lx = 2; %Plate width
Ly = 1; %Plate height
h = 0.0005; %plate thickness (m)

rho = 7850; %Material Density (kg/m^3)
E = 2e11; %Young's modulus
v = 0.3; % Poissons ratio
kSquared = (E*h^2)/(12*rho*(1-v^2)); %stiffness factor

%% Set input/output positions
p = [0.4 0.415]; %input position between (0-1)
qL = [0.1 0.45]; %left output position at (0-1)
```

```
qR = [0.84 0.45]; %right output position at (0-1)

%transform to fit plate dimensions
in = [p(1)*Lx p(2)*Ly];
outL = [qL(1)*Lx qL(2)*Ly];
outR = [qR(1)*Lx qR(2)*Ly];

%% Get input
[sound, soundfs] = audioread('rhodes2.aif');
offset = 0; %set possible offset (in samples) for testing after stretch
fixed = true;
if fixed == true
    len = 2; %set fixed length for input (in seconds)
else
    len = floor(length(sound)/soundfs); %set length to original ...
        length of input
end
input = zeros(soundfs*len+soundfs*5,1); %create room for reverb (5 ...
    seconds)
input(1+offset:length(sound(1:soundfs*len,1))+offset) = ...
    sound(1:soundfs*len,1); %insert sound
input(1+offset:length(sound(1:soundfs*len,1))+offset) = ...
    ones(soundfs*len,1); %insert sound

%% Set mode reduction options
square = false; %remove dependency
delModes = false; %delete unactivated modes
calcCent = false; %calculate cents
C = 1; %set C

%% Create Eigenfrequencies
disp('Create Omega')
val = 0;
m = 1;
m1 = 1;
m2 = 1;
omega = zeros(100000,3); %set to zeros for program speed
while val < fs*2 %check for stability
    val = ((m1/Lx)^2 + (m2/Ly)^2)*sqrt(kSquared)*pi^2; %calculate ...
        eigenfrequency
    %m1 is fixed, increase m2 until above stability condition
    if val < fs*2 %double check for stability
        omega(m,1) = val; %first column: eigenfrequency
        omega(m,2) = m1; %second column: horizontal mode
        omega(m,3) = m2; %third column: vertical mode
        m2 = m2 + 1;
        m = m+1;
    else
        if m1 == 1
            stablem2 = m2-1; %set highest stable vertical mode
        end
        m2 = 1; %reset m2
        m1 = m1 + 1; %increment m1
        val = ((m1/Lx)^2 + (m2/Ly)^2)*sqrt(kSquared)*pi^2; %check ...
            if (m1,1) > 2fs
        if val > fs*2
            stablem1 = m1 - 1; %set highest stable horizontal mode
            break; %if (m1,1)> 2fs , break out of the loop
        end
    end
end
omega = omega(1:m-1,:); %get rid of zeros

%% Remove dependency
if square == true
    highMode = omega(:,2).*omega(:,3);
    indexFound = find(highMode==max(highMode(:))); %find highest ...
        possible MN
    i = 1;
    maxM1 = omega(indexFound(1),2);
    maxM2 = omega(indexFound(1),3);
    while i <= length(omega(:,1))
        % if an eigenfrequency has a mode-value higher than the ...
            maximum, discard
        if omega(i,2) > maxM1 || omega(i,3) > maxM2
            omega(i,:) = [];
        else
            i = i + 1;
        end
    end
end

%% Delete Neclegible Modes From Input
if delModes == true
    disp('Delete neclegible modes from input')
    i1 = 1;
    answ1 = p(1);
    %multiply x_p with integers until the answer becomes an integer
    while rem(answ1,1) ~= 0
        i1 = i1+1;
        answ1 = p(1)*i1;
    end
    %multiply y_p with integers until the answer becomes an integer
    i2 = 1;
    answ2 = p(2);
    while rem(answ2,1) ~= 0
        i2 = i2+1;
        answ2 = p(2)*i2;
    end
    %discard the eigenfrequencies accordingly
    n = 1;
    while n <= length(omega(:,1))
```

```
        if mod(omega(n,2), i1) == 0 || mod(omega(n,3), i2) == 0
            omega(n,:) = [];
        else
            n = n + 1;
        end
    end
end

%% Calculate Cents
if calcCent == true
    disp('Calculate Cents')
    n = 1;
    omegaPrev = 0; %set omega_i
    omega = sortrows(omega,1);
    ncentSave = [];
    ncent = ...
        nthroot(2,12)^(C/100)*(omega(1,1)/(2*pi))-omega(1,1)/(2*pi); ...
        %set d in Hz
    while n < length(omega(:,1))
        %if f_c-f_i < d, discard eigenfrequency
        if omega(n,1)/(2*pi) - omegaPrev/(2*pi) < ncent
            omega(n,:) = [];
        else
            omegaPrev = omega(n,1); %otherwise set next f_i
            n = n+1;
            % and calculate next d
            ncent = nthroot(2,12)^(C/100)*...
                (omega(n,1)/(2*pi))-omega(n,1)/(2*pi);
        end
    end
    omega = sortrows(omega,[2,3]); %sort according to modes
end

%% Create PhiIn
M = length(omega(:,1));
phiIn = zeros(M,1);
for m = 1:M
    phiIn(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*in(1))/Lx)*...
        sin((omega(m,3)*pi*in(2))/Ly);
end

%% Set up Moving Outputs
disp('Set up Moving Outputs')
outputPointsX = 0:1/(10000*Lx):1;
outputPointsY = 0:1/(10000*Ly):1;
outputPointsX = outputPointsX*Lx;
outputPointsY = outputPointsY*Ly;

%set shape extremes
Rx = 0.4;
Ry = 0.4;
```

```

%set x and y speeds
Sx = 4;
Sy = 3;

%Create possible output positions
circX = outputPointsX(ceil(length(outputPointsX)*...
    (Rx*sin(Sx*2*pi*(1:2/max([Lx Ly]):fs)/fs)+0.5)));
circY = outputPointsY(ceil(length(outputPointsY)*...
    (Ry*sin(Sy*2*pi*(1:2/max([Lx Ly]):fs)/fs + 0.5*pi)+0.5)));

%Set speeds for left and right output
Lspeed = 50;
Rspeed = 30;

%% Create the Output Vector
phiOutL = zeros(M,1);
phiOutR = zeros(M,1);
disp('Create PhiOut')
if phasing == true %if true, precompute phiOutL and R for all ...
    possible output positions
    phiOutLPre = zeros(M,length(circX));
    phiOutRPre = zeros(M,length(circX));
    for t = 1:length(circX)
        for m = 1:M
            phiOutLPre(m,t) = ...
                (4/(Lx*Ly))*sin((omega(m,2)*pi*circX(t))/Lx)*...
                sin((omega(m,3)*pi*circY(t))/Ly);
            phiOutRPre(m,t) = ...
                (4/(Lx*Ly))*sin((omega(m,2)*pi*circX(t))/Lx)*...
                sin((omega(m,3)*pi*circY(t))/Ly);
        end
    end
else %if false, create PhiOutL and R for a their set output positions
    for m = 1:M
        phiOutL(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*outL(1))/Lx)*...
            sin((omega(m,3)*pi*outL(2))/Ly);
        phiOutR(m,1) = (4/(Lx*Ly))*sin((omega(m,2)*pi*outR(1))/Lx)*...
            sin((omega(m,3)*pi*outR(2))/Ly);
    end
end

%% If the platemesh is desired, set to 'true'
creaMesh = false;
if creaMesh == true
    disp('Create Mesh')
    gridSize = 102; %set gridsize/meter
    modeAdd = false;
    meshFunc = zeros(gridSize*Ly-(Ly-1),gridSize*Lx-(Lx-1)); ...
        %initialise the mesh
    %create modeshape over x and y for every mode
    for x = 1:gridSize*Lx-(Lx-1)
        for y = 1:gridSize*Ly-(Ly-1)
            for mode = 1:length(omega(:,1))
                meshFunc(y,x,mode) = (4/(Lx*Ly))*...
                    sin(omega(mode,2)*pi*((x-1)/(gridSize-1))/Lx)*...
                    sin(omega(mode,3)*pi*((y-1)/(gridSize-1))/Ly);
            end
        end
    end
end

%% Damping
if physDamp == true %if physical damping is true..
    %% Calculate thermoelastic damping
    %Set thermal coefficients
    R1 = 4.94e-3;
    C1 = 2.98e-4;
    %Calculate damping coefficient and damping factor
    n1 = (omega(:,1)*R1*C1)./((omega(:,1).^2*h^2)+((C1^2)/(h^2)));
    alphaTH = (omega(:,1)/2).*n1;

    %% Calculate Radiation Damping
    fc = (ca^2)/(2*pi*sqrt(kSquared)); %calculate critical frecuency
    phiRad = sqrt((omega(:,1)/(2*pi))/fc);
    g = ((1-phiRad.^2).*log((1+phiRad)./(1-phiRad))+2.*phiRad)./...
        ((1-phiRad.^2).^(3/2));
    alphaRadPre = ...
        (1/(4*pi^2))*(ca*pa)/(rho*h)*((2*(Lx+Ly))/(Lx*Ly))*(ca/fc);
    alphaRad = alphaRadPre.*g;

    %% Calculate total damping
    alphaTot = alphaRad+alphaTH;
    T60 = 3.*log(10)./alphaTot; %reverberation time
end
if physDamp == true %if physical damping is true calculate loss ...
    coefficients using
    cm(1:length(omega(:,1)),1) = 12.*(log(10)./T60); %physical damping
else
    cm(1:length(omega(:,1)),1) = 12.*(log(10)./decay); %decay value
end

%% Initialise update equation
k = 1/fs;
qNext = zeros(length(omega(:,1)),1);
qNow = zeros(length(omega(:,1)),1);
qPrev = zeros(length(omega(:,1)),1);

%calculate coefficients
factorA = (1/k^2)+(cm/(rho*h*k));
factorB = ((2/k^2)-(omega(:,1)).^2);
factorIn = ((phiIn)./(rho*h));
```

```
factorC = ((cm/(rho*h*k))-(1/(k^2)));

%calculate normalised (with respect to A) coefficients
factorIndA = factorIn./factorA;
factorCdA = factorC./factorA;
factorBdA = factorB./factorA;

%% Set up dynamic variables if stretching == 1
if stretching == 1
    Lxnew = Lx;
    Lynew = Ly;
    hNew = h;
    kSquaredNew = kSquared;
    start = 44100; %set startingpoint of stretching (in samples)
    ending = 44100+44100; %set endpoint of stretching (in samples)
    omega = sortrows(omega,[2,3]); %sort according to modes
    omegaShift = omega;
end
change = 0;

outputShapeL = zeros(2,length(input));
outputShapeR = zeros(2,length(input));
ind = 1;

%% Main loop
disp('Loop');
tic
output = zeros(2,length(input)); %initialise output
output2 = zeros(2,length(input)); %initialise normalised output

update = true; %if false, use the filter function, else use the ...
    update equation
if update == false
    %initialise filter coefficients
    b = zeros(length(omega(:,1)),2);
    a = zeros(length(omega(:,1)),3);
    for mode = 1:length(omega(:,1))
        b(mode,:) = [0 factorIn(mode)];
        a(mode,:) = [1 -factorBdA(mode) -factorCdA(mode)];
         output(1,:) = output(1,:) + ...
             phiOutL(mode).*filter(b(mode,:),a(mode,:),input)';
         output(2,:) = output(2,:) + ...
             phiOutR(mode).*filter(b(mode,:),a(mode,:),input)';
    end
    for i = 1:2
        output2(i,:) = output(i,:)/max(abs(output(i,:)));
    end
else

    for t = 1:length(input)
    %the update equation

    if t == 1 %there is no value t-1 at t=1, so set input to 0
        qNext =(factorBdA.*qNow+factorCdA.*qPrev+factorIndA.*0);
    else
        qNext ...
            =(factorBdA.*qNow+factorCdA.*qPrev+factorIndA.*input(t-1));
    end

    %% draw functions
    if t > 0 && mod(t,1) == 0 && creaMesh == true
        plate = zeros(gridSize*Ly-(Ly-1),gridSize*Lx-(Lx-1));
        for modeM = 1:length(omega(:,1))
            plate = plate + qNext(modeM).*meshFunc(:,:,modeM);
            if modeAdd == true && mod(modeM,ceil(modeM/50)) == 0 && ...
                    t == 5
                mesh(plate);
                title(['Modes included: ' num2str(modeM)])
                drawnow;
            end
        end
        mesh(plate);
        zlim([-5e-4 5e-4])
        set(gca,'CLim',[-7e-7 7e-5])
        title(['Sample: ' num2str(t)]);
        drawnow;
    end

    %% stretching (only works correctly if all mode reduction is false)
    if stretching ~= 0 %if sheet size changes
        if t > start
            change = 1;
            if mod(t,100) == 0 || stretching == 2 %for every 100th ...
                    sample check:
                m1check = 1;
                m1Prev = stablem1;
                m2Prev = stablem2;
                stableValm1 = 0;
                stableValm2 = 0;

                %change dynamic variables over time
                %Lxnew = Lx*(1+(t-start)/44100); %change in width
                Lynew = Ly*(1-(t-start)/(4*fs)); %change in height
                %hNew = h * (1-(t-start)/(4*fs)); %change in thickness
                %kSquared = (E*hNew^2)/(12*rho*(1-v^2)); %update ...
                    stiffnessfactor

                %% Draw the plate every 1000th sample
                draw = false;
                if draw == true && mod(t,1000) == 0
                    clf
                    if flanging == false
                    plateScat = [-Lxnew/2 -Lynew/2 hNew/2 ...
                        -hNew/2;Lxnew/2 -Lynew/2 hNew/2 ...
                        -hNew/2; Lxnew/2 Lynew/2 hNew/2 ...
                        -hNew/2; -Lxnew/2 Lynew/2 hNew/2 ...
                        -hNew/2;-Lxnew/2 -Lynew/2 hNew/2 -hNew/2];
                    micsOut = [qL(1)*Lxnew-Lxnew/2 ...
                        qL(2)*Lynew-Lynew/2 0; ...
                        qR(1)*Lxnew-Lxnew/2 qR(2)*Lynew-Lynew/2 0];
                    micIn = [p(1)*Lxnew-Lxnew/2 ...
                        p(2)*Lynew-Lynew/2 0];
                end
                if flanging == true
                    plateScat = [-Lxnew/2 -Lynew/2 hNew/2 ...
                        -hNew/2;Lxnew/2 -Lynew/2 hNew/2 ...
                        -hNew/2; Lxnew/2 Lynew/2 hNew/2 ...
                        -hNew/2; -Lxnew/2 Lynew/2 hNew/2 ...
                        -hNew/2;-Lxnew/2 -Lynew/2 hNew/2 -hNew/2];
                    micsOut = ...
                        [(circX(floor(mod(t,length(circX))+1))...
                    /Lx)*Lxnew-Lxnew/2...
                    (circY(floor(mod(t,length(circY))+1))/Ly)*...
                    Lynew-Lynew/2 0;...
                    (circX(floor(mod(t/2,length(circX))+1))/Lx)*...
                    Lxnew-Lxnew/2...
                    (circY(floor(mod(t/2,length(circY))+1))/Ly)*...
                    Lynew-Lynew/2 0];
                    micIn = [p(1)*Lxnew-Lxnew/2 ...
                        p(2)*Lynew-Lynew/2 0];
                end
                hold on;
                plot3(plateScat(:,1),plateScat(:,2), ...
                    plateScat(:,3));
                plot3(plateScat(:,1),plateScat(:,2), ...
                    plateScat(:,4));
                scatter3(micsOut(:,1),micsOut(:,2),...
                micsOut(:,3),[],[1 0 0]);
                scatter3(micIn(:,1), ...
                    micIn(:,2),micIn(:,3),[],[0 1 0]);
                ylim([-3 3])
                xlim([-3 3])
                zlim([-1*h 1*h])
                view(38,89)
                drawnow;
            end

            % calculate highest stable modes
            m1Var = 0;
            while m1Var == 0
                stableValm1 = ((m1check*pi/Lxnew)^2 + ...
                    (1*pi/Lynew)^2)*sqrt(kSquared);
                if stableValm1 > fs*2
```

```
        m1Var = 1;
    else
        m1check = m1check+1;
    end
end
stablem1 = m1check-1;
m2check = 1;
m2Var = 0;
while m2Var == 0
    stableValm2 = ((1*pi/Lxnew)^2 + ...
        (m2check*pi/Lynew)^2)*sqrt(kSquared);
    if stableValm2 > fs*2
        m2Var = 1;
    else
        m2check = m2check+1;
    end
end
stablem2 = m2check-1;

%% Check for unstable eigenfrequencies (happens ...
    when decreasing Lx or Ly and increasing h)
tooHighVect = find(omega(:,1) > fs*2);
if ~isempty(find(omega(:,1) > fs*2,1))
    % delete if omega > 2fs
    omega(tooHighVect,:) = [];
    qNext(tooHighVect,:)= [];
    qNow(tooHighVect,:) = [];
    qPrev(tooHighVect,:) = [];
else
    %% Add eigenfrequencies that are now stable ...
        (happens when increasing Lx or Ly and ...
        decreasing h)
    if m1Prev < stablem1 || m2Prev < stablem2
    % if highest stable mode has changed create new ...
        omega matrix
        omegaPrev = sortrows(omega,[2,3]); %set for ...
            comparison later
        omega = zeros(100000,3);
        val = 0;
        m = 1;
        m1 = 1;
        m2 = 1;
        while val < fs*2
            val = ((m1*pi/Lxnew)^2 + ...
                (m2*pi/Lynew)^2)*sqrt(kSquared);
            if val < fs*2
                omega(m,1) = val;
                omega(m,2) = m1;
                omega(m,3) = m2;
                m2 = m2 + 1;
                m = m+1;
        else
            m2 = 1;
            m1 = m1 + 1;
            val = ((m1*pi/Lxnew)^2 + ...
                (m2*pi/Lynew)^2)*sqrt(kSquared);
            if val > fs*2
                break;
            end
        end
    end
    omega = omega(1:m-1,:);
    %% OPTION ONE: stable horizontal mode increased
    if m1Prev < stablem1
        index = 0;
        for j = 1:length(omega)
            if j-index < length(omegaPrev)
                if omegaPrev(j-index,2) ~= ...
                    omega(j,2)
                    %insert 0's at right locations
                    qNext = [qNext(1:j-1,1); 0; ...
                        qNext(j:end,1)];
                    qNow = [qNow(1:j-1,1); 0; ...
                        qNow(j:end,1)];
                    qPrev = [qPrev(1:j-1,1); 0; ...
                        qPrev(j:end,1)];
                    index = index + 1;
                end
            else
                %insert 0's at the end
                qNext = [qNext ; ...
                    zeros(length(omega)-...
                    length(qNext),1)];
                qNow = [qNow ; ...
                    zeros(length(omega)-...
                    length(qNow),1)];
                qPrev = [qPrev ; ...
                    zeros(length(omega)-...
                    length(qPrev),1)];
            end
        end
    else
        %% OPTION TWO: stable vertical mode ...
            increased
        if m2Prev < stablem2 % Just adding ...
            zeros instead of this smart-adding ...
            gives metallic sounds
            index = 0;
            for j = 1:length(omega)
                if j-index < length(omegaPrev)
                    %insert 0's at right locations
```

```
                            if omegaPrev(j-index,2) ~= ...
                                omega(j,2)
                                qNext = ...
                                    [qNext(1:j-1,1); 0; ...
                                    qNext(j:end,1)];
                                qNow = [qNow(1:j-1,1); ...
                                    0; qNow(j:end,1)];
                                qPrev = ...
                                    [qPrev(1:j-1,1); 0; ...
                                    qPrev(j:end,1)];
                                index = index + 1;
                            end
                        else
                            %insert 0's at the end
                            qNext = [qNext ; ...
                                zeros(length(omega)-...
                                length(qNext),1)];
                            qNow = [qNow ; ...
                                zeros(length(omega)-...
                                length(qNow),1)];
                            qPrev = [qPrev ; ...
                                zeros(length(omega)-...
                                length(qPrev),1)];
                        end
                    end
                end
            end
        end
    end

    %change eigenfrequency values according to Lxnew ...
        and Lynew
    for i = 1:length(omega(:,1))
        omega(i,1)=(((omega(i,2)*pi)/Lxnew)^2 + ...
            ((omega(i,3)*pi)/Lynew)^2)*sqrt(kSquared);
    end

    %% Create PhiIn and -Outs based on new eigenfrequencies
    omega = sortrows(omega,[2,3]);
    M = length(omega);
    phiIn = zeros(M,1);
    phiOutL = zeros(M,1);
    phiOutR = zeros(M,1);
    for m = 1:M
        phiIn(m,1) = (4/(Lxnew*Lynew))*...
        sin((omega(m,2)*pi*p(1)*Lxnew)/Lxnew)*...
        sin((omega(m,3)*pi*p(2)*Lynew)/Lynew);
        if flanging == false
            %they move relative to the plate dimension
            phiOutL(m,1) = (4/(Lxnew*Lynew))*...
            sin((omega(m,2)*pi*qL(1)*Lxnew)/Lxnew)*...
```

```
                            sin((omega(m,3)*pi*qL(2)*Lynew)/Lynew);
                            phiOutR(m,1) = (4/(Lxnew*Lynew))*...
                            sin((omega(m,2)*pi*qR(1)*Lxnew)/Lxnew)*...
                            sin((omega(m,3)*pi*qR(2)*Lynew)/Lynew);
                        end
                    end
                    if physDamp == true
                        %Calculate thermoelastic damping
                        n1 = ...
                            (omega(:,1)*R1*C1)./((omega(:,1).^2*hNew^2)+...
                        ((C1^2)/(hNew^2)));
                        alphaTH = (omega(:,1)/2).*n1;

                        % calculate radiation damping
                        fc = (ca^2)/(2*pi*sqrt(kSquared));
                        phiRad = sqrt((omega(:,1)/(2*pi))/fc);
                        g = ((1-phiRad.^2).*log((1+phiRad)./...
                        (1-phiRad))+2.*phiRad)./((1-phiRad.^2).^(3/2));
                        alphaRadPre = (1/(4*pi^2))*(ca*pa)/(rho*hNew)*...
                        ((2*(Lxnew+Lynew))/(Lxnew*Lynew))*(ca/fc);
                        alphaRad = alphaRadPre.*g;
                        alphaTot = alphaRad+alphaTH;
                        T60 = 3.*log(10)./alphaTot;

                        cm = zeros(length(omega(:,1)),1);
                        cm(1:length(omega(:,1)),1) = 12.*(log(10)./T60);
                    else
                        cm = zeros(length(omega(:,1)),1);
                        cm(1:length(omega(:,1)),1) = 12.*(log(10)./decay);
                    end
                    %calculate coefficients
                    factorA = (1/k^2)+(cm/(rho*hNew*k));
                    factorB = ((2/k^2)-(omega(:,1)).^2);
                    factorIn = ((phiIn)./(rho*h));
                    factorC = ((cm/(rho*hNew*k))-(1/(k^2)));

                    %calculate normalised (with respect to A) coefficients
                    factorIndA = factorIn./factorA;
                    factorCdA = factorC./factorA;
                    factorBdA = factorB./factorA;
                    if stretching == 2
                        stretching = 0;
                    end
                end
                if t >= ending && stretching ~= 0 %stop stretching if t ...
                    > ending
                    stretching = 2;
                end
            end
        end

```

```
    %% Flanging
    if flanging == true
        if change == 1 %if stretching occurs
            for m = 1:M
                phiOutL(m,1) = (4/(Lxnew*Lynew))*...
                sin((omega(m,2)*pi*...
                ((circX(floor(t/Lspeed)+1))/Lx)*Lxnew)/Lxnew)*...
                sin((omega(m,3)*pi*...
                ((circY(floor(t/Lspeed)+1))/Ly)*Lynew)/Lynew);
                phiOutR(m,1) = (4/(Lxnew*Lynew))*...
                sin((omega(m,2)*pi*...
                ((circX(floor(t/Rspeed)+1))/Lx)*Lxnew)/Lxnew)*...
                sin((omega(m,3)*pi*...
                ((circY(floor(t/Rspeed)+1))/Ly)*Lynew)/Lynew);
            end
        else %otherwise use precalculated PhiOuts
            phiOutL = ...
               phiOutLPre(:,floor(mod(t/Lspeed,length(circX))+1));
            phiOutR = ...
               phiOutRPre(:,floor(mod(t/Rspeed,length(circX))+1));
        end
    end

    %fill in the output at sample t
    output(1,t) = qNext'*phiOutL;
    output(2,t) = qNext'*phiOutR;

    %update qVectors
    qPrev = qNow;
    qNow = qNext;
end
    % Normalise Output
    for i = 1:2
        output2(i,:) = output(i,:)/max(abs(output(i,:)));
    end
end
```

B.2 Real-Time Plugin

Notes: This code works only with MATLABs Audio System Toolbox and can be tested in its Audio Test Bench. The function `initPlate` *contains the first part (before the Main Loop) of the algorithm above.*

```
classdef realTimePlateReverbPlugin < audioPlugin
    properties
        % Use this section to initialize properties that the end-user
        % interacts with.
        wetness = 50;
        cm = false;
        Lx = 2;
        Ly = 1;
        cents = 10;
        calcCents = true;
        delModes = true;
        init = true;
        phasing = false;
        smoothLx = false;
        smoothLy = false;
        square = true;
        stretching = true;
        LFO = false;
    end
    properties (Access = private)
        % Use this section to initialize properties that the ...
            end-user does
        % not interact with directly.
        currentSample = 0;
        circXLength = 0;
        LxSmooth = 2;
        LySmooth = 1;
        Lxpre = 2;
        Lypre = 1;
        T60 = 4;
        rho = 7850;
        h = 0.0005;
        lengthOmega = 832;
        omega = zeros(832,3);
        factorBdA = zeros(832,1);
        factorCdA = zeros(832,1);
        factorIndA = zeros(832,1);
        phiOutL = zeros(832,1);;
        phiOutR = zeros(832,1);
        phiOutLPre = zeros(832,22050);
        phiOutRPre = zeros(832,22050);
        qNext = zeros(832,1);
        qNow = zeros(832,1);
        qPrev = zeros(832,1);
        qPre = zeros(832,1);
        samp = 0;
        prevLengthOmega = 0;
        p = [0.4 0.415];
        qL = [0.1 0.45];
        qR = [0.84 0.45];
        saveMat = zeros(10000,2);
        initNum = 1;
    end
    properties (Constant)
        % This section contains instructions to build your audio plugin
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public
        % property with a tunable parameter.
        PluginInterface = audioPluginInterface(...
        audioPluginParameter('wetness',...
            'DisplayName','Dry/Wet',...
            'Label','%',...
            'Mapping',{'lin',0,100}),...
        audioPluginParameter('Lx',...
            'DisplayName','Plate Width',...
            'Label','m',...
            'Mapping',{'lin',1,3}),...
        audioPluginParameter('Ly',...
            'DisplayName','Plate Height',...
            'Label','m',...
            'Mapping',{'lin',0.5,2}),...
        audioPluginParameter('cents',...
            'DisplayName','Cents',...
            'Label','cents',...
            'Mapping',{'lin',0.01,10}),...
        audioPluginParameter('calcCents',...
            'DisplayName','Calculate Cents',...
            'Label','off/on',...
            'Mapping',{'enum','off','on'}),...
        audioPluginParameter('phasing',...
            'DisplayName','Phasing',...
            'Label','off/on',...
            'Mapping',{'enum','off','on'}),...
        audioPluginParameter('stretching',...
            'DisplayName','Stretching',...
            'Label','off/on',...
            'Mapping',{'enum','off','on'}),...
        audioPluginParameter('LFO',...
            'DisplayName','LFO Stretch',...
            'Label','off/on',...
            'Mapping',{'enum','off','on'}),...
        audioPluginParameter('cm',...
            'DisplayName','Physical Damping',...
            'Label','off/on',...
            'Mapping',{'enum','off','on'}),...
        audioPluginParameter('init',...
            'DisplayName','Re-initialise',...
            'Label','Click twice',...
            'Mapping',{'enum','off','on'}))
    end
    methods
        function plugin = realTimePlateReverbPlugin        %<---

        end
        function out = process(plugin, in)
            % This section contains instructions to process the ...
                input audio
            % signal. Use plugin.MyProperty to access a property of ...
                your
            % plugin.
            if plugin.init == true
                options = [plugin.delModes plugin.square ...
                    plugin.calcCents plugin.phasing ...
                    plugin.stretching plugin.cents plugin.cm]; ...
                    %[delModes calcCent phasing stretching cents]
                disp('Initialising Plugin')
                [plugin.factorBdA, plugin.factorCdA, ...
                    plugin.factorIndA, plugin.omega, ...
                    plugin.phiOutL, ...
                plugin.phiOutR, plugin.phiOutLPre, ...
                    plugin.phiOutRPre, plugin.p, plugin.qL, ...
                    plugin.qR, plugin.circXLength]...
                = initPlate(plugin.Lx,plugin.Ly,options);
%                 Lspeed = 50;
%                 Rspeed = 30;
                plugin.qNext = zeros(length(plugin.omega(:,1)),1);
                plugin.qPre = zeros(length(plugin.omega(:,1)),1);
                plugin.qNow = zeros(length(plugin.omega(:,1)),1);
                plugin.qPrev = zeros(length(plugin.omega(:,1)),1);
                plugin.samp = 0;
                plugin.init = false;
                plugin.lengthOmega = length(plugin.omega(:,1));
                plugin.prevLengthOmega = length(plugin.omega(:,1));
                plugin.initNum = plugin.initNum + 1;
                disp(plugin.lengthOmega)
            end
            out = zeros(length(in),2);
            if plugin.prevLengthOmega > plugin.lengthOmega
                qNextLoop = plugin.qNext(1:plugin.lengthOmega);
                qNowLoop = plugin.qNow(1:plugin.lengthOmega);
                qPrevLoop = plugin.qPrev(1:plugin.lengthOmega);
                factorBdALoop = plugin.factorBdA(1:plugin.lengthOmega);
                factorCdALoop = plugin.factorCdA(1:plugin.lengthOmega);
                factorIndALoop = ...
                    plugin.factorIndA(1:plugin.lengthOmega);
                phiOutLLoop = plugin.phiOutL(1:plugin.lengthOmega);
                phiOutRLoop = plugin.phiOutR(1:plugin.lengthOmega);
            else
                if plugin.prevLengthOmega < plugin.lengthOmega
                    qNextLoop = ...
                        [plugin.qNext;zeros(plugin.lengthOmega - ...
                        length(plugin.qNext),1)];
                    qNowLoop = ...
                        [plugin.qNow;zeros(plugin.lengthOmega - ...
                        length(plugin.qNow),1)];
                    qPrevLoop = ...
                        [plugin.qPrev;zeros(plugin.lengthOmega - ...
                        length(plugin.qPrev),1)];
                    factorBdALoop = ...
                        [plugin.factorBdA;zeros(plugin.lengthOmega ...
                        - length(plugin.factorBdA),1)];
                    factorCdALoop = ...
                        [plugin.factorCdA;zeros(plugin.lengthOmega ...
                        - length(plugin.factorCdA),1)];
                    factorIndALoop = ...
                        [plugin.factorIndA;zeros(plugin.lengthOmega ...
                        - length(plugin.factorIndA),1)];
                    phiOutLLoop = ...
                        [plugin.phiOutL;zeros(plugin.lengthOmega - ...
                        length(plugin.phiOutL),1)];
                    phiOutRLoop = ...
                        [plugin.phiOutR;zeros(plugin.lengthOmega - ...
                        length(plugin.phiOutR),1)];
                else
                    qNextLoop = plugin.qNext;
                    qNowLoop = plugin.qNow;
                    qPrevLoop = plugin.qPrev;
                    factorBdALoop = plugin.factorBdA;
                    factorCdALoop = plugin.factorCdA;
                    factorIndALoop = plugin.factorIndA;
                    phiOutLLoop = plugin.phiOutL;
                    phiOutRLoop = plugin.phiOutR;
                end
            end

            if plugin.stretching == true
                M = plugin.lengthOmega;
                pLoop = plugin.p;
                qLLoop = plugin.qL;
                qRLoop = plugin.qR;
                kSquared = 0.5833;
                k = 1/44100;
                LxSmoothUse = plugin.LxSmooth;
                LySmoothUse = plugin.LySmooth;
                sS = round(M/20);
                if abs(plugin.Lx - plugin.Lxpre) > 1/sS
                    plugin.smoothLx = true;
%                     disp('smoothLx:');
%                     disp(plugin.smoothLx);
                else
                    plugin.smoothLx = false;
                end
                if abs(plugin.Ly - plugin.Lypre) > 1/sS
                    plugin.smoothLy = true;
%                     disp(plugin.smoothLy);
                else
                    plugin.smoothLy = false;
%                     disp('smoothLy:')
%                     disp(plugin.smoothLy);
```

```
                end
                if plugin.smoothLx == true
                    if round(LxSmoothUse*sS)/sS > ...
                        round(plugin.Lx*sS)/sS
                        LxSmoothUse = LxSmoothUse - 1/sS;
                    else
                        if round(LxSmoothUse*sS)/sS < ...
                            round(plugin.Lx*sS)/sS
                            LxSmoothUse = LxSmoothUse + 1/sS;
                        end
                    end
                else
                    LxSmoothUse = plugin.Lx;
                end
                if plugin.smoothLy == true
                    if round(LySmoothUse*sS)/sS > ...
                        round(plugin.Ly*sS)/sS
                        LySmoothUse = LySmoothUse - 1/sS;
                    else
                        if round(LySmoothUse*sS)/sS < ...
                            round(plugin.Ly*sS)/sS
                            LySmoothUse = LySmoothUse + 1/sS;
                        end
                    end
                else
                    LySmoothUse = plugin.Ly;
                end
                plugin.Lxpre = LxSmoothUse;
                plugin.Lypre = LySmoothUse;
%                 disp(LxSmoothUse);

                if plugin.LFO == true
                    LxLoop = LxSmoothUse+...
                    (sin(2*pi*plugin.currentSample/44100)/4);
                else
                    LxLoop = LxSmoothUse;
                end
                plugin.LxSmooth = LxSmoothUse;
                plugin.LySmooth = LySmoothUse;
                LyLoop = plugin.LySmooth;
                omegaLoop = plugin.omega;
                i = 0;
                phiOutLLoop = zeros(M,1);
                phiOutRLoop = zeros(M,1);
                factorALoop = (1/k^2)+((12.*(log(10)./plugin.T60))/...
                (7850*0.0005*k));
                rhoUse = plugin.rho;
                hUse = plugin.h;
                factorALoopAll = factorALoop*rhoUse*hUse;
                for m = 1:M
                    omegaLoop(m,1)= (((omegaLoop(m,2)*pi)/LxLoop)^2 ...
                        + ...
                        ((omegaLoop(m,3)*pi)/LyLoop)^2)*sqrt(kSquared);
                    if omegaLoop(m,1) < 44100*2
                        i = i+1;
                    end
                    factorBdALoop(m,1) = ...
                        ((2/k^2)-(omegaLoop(m,1)).^2)./factorALoop;
                    factorIndALoop(m,1) = ((4/(LxLoop*LyLoop))*...
                    sin((omegaLoop(m,2)*pi*pLoop(1)*LxLoop)/LxLoop)*...
                    sin((omegaLoop(m,3)*pi*pLoop(2)*LyLoop)/LyLoop))...
                    ./(factorALoopAll);
                    phiOutLLoop(m,1) = (4/(LxLoop*LyLoop))*...
                    sin((omegaLoop(m,2)*pi*qLLoop(1)*LxLoop)/LxLoop)*...
                    sin((omegaLoop(m,3)*pi*qLLoop(2)*LyLoop)/LyLoop);
                    phiOutRLoop(m,1) = (4/(LxLoop*LyLoop))*...
                    sin((omegaLoop(m,2)*pi*qRLoop(1)*LxLoop)/LxLoop)*...
                    sin((omegaLoop(m,3)*pi*qRLoop(2)*LyLoop)/LyLoop);
                end
                index = zeros(1,i);
                i = 1;
                for m = 1:M
                    if omegaLoop(m,1) < 44100*2
                        index(1,i) = m;
                        i = i + 1;
                    end
                end
                plugin.omega = omegaLoop;
                qNextLoopInd = qNextLoop(index);
                qNowLoopInd = qNowLoop(index);
                qPrevLoopInd = qPrevLoop(index);
                factorBdALoopInd = factorBdALoop(index);
                factorCdALoopInd = factorCdALoop(index);
                factorIndALoopInd = factorIndALoop(index);
                phiOutLLoopInd = phiOutLLoop(index);
                phiOutRLoopInd = phiOutRLoop(index);
            else
                qNextLoopInd = qNextLoop;
                qNowLoopInd = qNowLoop;
                qPrevLoopInd = qPrevLoop;
                factorBdALoopInd = factorBdALoop;
                factorCdALoopInd = factorCdALoop;
                factorIndALoopInd = factorIndALoop;
                phiOutLLoopInd = phiOutLLoop;
                phiOutRLoopInd = phiOutRLoop;
                index = 1:plugin.lengthOmega;
            end
%             phiOutLPhase = plugin.phiOutLPre;
%             phiOutRPhase = plugin.phiOutRPre;
            curSamp = plugin.currentSample;
            lengthCircX = plugin.circXLength;
            for t = 1:length(in)
```

```
                if t == 1
                    qNextLoopInd = (factorBdALoopInd.*qNowLoopInd+...
                    factorCdALoopInd.*qPrevLoopInd+...
                    factorIndALoopInd.*plugin.samp);
                else
                    qNextLoopInd = (factorBdALoopInd.*qNowLoopInd+...
                    factorCdALoopInd.*qPrevLoopInd+...
                    factorIndALoopInd.*in(t-1,1));
                end
%                     if plugin.phasing == true
%                         phiOutLLoopInd = ...
   phiOutLPhase(:,floor(mod((curSamp+t)/4,lengthCircX)+1));
%                         phiOutRLoopInd = ...
   phiOutRPhase(:,floor(mod((curSamp+t)/8,lengthCircX)+1));
%                     end
%
                out(t,1) = ...
                    plugin.wetness/100*25000*sum(qNextLoopInd.*...
                phiOutLLoopInd)...
                    +(1-plugin.wetness/100)*in(t,1);
                out(t,2) = ...
                    plugin.wetness/100*25000*sum(qNextLoopInd.*...
                phiOutRLoopInd)...
                    +(1-plugin.wetness/100)*in(t,1);
                qPrevLoopInd = qNowLoopInd;
                qNowLoopInd = qNextLoopInd;
            end
%             if toc > 0.015
%                 plugin.prevLengthOmega = plugin.lengthOmega;
%                 plugin.lengthOmega = plugin.lengthOmega - 100;
%                 disp(plugin.lengthOmega)
%             else
%                 if toc < 0.01
%                     plugin.prevLengthOmega = plugin.lengthOmega;
%                     plugin.lengthOmega = plugin.lengthOmega + 100;
%                     disp(plugin.lengthOmega)
%                 end
%             end
            %disp(get(gca, 'CurrentPoint'));
            plugin.samp = in(end);
            plugin.qNext(index) = qNextLoopInd;
            plugin.qPrev(index) = qPrevLoopInd;
            plugin.qNow(index) = qNowLoopInd;
            plugin.currentSample = plugin.currentSample + ...
                length(in(:,1));
%             figure(2);
%             sc    atter(1:length(omegaLoop),sort(omegaLoop(:,1)))
%             drawnow;
        end
        function reset(plugin)
            % This section contains instructions to reset the plugin
```

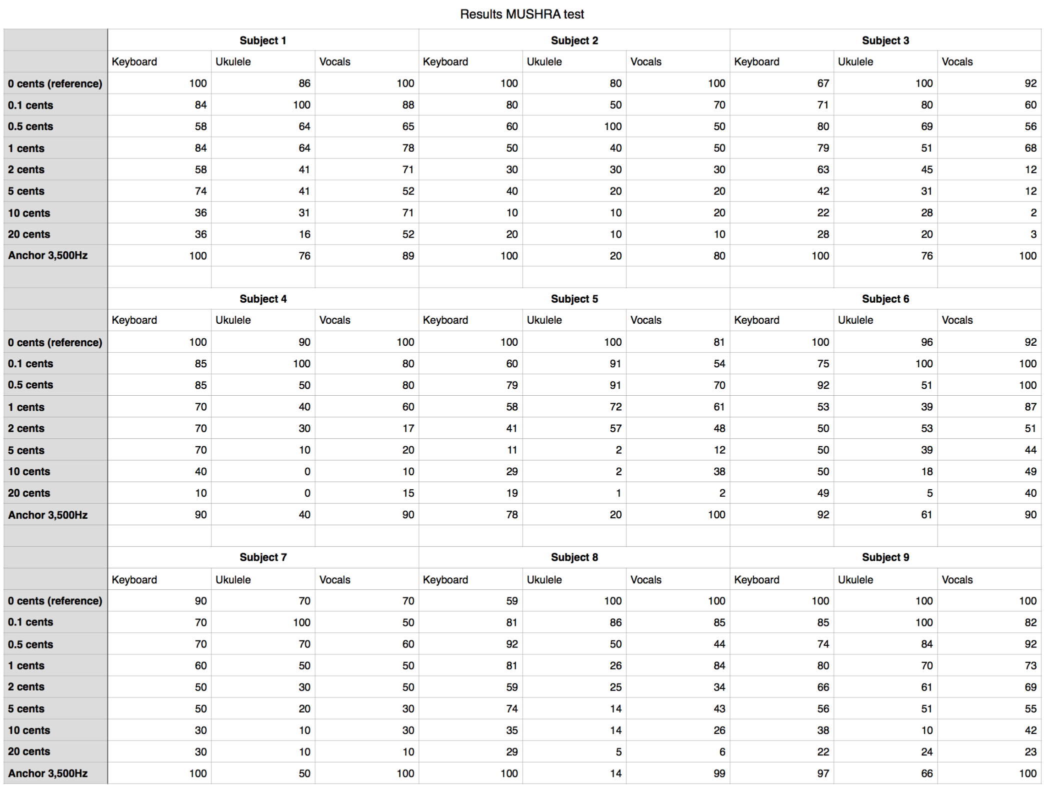
```
                % between uses or if the environment sample rate changes.
            end


        end
    end
```

Appendix C

Results Evaluation

C.1 Results MUSHRA test



Results MUSHRA test

	Subject 1			Subject 2			Subject 3		
	Keyboard	Ukulele	Vocals	Keyboard	Ukulele	Vocals	Keyboard	Ukulele	Vocals
0 cents (reference)	100	86	100	100	80	100	67	100	92
0.1 cents	84	100	88	80	50	70	71	80	60
0.5 cents	58	64	65	60	100	50	80	69	56
1 cents	84	64	78	50	40	50	79	51	68
2 cents	58	41	71	30	30	30	63	45	12
5 cents	74	41	52	40	20	20	42	31	12
10 cents	36	31	71	10	10	20	22	28	2
20 cents	36	16	52	20	10	10	28	20	3
Anchor 3,500Hz	100	76	89	100	20	80	100	76	100
	Subject 4			**Subject 5**			**Subject 6**		
	Keyboard	Ukulele	Vocals	Keyboard	Ukulele	Vocals	Keyboard	Ukulele	Vocals
0 cents (reference)	100	90	100	100	100	81	100	96	92
0.1 cents	85	100	80	60	91	54	75	100	100
0.5 cents	85	50	80	79	91	70	92	51	100
1 cents	70	40	60	58	72	61	53	39	87
2 cents	70	30	17	41	57	48	50	53	51
5 cents	70	10	20	11	2	12	50	39	44
10 cents	40	0	10	29	2	38	50	18	49
20 cents	10	0	15	19	1	2	49	5	40
Anchor 3,500Hz	90	40	90	78	20	100	92	61	90
	Subject 7			**Subject 8**			**Subject 9**		
	Keyboard	Ukulele	Vocals	Keyboard	Ukulele	Vocals	Keyboard	Ukulele	Vocals
0 cents (reference)	90	70	70	59	100	100	100	100	100
0.1 cents	70	100	50	81	86	85	85	100	82
0.5 cents	70	70	60	92	50	44	74	84	92
1 cents	60	50	50	81	26	84	80	70	73
2 cents	50	30	50	59	25	34	66	61	69
5 cents	50	20	30	74	14	43	56	51	55
10 cents	30	10	30	35	14	26	38	10	42
20 cents	30	10	10	29	5	6	22	24	23
Anchor 3,500Hz	100	50	100	100	14	99	97	66	100

C.2 Results Questionnaire

Timestamp	How old are you?	What is your gender?	What is the highest degree or level of school you have completed?	On a typical day, how many hours do you spend listening to music?	Do/did you play an instrument? And if so, for how long?
2017/05/11 12:16:52 pm EET	19	Female	Elementary School	6 or more hours	0-1 years
2017/05/11 12:58:07 pm EET	24	Female	Bachelor of science	6 or more hours	2-3 years
2017/05/11 12:58:55 pm EET	25	Male	B.Sc	4-5 hours	More than 6 years
2017/05/11 1:06:11 pm EET	31	Male	Bachelor of Engineering	2-3 hours	More than 6 years
2017/05/11 1:30:57 pm EET	27	Male	Bachelor	2-3 hours	2-3 years
2017/05/11 1:47:43 pm EET	27	Male	Bachelor	4-5 hours	More than 6 years
2017/05/11 1:52:13 pm EET	24	Male	Bachelor Computer Science	6 or more hours	0-1 years
2017/05/12 2:26:03 pm EET	22	Male	IB	0-1 hours	0-1 years
2017/05/15 2:42:06 pm EET	32	Male	PhD	2-3 hours	2-3 years

Table 1-2

If you answered yes on the previous question, what instrument(s) do/did you play? (guitar, piano, singing, etc.)	Do you produce music, or have you ever produced music?
	No
Bass, singing, guitar	No
Cornet, Jew's harp, didgeridoo	Yes
Mostly singing but also a bit of piano/keyboard and a very little cello	No
Guitar	No
Singing, Piano	Yes
	Yes
piano	No
Ukulele	No

Did you enjoy the sound of the reverberation effect in the following sounds: [Keyboard]	Did you enjoy the sound of the reverberation effect in the following sounds: [Ukulele]	Did you enjoy the sound of the reverberation effect in the following sounds: [Vocals]
5 - Yes, very much	3 - I'm indifferent	2 - No, not really
5 - Yes, very much	5 - Yes, very much	2 - No, not really
4 - Yes, quite	5 - Yes, very much	2 - No, not really
5 - Yes, very much	5 - Yes, very much	5 - Yes, very much
5 - Yes, very much	4 - Yes, quite	2 - No, not really
5 - Yes, very much	5 - Yes, very much	3 - I'm indifferent
4 - Yes, quite	5 - Yes, very much	4 - Yes, quite
2 - No, not really	4 - Yes, quite	3 - I'm indifferent
4 - Yes, quite	5 - Yes, very much	3 - I'm indifferent

What is your general opinion on the reverberation effect? If you're a musician/producer, would you use it? For what kind of music/sounds? Any additional comments?
I like the reverberation sound from the keyboard better because I think it have a more natural sound then the reverberation effect from the ukulele and the vocals.
I think it sounds really good, but for the vocals it sounds like they are singing in the shower or something.
it's good but very pronounced (guessing you can change the parameters though so probably fine). It's very metallic so perhaps it should be used for more dull sounds like a double bass. I'd probably not use it with my cornet but I haven't heard the resulting sound. Have you tried lowpassing it to get rid of the metallic effect?
It does have a very nice metallic sound to it. As if it was played in a shipping container, or the cargo of a big ship.   The amount of reverberation was quite a lot, so in this case I would make very ambient, slowgoing music with instruments playing single notes that would then add up to a nice harmony. I guess, I'd use it with my cello or perhaps even my voice.
I would use it for enhancing the sound of instruments. I like that you can implement such a powerful effect so easily without cumbersome plates.
Nice big and cathedral-like, fo sho I would use da shieet outta dat!
Very nice sounding reverb, would use for pads, piano, and maybe a little on vocals.
Yes, but only sometimes.
As indicated above, I found it too "rich" for the vocals, but the can probably be changed by tuning.