A Real Time Finite Difference Time Domain Spring Reverberation Simulation

Oliver Frank

A final project dissertation submitted in partial fulfilment
of the requirements for the degree of

Master of Science (MSc)

Acoustics and Music Technology

Acoustics and Audio Group

Edinburgh College of Art
University *of* Edinburgh

2025-06-23

Supervisor: Stefan Bilbao

Abstract

Spring reverberation simulation techniques are an area of active research, and of much interest to commerical audio plugin authors. In this project, several methods for simulating spring reverberation systems via a finite difference scheme were evaluated, and compared to determine their runtime characteristics. These tests were specifically carried out for single core vector processors, as is common for audio plugins. The results were quite successful, with some methods operating in realtime even for oversampled systems.

Declaration

I do hereby declare that this dissertation was composed by myself and that the work described within is my own, except where explicitly stated otherwise.

Oliver Frank
2025-06-23

Acknowledgements

I would like to thank my supervisor, Stefan Bilbao, for his guidance while working on this project. I would also like to thank my parents for their support, both financial and moral.

Contents

List of Figures

Chapter 1

Introduction

The primary objective of this project was to create an accurate physical simulation of a spring reverberation unit that was able to run at realtime. This report will document the background research necessary to understand the topic, the efforts made in pursuit of this aim, as well as the results achieved.

Spring reverberation is one of the oldest forms of artificial reverberation, and it provided a cheap and small alternative to plate reverberation and echo chambers, particularly prior to the rise of low cost, high powered integrated circuits (ICs) that make digital hardware so cheap today. Because of this, it characterizes the sound of 1960s surf and garage rock, and is still popular today because of its distinct sound. This project presents a real time implementation of a physical model of a spring reverb, and surveys various methods of efficiently computing the update of said scheme. The results achieved were promising, but require more refinement before they would be able to be used in a commercial plugin.

Initially, a more interactive simulation was proposed for this project, but on learning that there were currently no real time implementations of the finite difference time domain (FDTD) scheme that was going to be used, efforts were concentrated instead on creating one, such that the further goals laid out in the project proposal might be achieved in a commercially viable format, such as an audio plugin.

Chapter 2

Background

2.1 A Brief History of Artificial Reverberation

Reverberation is one of the more complex psychoacoustic phenomena, and plays a large role in shaping our sense of the space we are in as we move through the world. Since the dawn of recorded music, we have sought to tame reverberation, and use it to enrich the quality of the recordings we make. In this section, a short accounting of our efforts to manipulate and produce reverberation-like effects will be given, in order to place this work within the proper historical context.

The earliest methods of capturing reverberation on recordings were probably unintentional; it is an unavoidable consequence of making a recording of any sort of sound. This is because as sound waves propagate throughout a space, they collide with the various surfaces in that space, reflecting back the sound towards the recording device, be that a stylus etching a phonograph cylinder, or a microphone tracking in to a DAW. Generally, the direct sound of the source will predominate in the recording, but the delayed reflections of the sound will also be recorded, as can be seen in Figure

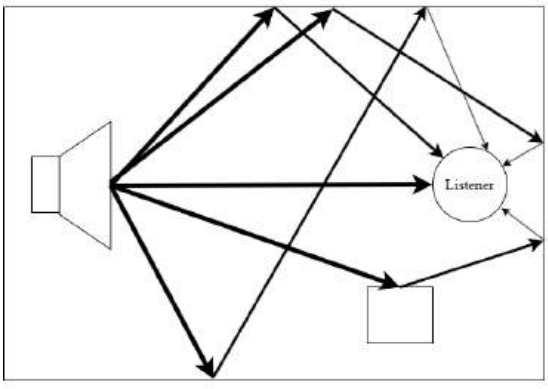

Figure 2.1: Reverberation Reflections

Figure 2.2: EMT-140 Plate Reverberator [EMT-Archiv-Lahr CC BY-SA 4.0]

2.1. The first few reflections can often be heard individually (called early reflections), but after that they tend to "smear" out.

During the early to mid twentieth century, there were many methods devised to add reverberation to an existing recording, to make it sound as though the sound was recorded in a different space than it actually had been. Echo chambers were one of the earliest, and perhaps the most straightforward. They operated by having a separate room dedicated to the effect, with a speaker and one or more microphones. Sound could be fed through the speaker, and then the reverberations would be picked up by the microphones. The recording engineer could then choose how much of this reverberant sound to add back to the original mix.

Another early technique for creating artificial reverberation was the plate reverberation device. Plate reverbs operate by having a speaker transmit sound into a large metal plate, and then having a magnetic pickup record the output elsewhere on the plate. The sound waves bounce around on the plate, in a similar manner in which they do in a room, albeit with some subtle differences that give plate reverbs their characteristic sound. An early example of a plate reverb, the EMT-140, can be seen in Figure 2.2.

One commonality shared by both echo chambers and plate reverberation units are that they were both very expensive, and took up huge amounts of space. This restricted their use almost exclusively to commercial recording studios. For DIY musicians of the 50s and 60s, a much more popular option was the spring reverb (shown in Figure 2.3), a small device that housed one or more springs through which sound is played, and then recorded at the other end via electromagnetic pickups. These devices epitomize the sound of surf and garage rock, and although they don't sound particularly realistic

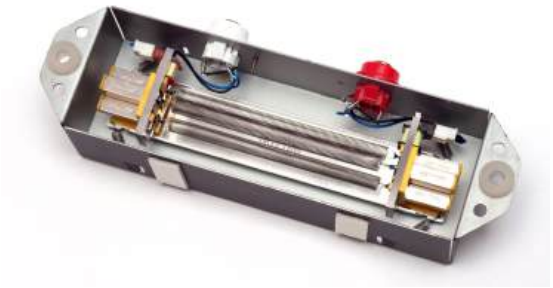

Figure 2.3: Spring Reverb Unit [Ashley Pomeroy CC BY-SA 4.0]

Figure 2.4: Altiverb 8 [Altiverb]

in terms of natural reverberation, many recording artists love their twangy sound on its own terms. Of course, these are the focus of this project, and so a much more detailed accounting of their properties will be given later on.

In the latter half of the twentieth century, the rise of increasingly cheaper and powerful digital ICs led to an increase in interest of digital audio processing techniques, attempts at digital reverberation being no exception. The earliest of these digital reverbs were developed by Manfred Schroeder, and involved the use of networks of allpass filters to maximize "echo density," or the number of simulated reflections ([1]). Although Schroeder's original reverbs don't sound particularly good by today's standards, many modern reverbs still use allpass networks to great effect.

Another common approach to digital reverberation is a convolution based approach. With convolution, an impulse response (IR) is measured for a physical space, and then that IR can be used to reproduce the physical characteristics of the space without needing to do any complex modelling. Although convolution reverbs can be very resource intensive, they can provide very accurate results that are difficult to match via other techniques. One popular plugin implementing this style of reverb is Altiverb, shown in Figure 2.4.

Finally, there is direct physical simulation of reverberation. In this technique, a physical model of a space, or in our case electromechanical device, is developed, and then simulated directly. There are a variety of simulation techniques, such as modal approaches, and FDTD methods. These physical simulations have the obvious

Figure 2.5: Left to right, top to bottom: Arturia Rev SPRING-636, Eventide H9 Spring, GSI TimeVerb-X, Physical Audio Dual Spring Reverb

advantage of realism and accuracy, as they can model intricate systems, but they often prove tricky to run at realtime on consumer hardware, given the potentially large numbers of degrees of freedom that must be simulated at each timestep.

In today's market, there are a huge quantity of reverb devices available, from digital plugins to physical spring tanks. This project looks at one of the oldest physical devices, the spring reverb, and attempts to create a physical simulation that can operate at realtime using a direct FDTD approach. To the author's knowledge, there are not currently any results published or plugins that use this approach succesfully at realtime.

2.2 Spring Reverberation Simulation

The past few decades have seen a plethora of audio plugins simulating classic electromechanical audio effects. These range from tape echo emulations, like the Outer Space by Audio Thing [1] to full echo chamber emulation in Waves' Abbey Road Chambers plugin[2]. Spring reverb's are no exception to this, with popular emulations including the Arturia Rev SPRING-636[3], and u-he's Twangström[4]. Some of these emulations can be seen in Figure 2.5, including the Physical Audio Dual Spring Reverb[5], which was developed by members of the University of Edinburgh's Acoustics and Music Technology group.

While there is no way of knowing for sure what technology each of these plugins use, given their proprietary nature, the available literature suggests that they use a variety techniques, including digital waveguides, convolution based methods, as well as modal simulation. While this project does not use these techniques, it is worth understanding the existing techniques used to simulate spring reverberation.

[1] Outer Space
[2] Abbey Road Chambers
[3] Rev SPRING-636
[4] Twangström
[5] Dual Spring Reverb

2.2.1 Digital Waveguides

Digital waveguide synthesis is an early form of physical modelling synthesis for strings. It operates by simulating the classic travelling wave solution to the 1d wave equation, using a pair of delay lines [2]. This method, while very simple, is surprisingly good, considering its very low runtime cost, and is very popular due to its quality-to-performance ratio.

The technique has also been adapted to simulate springs, as in [3]. This method uses allpass filters alongside the waveguide section, and tunes those filters to emulate the dispersion and frequency response of a given spring reverb. While this approach can give okay results, and is very computationally efficient, it is not particularly grounded in the underlying physics of the helical spring, and as such will not be evaluated further in this project.

2.2.2 Convolution

Another approach, which can be used to model any linear and time invariant (LTI) system, is convolution. With convolution, an impulse response is captured of the system, and then that impulse response is treated as the coefficients for an FIR filter, such that for an input signal $\boldsymbol{x}$, sampled impulse response $\boldsymbol{h}$, and output signal $\boldsymbol{y}$ we have

$$\boldsymbol{y}[n] = \sum_{i=0}^{N} \boldsymbol{h}[i]\boldsymbol{x}[n-i] \tag{2.1}$$

While this formulation can accurately characterize a given spring reverb for which an impulse response is captured (assuming linearity and time invariance), it has some drawbacks. For one, it is very costly to compute the filter for each sample directly. For this reason, the following correspondence between time domain and frequency domain signals is used

$$a * b \Leftrightarrow \mathcal{F}(a)\mathcal{F}(b) \tag{2.2}$$

to transform the problem from a convolution per sample to a multiplication. Another drawback of the approach, regardless of which method is used to compute the convolution, is that it treats the system being measured as a black box, making it difficult to have parametric control over the effect. To get around this, many impulse responses at various settings of the device must be taken, and then interpolated between to get said parametric control. While this works, it can carry a high cost in terms of disk space used for any plugin using the technique. Finally, it provides no insight into the underlying nature of the system, and so extrapolating the effect beyond the bounds of already measurable systems becomes difficult.

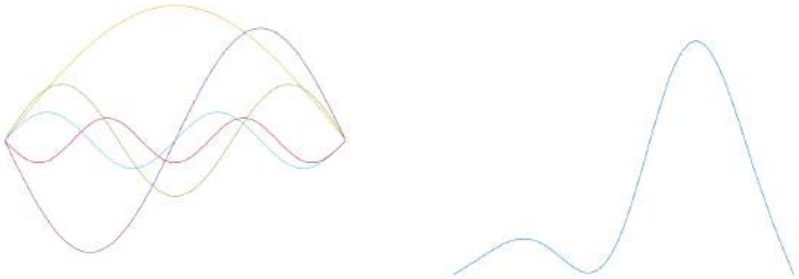

Figure 2.6: Modes and their composition

2.2.3 Modal Simulation

The most similar approach to the one used in this project currently used in commercial plugins are modal schemes, such as the one presented in [4]. Using this method, the system is modelled as many uncoupled ordinary differential equations, simulated separately, and then combined together to get the final result. A simplified example of this can be seen in Figure 2.6. Modal schemes provide an accurate simulation of the physical model, and they are also generally cheaper to compute than the full scheme, given that care is taken to implement the scheme properly, and trim modes that do not impact the final scheme much.

However, despite these benefits, they are also limited in some important ways. For one, they require that the boundary conditions of the system are amenable to the modal approach, which restricts the parameters of the system. In addition, the modal parameters must be computed ahead of time, as doing so at realtime is too costly. Because of this, tables of these parameters must be stored, which can take up large amounts of disk space, and also restricts the scheme to the parameters that have been precalculated.

2.3 Physical Model

Now we will move to the physical model. We will start with a system in 12 variables, developed in [5]. We will then reduce this to a smaller system, in two variables, that retains much of the same characteristics as the 12 variables system. Then, we will discretize this system using standard FDTD techniques, so that we are able to write a computer program that can run the simulation. This simplification from the full model to the FDTD scheme is due to [6].

2.3.1 Nondimensionalized Thin Model

For the model, the following scheme parameters are required:

- α - Spring Pitch (radians)
- r - Minor Radius (m)
- R - Major Radius (m)
- E - Young's modulus (Pa)
- ρ - Density (kg/m^3)
- υ - Poisson's ratio
- L - Unwound length (m)
- A - Cross sectional area (m^2)

Combined, these parameters fully characterize the spring we are simulating. For all of the subsequent models, we will assume standard values for steel: $E = 2\text{e}11$, $\rho = 7850$, and $\upsilon = 0.23$. We must also derive some additional physical characteristics: the shear modulus,

$$G = \frac{E}{2(1-\upsilon)}, \tag{2.3}$$

and the transverse and polar moments of inertia, respectively,

$$I = \frac{\pi r^4}{4} \qquad\qquad I_\phi = 2I. \tag{2.4}$$

The derivation from the full thick 12 variable model is omitted here, but can be seen in [6]. The nondimensionalized thin model we will be working with depends on the following scheme parameters:

$$\mu = \tan\alpha \qquad\qquad b = \frac{EI}{GI_\phi} \qquad\qquad \lambda = \frac{L}{s_0} \tag{2.5}$$

and is defined in terms of s and t, which are scaled by factors

$$s_0 = \frac{R}{\cos^2\alpha} \qquad\qquad t_0 = \frac{R^2\sqrt{\frac{\rho A}{EI}}}{\cos^4\alpha}. \tag{2.6}$$

The differential operators

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1-\partial_{ss} \end{bmatrix} \quad \boldsymbol{D} = \begin{bmatrix} 1 & 0 \\ 0 & b-\partial_{ss} \end{bmatrix} \boldsymbol{R} = \begin{bmatrix} -2\mu & 1-\mu^2+\partial_{ss} \\ 1-\mu^2+\partial_{ss} & 2\mu(1+\partial_{ss}) \end{bmatrix} \tag{2.7}$$

can then be used to construct the scheme

$$\boldsymbol{A}\partial_{tt}\boldsymbol{\xi} = \partial_{ss}\boldsymbol{R}\boldsymbol{D}^{-1}\boldsymbol{R}\boldsymbol{\xi} \tag{2.8}$$

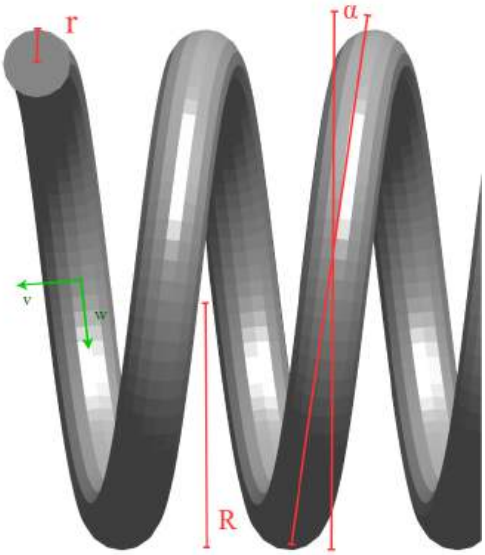

Figure 2.7: Spring Parameters

which depends only on the displacements in two axes,

$$\boldsymbol{\xi} = [v\ w]'. \tag{2.9}$$

These axes, as well as some of the physical parameters of the spring, can be seen in Figure 2.7.

2.3.2 Dispersion Relation

To examine the dispertion relation of this scheme, we can use the ansatz solution $e^{j(\omega t+\beta s)}$, under which the differential operators behave as multiplicative factors

$$\partial_{ss} \coloneqq -\beta^2 \qquad\qquad \partial_{tt} \coloneqq -\omega^2 \tag{2.10}$$

Then we can solve our system after substitution to get our dispersion relation $\omega(\beta)$

$$\omega = \sqrt{\text{eigs}\left(-\beta^2 \hat{\boldsymbol{A}}^{-1}\hat{\boldsymbol{R}}\hat{\boldsymbol{D}}^{-1}\hat{\boldsymbol{R}}\right)} \tag{2.11}$$

with the matrices with circumflexes having ∂_{ss} substituted as in Equation 2.10. This yields two solutions, which can be seen for varying spring pitches in Figure 2.8. Later, it will be useful to compare the dispersion relations of our numerical schemes to this one, as a measure of how close we are to the underlying physical model.

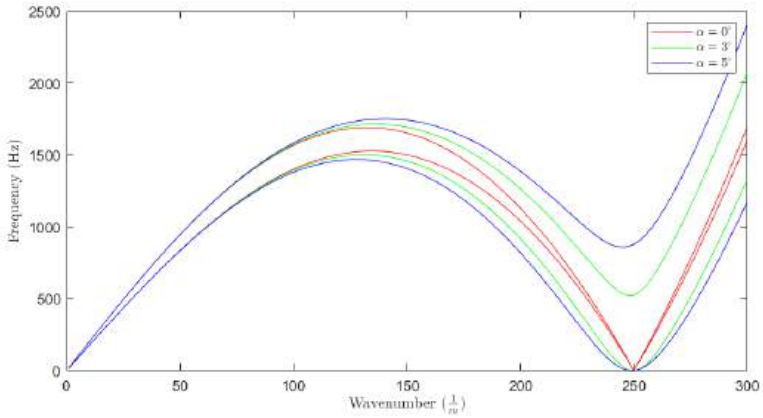

Figure 2.8: Spring model dispersion for varying values of α ($r = 0.2$mm $R = 4$mm $L = 5$m, values are dimensionalized)

2.3.3 Finite Time Difference Scheme

Now that we have a good physical model, we will discretize it, using a standard FDTD based approach. This will give us something to implement and then optimize, which is the main subject of the project.

Grid Operators

To move from the continuous domain to the discrete, we will also need to move from using normal differential operators to ones that operate on our finite grid. Take some grid function, f_n^l, where $n \in \mathbb{Z}$ is the discrete temporal coordinate, and $l \in \mathbb{Z}$ is the discrete spatial coordinate. The grid function samples the continuous function, $g(s,t)$, at spatial and temporal steps, h and k, such that

$$f_n^l = g(hn, kt). \tag{2.12}$$

Then, our temporal differential operators are

$$\delta_{t+} f_l^n = \frac{1}{k}(f_l^{n+1} + f_l^n) \qquad \delta_{t-} f_l^n = \frac{1}{k}(f_l^n - f_l^{n-1}), \tag{2.13}$$

and our spatial differential operators are

$$\delta_{s+} f_l^n = \frac{1}{h}(f_{l+1}^n + f_l^n) \qquad \delta_{s-} f_l^n = \frac{1}{h}(f_l^n - f_{l-1}^n). \tag{2.14}$$

These can be composed to get second order differential operators such as

$$\delta_{ss} = \delta_{s+}\delta_{s-} \qquad \delta_{tt} = \delta_{t+}\delta_{t-}. \tag{2.15}$$

2.3.4 Scheme Discretization

Now, the scheme can be discretized using the operators we have defined. This is simple enough, substituing ∂_{ss} and ∂_{tt} for their discrete equivalents, δ_{ss} and δ_{tt}, respectively. We now have

$$\boldsymbol{A}_d \delta_{tt} \boldsymbol{\xi} = \delta_{ss} \boldsymbol{R}_d \boldsymbol{D}_d^{-1} \boldsymbol{R}_d \boldsymbol{\xi} \tag{2.16}$$

where the differential operators subscripted with d are their discrete equivalents

$$\boldsymbol{A}_d = \begin{bmatrix} 1 & 0 \\ 0 & 1 - \delta_{ss} \end{bmatrix} \quad \boldsymbol{D}_d = \begin{bmatrix} 1 & 0 \\ 0 & b - \delta_{ss} \end{bmatrix} \boldsymbol{R}_d = \begin{bmatrix} -2\mu & 1 - \mu^2 + \delta_{ss} \\ 1 - \mu^2 + \delta_{ss} & 2\mu(1 + \delta_{ss}) \end{bmatrix}. \tag{2.17}$$

Now we must choose appropriate grid spacing. k is easy enough, simply $\frac{1}{F_s}$, where F_s is our sampling rate. For h, things are more complicated. As noted in [6], we cannot get an explicit formulation in terms of k, so we must attempt to get as close as possible to the stability condition of the system, in order to limit numerical dispersion. The system is stable when

$$eig\left(\frac{k^2}{h^2}\sin^2\left(\frac{\beta h}{2}\right)\hat{\boldsymbol{A}}_d^{-1}\hat{\boldsymbol{R}}_d\hat{\boldsymbol{D}}_d^{-1}\hat{\boldsymbol{R}}_d\right) \leq 1 \tag{2.18}$$

for all wavenumbers $0 \leq \beta \leq \frac{\pi}{h}$ where the matrices with circumflexes are identical to their previous formulations, substituting

$$\delta_{ss} \coloneqq -\frac{4\sin^2(\beta h/2)}{h^2}. \tag{2.19}$$

This factor comes from using the ansatz $e^{j(lh\beta + nk\omega)}$ as the solution to our numerical scheme, where l and n are integers, and β and ω are the wavenumber and angular frequency.

2.3.5 Numerical Dispersion

To find our new dispersion relations for the numerical scheme, we employ Von Neumann analysis as with the stability analysis, taking as an ansatz $e^{j(lh\beta + nk\omega)}$. Again, our differential operators are substituted as multiplicative factors,

$$\delta_{ss} \coloneqq -\frac{4\sin^2(\beta h/2)}{h^2} \qquad \delta_{tt} \coloneqq -\frac{4\sin^2(\omega k/2)}{k^2}. \tag{2.20}$$

We find the following dispersion relation $\omega(\beta)$,

$$\omega = \frac{2}{k}\sqrt{\text{eigs}\left(\frac{k^2}{h^2} sin(\beta h/2)^2 \boldsymbol{A}_d^{-1}\boldsymbol{R}_d\boldsymbol{D}_d^{-1}\boldsymbol{R}_d\right)}. \tag{2.21}$$

This relation for various choices of grid spacing can be seen in Figure 2.9, motivating

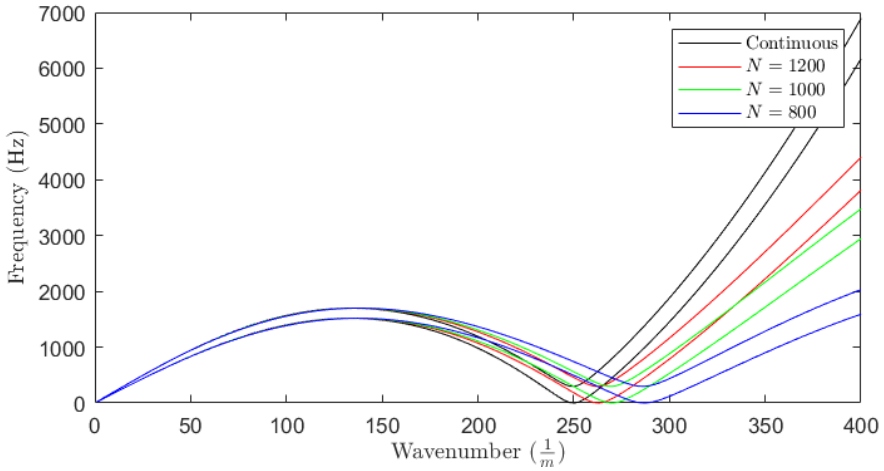

Figure 2.9: Numerical Dispersion for various choices of N ($F_s = 44.1$kHz, $\alpha = 1.7°$, $r = 0.2$mm, $R = 4$mm, $L = 5$m values are nondimensionalized)

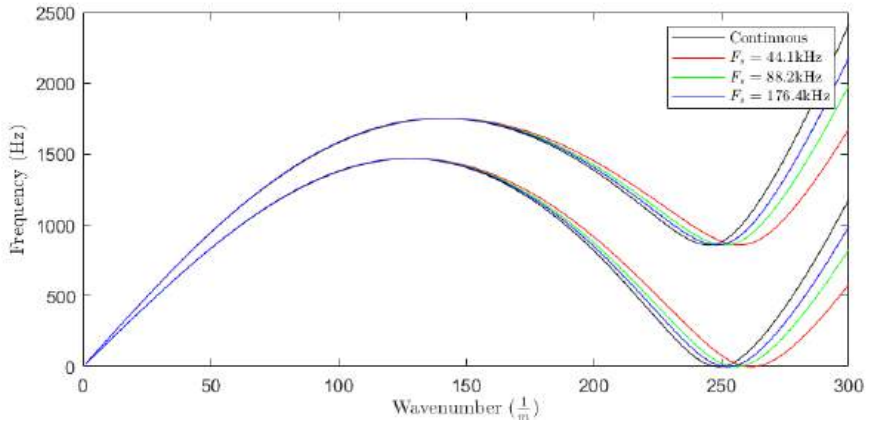

Figure 2.10: Numerical dispersion for various choices of sampling rate ($\alpha = 5°$, $r = 0.2$mm, $R = 4$mm, $L = 5$m values are nondimensionalized)

our desire to have the grid spacing be as small as possible, without violating stability. The numerical dispersion can also be seen for various choices of sampling rate, in Figure 2.10, showing that we would prefer to oversample the model for improved accuracy, if possible.

2.4 Optimization Techniques for CPUs

There are many challenges in optimizing computer programs to run quickly on modern processors. This is because, unlike older processors, CPUs are no longer scalar data processors, and provide many mechanisms to accelerate computation, beyond mere algorithmic design, although that is crucial as well. This section shall enumerate some

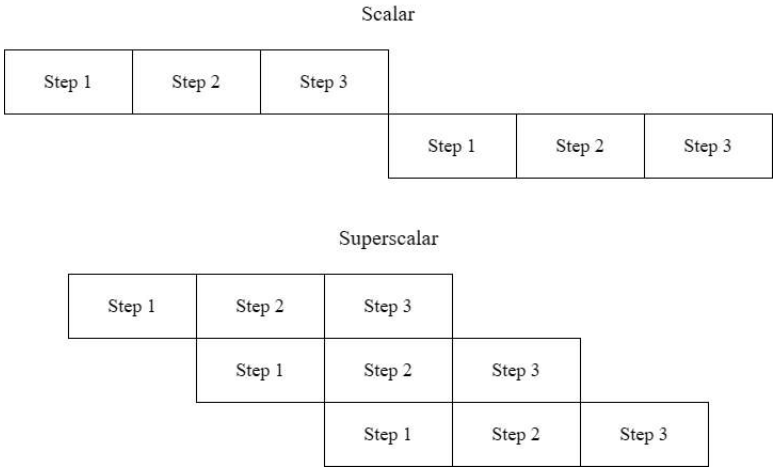

Figure 2.11: Scalar and Superscalar architectures

of the methods we must explore if we wish to fully make use of the power of modern hardware.

2.4.1 Superscalar Processing

Traditionally, CPUs were scalar processors, meaning that they executed one operation at a time, in order. On modern devices, however, instructions are "pipelined," such that while one instruction is executing, the CPU can begin executing subsequent instructions, and by doing so dramatically speed up throughput of instructions ([7]). An simplified example of this can be seen in Figure 2.11. In the example labeled "Scalar," the processor must wait for each instruction to execute before proceeding, while in the "Superscalar" example, multiple instructions are processed at once.

Taking advantage of this feature is not always simple. Often, the input of one instruction is the output of the previous one. This case is called a "dependency chain." Because breaking dependency chains is an essential part of optimizing modern software, modern optimizing compilers such as LLVM[6] attempt to automatically break these dependency chains, where possible. However, in many cases, the compiler will be unable to do so, and human intervention is required.

2.4.2 Vector Processing Units

Another feature of modern processors that allows for increased data throughput is the vector processing unit. These units allow programs to load specialized registers on the CPU with many separate pieces of data, and then execute an operation on all of them simultaneously [7]. These special instructions are referred to as simultaneous data, multiple execution (SIMD) instructions. This has the potential for large speedups in

[6] LLVM

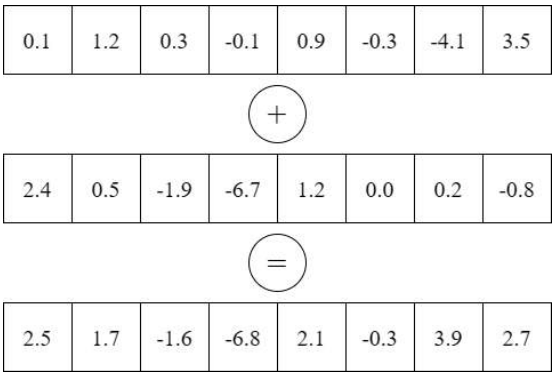

Figure 2.12: SIMD Addition Instruction

data throughput, as it is a direct multiplier on the number of operations we can do per clock cycle. On modern CPUs that support the AVX instruction set (since the Sandy Bridge architecture for Intel CPUs [8]), SIMD registers are 256 bits wide, which allows them to store 8 single precision floating point numbers, or 4 double precision floating point numbers.

These instructions, while having the potential for massive speedups, can present some difficult implementation challenges. Optimizing compilers will attempt to auto-vectorize routines, and while this is often successful, they may do so sub-optimally, or they may fail to vectorize all together. Often, with critical sections of code, it is required that the programmer either write assembly code directly, or use so-called compiler "intrinsics" that allow the programmer to specify the exact hardware instructions to be carried out, albeit while still looking like regular code. These intrinsics can be difficult to use however, with complex mneumonics such as `_mm_bcstnesh_ps`.

2.4.3 Caching Effects

The final factor we shall consider with regards to optimization (although this is by no means an exhaustive list of the optimization techniques a programmer must be aware of) is that of the caching effects. One of the most expensive operations a modern CPU can execute is reading from the computers random access memory (RAM) [7]. To limit the amount that the processor must do this, several levels of memory caching are provided, where the processor can store memory it has fetched recently in a closer location, such that subsequent accesses will be much faster.

The actual mechanics of CPU caching are complex, and vary between CPU microarchitectures. For the purposes of this report, the important takeaway is that maintaing locality of access within a program is very important for optimal performance. That is, we want to group our operations to and from memory in terms of how close they are to one another, where possible. This is as opposed to randomly

scattering our memory accesses, which will lead to many unnecessary trips out to main memory and back.

2.5 Zig Language

For the implementation of the program, the Zig programming language was chosen. This is due to the fact that it provides significantly nicer ergonomics than C and C++, while still providing low level control over memory allocation and interaction with the underlying operating system and hardware. It also has very nice interoperation with C libraries, so using libraries is more or less seamless. One important ergonomic consideration is the use of vector processors. As mentioned previously, traditionally working with a CPUs vector processing units requires tinkering with complex compiler intrinsics. In Zig, however, vectors units have first class support, and can be declared via the syntax `@Vector(<type>, <number of slots>)`. For example, the following code loads two vectors, adds them, and checks the result.

```
const a = @Vector(f64, 4) { 0.0, 1.1, 2.2, 3.3 };
const b = @Vector(f64, 4) { 4.4, 5.5, 6.6, 7.7 };
const c = a + b;
assert(@reduce(.And, c == { 4.4, 6.6, 8.8, 11.0 }));
```

Chapter 3

Methodology

3.1 Prototyping in MATLAB

Before the scheme could be optimized, there first needed to be a working version of it to check results against. Instead of jumping right into Zig, a naive prototype was developed MATLAB. MATLAB was chosen because its syntax mirrors closely that of the math, and it has superior debugging tools to those available for Zig. To get a version of the scheme up and running, the scheme was inputted more or less directly as written in the maths, and the system solves were using MATLAB's backslash operator. Of course, the scheme ran very slowly, but it provided a solid foundation on which to build the rest of the project. MATLAB was also used to produce many of the diagrams in this project, as well as determining things like numerical stability conditions, and numerical dispersion.

3.2 Program Architecture

As the primary aim of this project was to improve the runtime performance of the FDTD spring scheme. As such, the program was written with this aim in mind, to make prototyping each technique as simple as possible, and making it easy to test the software. To this end, the software is a single file of Zig code, with each separate solver consisting of a single `struct`, an initialization procedure, and a solving procedure. The program takes a hardcoded wave file as input, and produces a single wave file as ouptut. Notably, it is **not** a fleshed out plugin, as dealing with the intricacies of libraries such as JUCE[1] or the VST3 SDK[2] was ancillary to this project.

The program executes in the following manner

- Read the input from disk into memory as a buffer of samples.

[1] JUCE
[2] VST3 SDK

- Initialize temporary buffers and solvers
- Process the input in a loop, one sample at a time, using the FDTD model
- Output the resulting audio to a wave file

This simple procedure, taking place mostly in a single function, makes it very easy to modify the program quickly, and insert instrumentation into the program to get quick and accurate readings of which sections were consuming the most time. To test all of the different permutations of input parameters, sampling rates, and solver types, compiler flags are exposed to the user.

3.3 Benchmarking

Before attempting to optimize the program, it was important to collect accurate timing data for the program. This is notoriously fraught, particularly when working with real time software, as the process of collecting timing data can change the performance characteristics of the program we are timing. This is particularly the case when calling operating system provided functionality, such as `QueryPerformanceCounters` on Windows, due to the fact that we have no control over what instructions these subroutines perform. For this reason, a small piece of benchmarking software was developed, that first calibrates itself using the operating system timers, and then makes calls directly to the CPU for information. This software was informed by the profiling tool developed as part of Casey Muratori's "Computer Enhance" lecture series, adapted significantly ([9]).

To do this, the benchmarking software uses the `rdtsc` (**r**ea**d** **t**imes**t**amp **c**ounter) instruction provided by x86-64 processors [10]. This instruction provides a monotonically increasingly integer clock, with each interval representing roughly[3] one clock cycle. Given that `rdtsc` is a single processor instruction, along with a few more instructions to store the timestamp, the collection of timestamps is very quick. While this is indeed a very lightweight timing mechanism, compared to other more common approaches, it still adds some overhead. To ensure that the instrumentation does not cloud the results, a flag is provided to turn off the instrumentation, and only time the overall runtime of the program. This can then be compared to the instrumented timing, and a rough heuristic of timer overhead can be determined.

An important consideration here is that of repetition testing. There are many factors outside of the programmers control when it comes to benchmarking software. The operating system may at any time pause program execution to let another program run, for instance. To control for this, all of the experiments run for this project were

[3] On modern processors this is no longer the case, due to features such as turboboost. However, for our purposes, the important bit is that the instruction provides **consistent** results, regardless of the actual momentary clock speed of a given core.

run a total of 10 times, and then averaged. One additional consideration was that because the program has many different options for solvers, it was undesirable to have the control flow scattered through the program to account for differences between solver implementations to affect the runtime of the programs. To get around this issue, the parameters that changed between tests were pulled out into compiler flags, and then the program was recompiled for each run of the program. Luckily, this did not add much additional overhead to the runtime of the test suites due to Zig's rather fast compile times.

3.4 Linear System Solvers

At the heart of this project is the two linear system solves that occur sample, and take the majority of the runtime of the scheme. Quite a few methods were tried, which will be detailed here. While the results are not discussed in detail in this section, the results of each method heavily informed the next steps taken in the project, and so relative results will be discussed briefly.

As mentioned previously, both of the systems we wish to solve are of the form

$$\begin{bmatrix} 1 & & & & & & & \\ & \ddots & & 0 & & & & \\ & & \ddots & & & 0 & & \\ & 0 & & 1 & & & & \\ & & & & a & b & & 0 \\ & & 0 & & b & \ddots & \ddots & \\ & & & & & \ddots & \ddots & b \\ & & & & 0 & & b & a \end{bmatrix}$$

Because upper left corner of the matrix is simply the identity, and the upper right and lower left corners are 0, we may focus our attention exclusively on the lower right corner of the matrix. This leaves us with a matrix of the form

$$\begin{bmatrix} a & b & & 0 \\ b & \ddots & \ddots & \\ & \ddots & \ddots & b \\ 0 & & b & a \end{bmatrix}$$

to solve. This matrix can be said to be tridiagonal, symmetric, toeplitz, and positive definite. There are a number of potential methods we can use to solve these systems, all of which have their own trade-offs to consider.

3.4.1 LAPACK Solvers

Before building any custom solvers, an off-the-shelf solver from the OpenBLAS[4] implementation of the LAPACK[5] linear algebra library was used. OpenBLAS was chosen as it contains highly tuned solvers for modern hardware. Intel's Math Kernel Library[6] was also considered, but this project was conducted using an AMD processor, and the MKL implementations are known to be artificially crippeled on AMD hardware.

The `sgttrs` and `spttrs` solvers were used, along with their factorization routines `sgttrf` and `spttrf`, which operate on tridiagonal and symmetric positive-definite systems, respectively. Unfortunately, LAPACK does not provide any solvers for symmetric positive definite systems that are also toeplitz, and so an array of each of our coefficients must be provided, which requires a fair bit of redundant memory usage. As expected, the solver built for the more specific case of matrix (symmetric positive definite) was faster, but not by much. There was clearly a need for linear system solvers that are faster than those provided by libraries such as OpenBLAS.

3.4.2 Iterative Methods

The first method attempted was to use an iterative method. Iterative methods operate by beginning with some initial guess for the next state, x_0. Then by some procedure $f(x)$, depending on the method, a "next guess" is determined

$$x^{(k+1)} = f(x^{(k)})$$

This is repeated until some condition is reached, in our case we will just check that the distance between the previous guess and current guess is smaller than some ϵ

$$||x^{(k+1)} - x^{(k)}|| < \epsilon$$

It is worth noting that depending on the specific method used and system being solved for, the solver will not converge. However, the systems we deal with are fairly well behaved, and so we will not need to concern ourselves with this issue.

The method first used was Jacobi iteration, which is a very simple method, guaranteed to converge for diagonally dominant systems such as ours [11]. For a linear system $\boldsymbol{Ax} = \boldsymbol{b}$, where

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{n,1} \\ \vdots & \ddots & \vdots \\ a_{1,n} & \cdots & a_{n,n} \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \tag{3.1}$$

[4]OpenBLAS
[5]LAPACK
[6]Intel oneMKL

Each iteration of the state is then computed as

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^{(k)} \right)$$

For our case of a tridiagonal matrix this reduces to

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - (a_{i,j-1} \cdot x_{j-1}^{(k)} + a_{i,j+1} \cdot x_{j+1}^{(k)}) \right)$$

This approach provides several advantages. For one, it is very simple to implement. Another is that the compiler is easily able to vectorize this operation, and there are not many serial dependencies, so we can take full advantage of the hardware. Unfortunately, having to perform the iterations many times proves unacceptably slow for our purposes.

There are many other iterative solvers for these types of systems. One such method is the conjugate gradient method, which is commonly used in large computational fluid dynamics (CFD) problems. Unfortunately, this technique is rather complex to implement, and so direct methods were explored instead.

3.4.3 Thomas Algorithm

The next method tried after Jacobi Iteration was the Thomas algorithm, which is a modified version of Gaussian elimination for tridiagonal systems that runs in linear time, as opposed to standard Gaussian elimination, which runs in cubic time [12]. Similarly to Jacobi iteration, the algorithm itself is simple. It is a 2 stage process, where first our system is changed from a tridiagonal system, to an upper diagonal system $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{b}'$, where

$$\boldsymbol{U} = \begin{bmatrix} 1 & \mu_1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \mu_n \\ 0 & \cdots & \cdots & 0 & 1 \end{bmatrix} \qquad \boldsymbol{b}' = \begin{bmatrix} b_1' \\ \vdots \\ b_n' \end{bmatrix} \tag{3.2}$$

$$\mu_1 = \frac{a_{2,1}}{a_{1,1}} \qquad \mu_i = \frac{a_{i+1,i}}{a_{i,i} - a_{i-1,i} \cdot \mu_{i-1}} \tag{3.3}$$

$$b_1' = \frac{b_1}{a_{1,1}} \qquad b_i' = \frac{b_i - a_{i-1,i} \cdot b_{i-1}'}{a_{i,i} - a_{i-1,i} \cdot \mu_{i-1}} \tag{3.4}$$

Then, $\boldsymbol{x}$ can be solved for

$$x_n = b'_n \qquad\qquad x_i = b'_i - \mu_i x_{i+1} \tag{3.5}$$

This direct method is both simple, and very fast. However, there is one major flaw for our purposes: it is an entirely serial algorithm. This can be seen both in the first step, which works its way down the diagonal, using the previous result, and the second step, which works its way back up the diagonal, also using the previous step. This defeats multiple potential optimizations, both vector optimizations, and instruction pipelining.

Before moving to other methods of solving the system, some optimizations of this scheme are possible. While the Thomas algorithm alone will not get us to realtime, these optimizations are still valuable, as this solver will still figure into the cyclic reduction method. It should be noted that the values of $\boldsymbol{\mu}$ in 3.3 can be computed ahead of time, and do not require recalculation unless the scheme parameters change. In addition, the denominator of the update in 3.4 can be precomputed. In fact, because of the fact that floating point multiplication is quicker than a floating point division on x86 hardware, we would rather precompute the inverse, and then multiply by it, instead of having to divide each sample.

3.4.4 Cyclic Reduction

Luckily, there exists in the literature a technique for getting around this limitation, and parallelizing our solver referred to as cyclic reduction, developed by Gene Golub in the 1960s ([13]). For the implementation, the accounting given in [14] was used, which is summarized nicely in [15]. In this method, we rearrange our original tridiagonal system (the need for labelling the corner elements will become apparent)

$$\begin{bmatrix} a_1 & b & & & \\ b & a & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & a & b \\ & & & b & a_n \end{bmatrix} \tag{3.6}$$

into the block tridiagonal system

$$\begin{bmatrix}
a'_1 & b' & & & & & & & & \\
b' & a' & \ddots & & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & & \\
 & & \ddots & a' & b' & & & & & \\
 & & & b' & a'_n & & & & & \\
 & & & & & c'_1 & d' & & & \\
 & & & & & d' & c' & \ddots & & \\
 & & & & & & \ddots & \ddots & \ddots & \\
 & & & & & & & \ddots & c' & d' \\
 & & & & & & & & d' & c'_n
\end{bmatrix}, \tag{3.7}$$

where

$$b'_i = d'_i = -\frac{b^2}{a} \tag{3.8}$$

$$a'_1 = a_1 - \frac{b^2}{a} \tag{3.9}$$

$$c'_1 = a - \frac{b^2}{a_1} - \frac{b^2}{a} \tag{3.10}$$

$$a' = c' = a - \frac{2b^2}{a} \tag{3.11}$$

$$a'_n = a - \frac{b^2}{a} - \frac{b^2}{a_n} \tag{3.12}$$

$$c'_n = a - \frac{b^2}{a}. \tag{3.13}$$

The state, $\boldsymbol{y} = [y_1, \cdots, y_n]'$, must also be rearranged into $\boldsymbol{y}' = [y'_1, \cdots, y'_n]$, where

$$\alpha = \frac{b}{a} \tag{3.14}$$

$$y'_1 = y_1 + \alpha y_2 \tag{3.15}$$

$$y'_i = y_{2i} + \alpha(y_{2i-1} + y_{2i+1}),\ 2 \le i \le n/2 \tag{3.16}$$

$$y'_{n/2} = y_{n-1} + \alpha(y_{n-2} + y_n) \tag{3.17}$$

$$y'_{n/2+1} = y_2 + \alpha y_0 + y_3 \tag{3.18}$$

$$y'_{n/2+i} = y_{2i+1} + \alpha(y_{2i} + y_{2i+2}),\ 2 \le i \le n/2 \tag{3.19}$$

$$y'_n = y_n + \alpha y_{n-1}. \tag{3.20}$$

Once these transformations are carried out, the resulting system can be treated as two independent tridiagonal linear systems to be solved, and they can be solved using

the Thomas algorithm, working two at a time for both steps of the algorithm. This is not a direct two times speedup, as the transformation of the state required (and then the subsequent transformation of the output back to the previous form) is not an insignificant calculation. This transformation step can then be repeated as many times as the system can be divided nicely in two, although in this case we only need to divide thrice to parallelize to the full 8 SIMD lanes available on modern x86 machines.

3.4.5 FFT Based Methods

One additional method that was attempted, but not measured fully, was a fast fourier transform (FFT) based method. The toeplitz triadiagonal systems being worked with,

$$\begin{bmatrix} a & b & & 0 \\ b & \ddots & \ddots & \\ & \ddots & \ddots & b \\ 0 & & b & a \end{bmatrix},$$

are very similar to circulant matrices of the form

$$\begin{bmatrix} a & b & 0 & & b \\ b & \ddots & \ddots & \ddots & \\ 0 & \ddots & \ddots & \ddots & 0 \\ & \ddots & \ddots & \ddots & b \\ b & & 0 & b & a \end{bmatrix}.$$

Because of this, instead of treating our system as a linear system to solve, $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, we can instead treat it as a deconvolution of the form $\boldsymbol{a} * \boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{a}$ is a kernel of the form

$$\boldsymbol{a} = [a, b, 0, \cdots, 0, b]'$$

Due to the well known correspondence between time domain convolution and frequency domain multiplication, we can reframe this problem as

$$\boldsymbol{x} = \mathcal{F}^{-1}\left(\frac{\mathcal{F}(\boldsymbol{b})}{\mathcal{F}(\boldsymbol{a})}\right).$$

Unlike the previous methods surveyed, which work for any tridiagonal system, this method makes use of the toeplitz nature of our system. Of course, it is not quite accurate, given that it requires a perturbation of the initial system to a circular matrix. In addition, it becomes difficult to change the boundary conditions of the system, which is one of the primary reasons to choose the FDTD approach in the first place.

Chapter 4

Results

In this section, we will examine the results from the various tests ran for each of the various solvers. As mentioned previously, the benchmarking system was custom, and the full output of a run of these tests can be seen in Appendix A. For the test suite I ran, the following parameters remained fixed between runs:

- $\alpha = 1.7°$
- $r = 0.2\text{mm}$
- $R = 4\text{mm}$
- $E = 200\text{GPa}$
- $\rho = 7850\text{kg/m}^3$
- $\upsilon = 0.29$

The parameters that were varied between runs were the sampling rate, F_s, the number of grid points, N[1], and the unwound spring length, L. The spring length was varied so that N would need to change, thus simulating various different configurations of the spring a user might want. The results of these tests[2] can be seen in Figures 4.1, 4.2, 4.3.

There are some trends that can be remarked upon. Firstly, it seems that 4x oversampling is too computationally expensive, even for the smallest spring and the fastest solver. This was true across the solvers, although it was more severe among the slower solvers of course. The second is that, interestingly, the LAPACK solver is quite slow, losing out to all but the very simple iterative solver. This was unexpected, as the solver is written by engineers who theoretically have the most knowledge of the underlying hardware, and are very experienced at writing this kind of numerical

[1] Chosen to be close to the maximum allowed by stability conditions.

[2] All tests were carried out on my laptop, which has an AMD Ryzen 9 5900HS, running at 4.3 GHz.

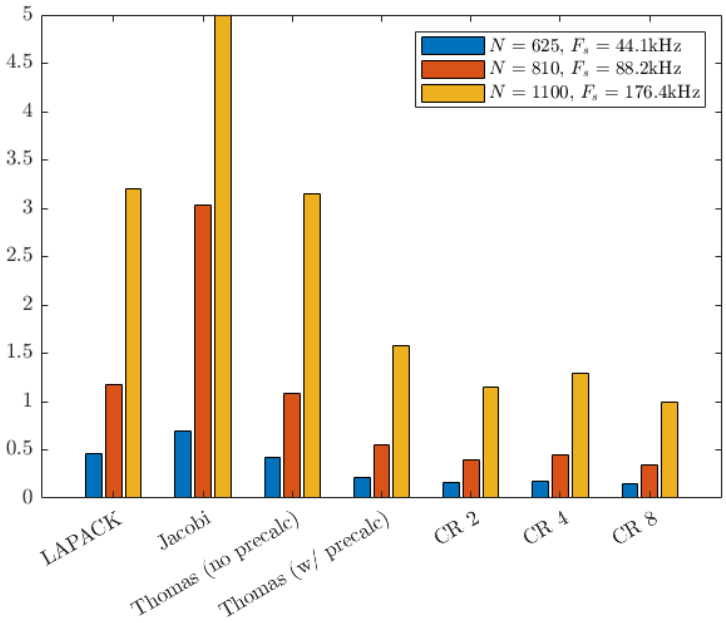

Figure 4.1: Solver results for $L = 2.5$

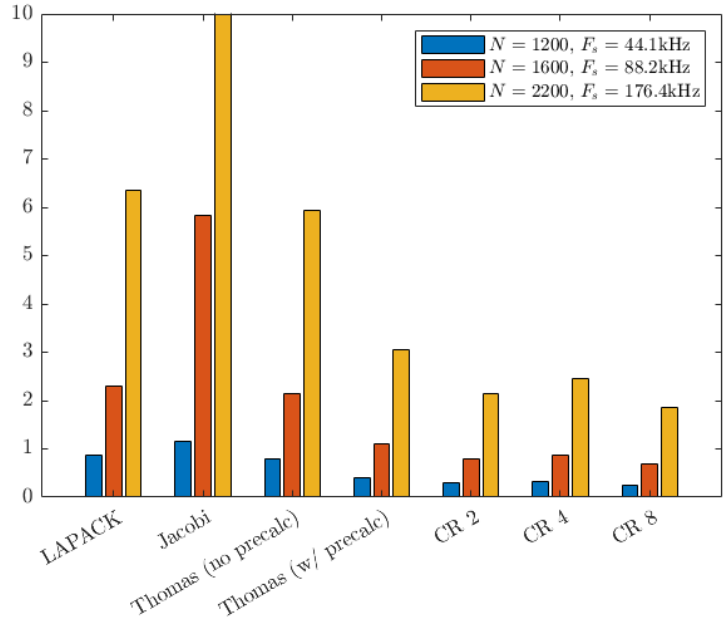

Figure 4.2: Solver results for $L = 5$

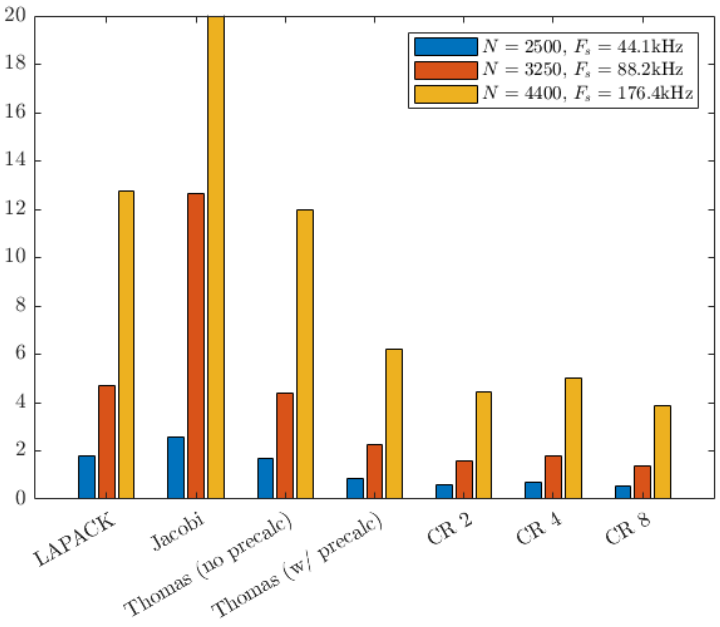

Figure 4.3: Solver results for $L = 10$

calculation code. As mentioned previously, it is likely that this is due to the fact that more simplifying assumptions can be made in the case of the custom solvers than can be made for the LAPACK solver, such as not requiring any pivoting.

Moving on to the custom solvers, we will start with the first one implemented, and also the slowest, by far, Jacobi iteration. In each of the Figures showing the results, the 4x oversampled solver for Jacobi iteration has been truncated as including them does not leave much space for the others[3]. Clearly, this solver is unacceptable for practical use. There are other iterative solvers to explore, but they are quite complex, and seem mostly to be used for much larger CFD problems in massively parallel environments, which is why I pursued direct methods instead.

Both of the Thomas algorithm approaches were fairly fast, at least for the non-oversampled cases. However, the solver with precomputed factors was massively faster, by nearly a factor of two, in all cases. While it is true that each time the parameters for the spring change, the factors must be recomputed, that recomputation would become not so significant when amortized over, say, a 512 sample block, as is common in many DAWs.

Finally, the cyclic reduction solvers were by far the most successful, achieving realtime performance in many cases. One interesting point about the results is that the cyclic reduction solver utilizing four lanes is actually slower than the one using two. This is due to the fact that, as mentioned, cyclic reduction introduces a permutation of the state, which is fairly costly, and in the case of the four lane cyclic reduction

[3]The times are 17, 35, and 70 seconds, for 1, 2, and 4 times oversampling, respectively.

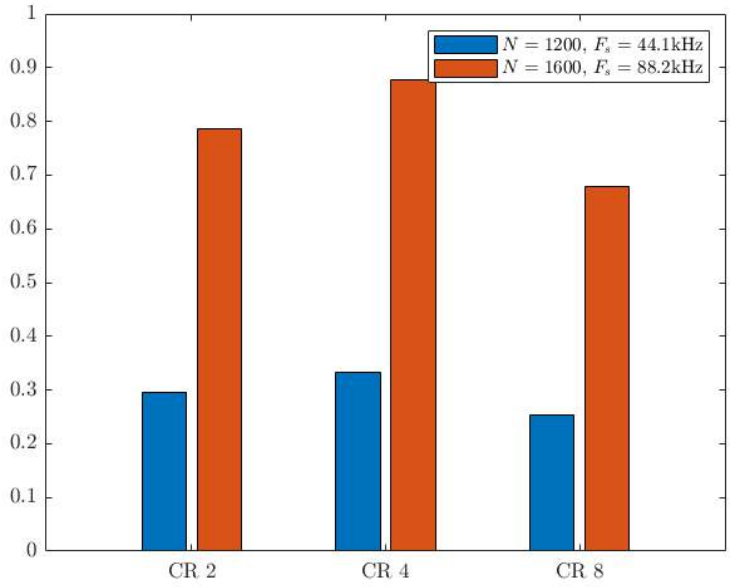

Figure 4.4: Cyclic reduction solver results for $L = 5$

solver, its cost outweighs the gains made up by being able to solve the systems in parallel. This is a major area where the program could potentially be made faster, as the permutation operation is currently serial, but not inherently so. As the cyclic reduction solvers also precompute the system matrix factors ahead of time, the previous comment about amortizing recomputation over a block applies. A more detailed graph of the real-time cyclic reduction solvers (for the $L = 5$ system) can be seen in Figure 4.4.

4.0.1 Real-world implications

For usage in audio plugins, CPU budgets for individual plugins are generally much lower than 100% of a full cores throughput. This is because the host DAW can potentially be running dozens of plugins for a single session, meaning that even if the DAW can make good use of multithreading, only a fraction of the power of each core may be utilized per plugin. Even giving ourselves a generous limit of 20%, only the non-oversampled cyclic reduction solvers for the 2.5m spring are acceptable here. This means that we would need to heavily restrict the size of the spring for a real world plugin, and something like a dual spring system would be out of the question. As metioned, there is definitely some play in the systems still, with the permutation of the state not being vectorized, and while the matrix multiplies are vectorized, the code generation may not be optimal, and require hand tuning to get faster. In this case, it may be worth writing

custom assembler routines, but unfortunately that was outside the scope of this project, although it would need to be explored for a commercial plugin to be developed.

Chapter 5

Conclusions

Although the cyclic reduction model developed is still too slow for use in a commercial plugin, it is the authors belief that with some additional optimization, particularly in the sections of the program that were not treated with as much detail in this report, the cyclic reduction solvers could be a viable option for a plugin implementation. This project gave the author significant insight into the nature of spring reverbs, the physics that underlie their operation, and the simulation of these devices. In addition, the author learned a lot about the benchmarking and optimization of high performance software, and the optimization techniques necessary for physical simulations of this nature.

Despite the successes of the project, there are many avenues that could be explored given more time. Besides looking for more speedups, the flexibility of the FDTD scheme used in this paper opens up many possibilities in terms of interaction that are much trickier to implement for other methods. For example, one much beloved feature of physical spring reverberation devices is the ability to kick the device, or open up the tank and pluck the springs. This would be essentially trivial with the scheme presented here. Additionally, one part of the model not given much attention in this paper is that of boundary conditions and input/output from the system. This was intentional, as they do not have much of an impact on performance, but with this model any number of schemes could be used to more closely model the behaviour of a spring reverb. All in all, this was a very rewarding project to carry out, and much was learned in the process of completing it.

Appendix A

Example Timing Results

Below is the output produced by a run of the program, using the settings described for a 2.5m spring, at various sampling rates and grid spacings.

```
-----------
Fs=44100
N=1200
L=2.5
-----------

Solver: LAPACK
Total Time: 0.459 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  441000 |       24845176 |  0.007543 |  1.644% | 98.630%
 Rd Mult            ||  882000 |       47108944 |  0.014302 |  3.116% |  3.116%
 Dd Solve           ||  441000 |      700676200 |  0.212716 | 46.352% | 46.352%
 Dss Mult           ||  441000 |       17428540 |  0.005291 |  1.153% |  1.153%
 Ad Solve           ||  441000 |      700878000 |  0.212777 | 46.365% | 46.365%
 Update             ||  441000 |       15658223 |  0.004754 |  1.036% |  1.036%
 Circulate Vectors  ||  441000 |        1220826 |  0.000371 |  0.081% |  0.081%

Solver: Jacobi Iteration
Total Time: 0.698 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  441000 |       28740768 |  0.008725 |  1.250% | 99.090%
 Rd Mult            ||  882000 |       48314044 |  0.014667 |  2.101% |  2.101%
 Dd Solve           ||  441000 |      858368260 |  0.260587 | 37.319% | 37.319%
```

```
 Dss Mult           ||  441000 |       18826566 |  0.005715 |  0.819% |  0.819%
 Ad Solve           ||  441000 |     1324882000 |  0.402214 | 57.602% | 57.602%
 Update             ||  441000 |       15784908 |  0.004792 |  0.686% |  0.686%
 Circulate Vectors  ||  441000 |        1246926 |  0.000379 |  0.054% |  0.054%

Solver: Thomas, no precomputation
Total Time: 0.420 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  441000 |       28245136 |  0.008575 |  2.040% | 98.270%
 Rd Mult            ||  882000 |       50851852 |  0.015437 |  3.673% |  3.673%
 Dd Solve           ||  441000 |      631576900 |  0.191731 | 45.624% | 45.624%
 Dss Mult           ||  441000 |       17460528 |  0.005301 |  1.261% |  1.261%
 Ad Solve           ||  441000 |      632240400 |  0.191933 | 45.672% | 45.672%
 Update             ||  441000 |       18233782 |  0.005535 |  1.317% |  1.317%
 Circulate Vectors  ||  441000 |        1220616 |  0.000371 |  0.088% |  0.088%

Solver: Thomas, with precomputation
Total Time: 0.215 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  441000 |       26025110 |  0.007901 |  3.666% | 97.082%
 Rd Mult            ||  882000 |       51073920 |  0.015505 |  7.195% |  7.195%
 Dd Solve           ||  441000 |      297533150 |  0.090328 | 41.915% | 41.915%
 Dss Mult           ||  441000 |       16795076 |  0.005099 |  2.366% |  2.366%
 Ad Solve           ||  441000 |      297703070 |  0.090379 | 41.939% | 41.939%
 Update             ||  441000 |       14959642 |  0.004542 |  2.107% |  2.107%
 Circulate Vectors  ||  441000 |        1233341 |  0.000374 |  0.174% |  0.174%

Solver: Cyclic Reduction with 2 lanes
Total Time: 0.160 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  441000 |       28214908 |  0.008566 |  5.351% | 96.038%
 Rd Mult            ||  882000 |       47570596 |  0.014442 |  9.022% |  9.022%
 Dd Solve           ||  441000 |      206556430 |  0.062707 | 39.176% | 39.176%
 Dss Mult           ||  441000 |       18991112 |  0.005765 |  3.602% |  3.602%
 Ad Solve           ||  441000 |      205025440 |  0.062243 | 38.886% | 38.886%
 Update             ||  441000 |       15788850 |  0.004793 |  2.995% |  2.995%
 Circulate Vectors  ||  441000 |        1237953 |  0.000376 |  0.235% |  0.235%
```

```
Solver: Cyclic Reduction with 4 lanes
Total Time: 0.177 seconds
 Block Name          ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  441000 |       28010960 |  0.008504 |  4.798% | 96.483%
 Rd Mult             ||  882000 |       47607984 |  0.014453 |  8.154% |  8.154%
 Dd Solve            ||  441000 |      234394350 |  0.071158 | 40.145% | 40.145%
 Dss Mult            ||  441000 |       18942188 |  0.005750 |  3.244% |  3.244%
 Ad Solve            ||  441000 |      234371090 |  0.071151 | 40.142% | 40.142%
 Update              ||  441000 |       15494852 |  0.004704 |  2.654% |  2.654%
 Circulate Vectors   ||  441000 |        1211538 |  0.000368 |  0.208% |  0.208%

Solver: Cyclic Reduction with 8 lanes
Total Time: 0.143 seconds
 Block Name          ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  441000 |       28885736 |  0.008769 |  6.150% | 95.499%
 Rd Mult             ||  882000 |       48541868 |  0.014737 | 10.335% | 10.335%
 Dd Solve            ||  441000 |      174545600 |  0.052989 | 37.163% | 37.163%
 Dss Mult            ||  441000 |       19593448 |  0.005948 |  4.172% |  4.172%
 Ad Solve            ||  441000 |      176961780 |  0.053723 | 37.678% | 37.678%
 Update              ||  441000 |       15975487 |  0.004850 |  3.401% |  3.401%
 Circulate Vectors   ||  441000 |        1241717 |  0.000377 |  0.264% |  0.264%
-----------
Fs=88200
N=1600
L=2\.5
-----------

Solver: LAPACK
Total Time: 1.178 seconds
 Block Name          ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  882000 |       51862840 |  0.015745 |  1.336% | 98.710%
 Rd Mult             || 1764000 |      116227940 |  0.035285 |  2.995% |  2.995%
 Dd Solve            ||  882000 |     1813954700 |  0.550687 | 46.738% | 46.738%
 Dss Mult            ||  882000 |       35035584 |  0.010636 |  0.903% |  0.903%
 Ad Solve            ||  882000 |     1813994100 |  0.550699 | 46.739% | 46.739%
 Update              ||  882000 |       39973904 |  0.012135 |  1.030% |  1.030%
```

```
 Circulate Vectors  ||  882000 |        2461365 |  0.000747 |  0.063% |  0.063%

Solver: Jacobi Iteration
Total Time: 3.032 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  882000 |       57828760 |  0.017556 |  0.579% | 99.531%
 Rd Mult            || 1764000 |      120638560 |  0.036624 |  1.208% |  1.208%
 Dd Solve           ||  882000 |     3546455800 |  1.076658 | 35.511% | 35.511%
 Dss Mult           ||  882000 |       36887536 |  0.011199 |  0.369% |  0.369%
 Ad Solve           ||  882000 |     6178445000 |  1.875696 | 61.865% | 61.865%
 Update             ||  882000 |       36562616 |  0.011100 |  0.366% |  0.366%
 Circulate Vectors  ||  882000 |        2501178 |  0.000759 |  0.025% |  0.025%

Solver: Thomas, no precomputation
Total Time: 1.081 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  882000 |       50393250 |  0.015299 |  1.415% | 98.581%
 Rd Mult            || 1764000 |      128587780 |  0.039037 |  3.611% |  3.611%
 Dd Solve           ||  882000 |     1647406300 |  0.500130 | 46.258% | 46.258%
 Dss Mult           ||  882000 |       34083304 |  0.010347 |  0.957% |  0.957%
 Ad Solve           ||  882000 |     1650387800 |  0.501035 | 46.341% | 46.341%
 Update             ||  882000 |       39130212 |  0.011879 |  1.099% |  1.099%
 Circulate Vectors  ||  882000 |        2459595 |  0.000747 |  0.069% |  0.069%

Solver: Thomas, with precomputation
Total Time: 0.548 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  882000 |       47395384 |  0.014388 |  2.626% | 97.333%
 Rd Mult            || 1764000 |      124409860 |  0.037769 |  6.892% |  6.892%
 Dd Solve           ||  882000 |      775631000 |  0.235468 | 42.967% | 42.967%
 Dss Mult           ||  882000 |       33569930 |  0.010191 |  1.860% |  1.860%
 Ad Solve           ||  882000 |      776031800 |  0.235590 | 42.989% | 42.989%
 Update             ||  882000 |       36767880 |  0.011162 |  2.037% |  2.037%
 Circulate Vectors  ||  882000 |        2449244 |  0.000744 |  0.136% |  0.136%

Solver: Cyclic Reduction with 2 lanes
Total Time: 0.391 seconds
```

```
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  882000 |       56674190 |  0.017205 |  4.404% | 96.142%
 Rd Mult            || 1764000 |      116693770 |  0.035426 |  9.069% |  9.069%
 Dd Solve           ||  882000 |      511108930 |  0.155165 | 39.721% | 39.721%
 Dss Mult           ||  882000 |       40588988 |  0.012322 |  3.154% |  3.154%
 Ad Solve           ||  882000 |      512030100 |  0.155444 | 39.793% | 39.793%
 Update             ||  882000 |       39449612 |  0.011976 |  3.066% |  3.066%
 Circulate Vectors  ||  882000 |        2471286 |  0.000750 |  0.192% |  0.192%

Solver: Cyclic Reduction with 4 lanes
Total Time: 0.447 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  882000 |       54746020 |  0.016620 |  3.720% | 96.653%
 Rd Mult            || 1764000 |      117780180 |  0.035756 |  8.004% |  8.004%
 Dd Solve           ||  882000 |      606669100 |  0.184176 | 41.227% | 41.227%
 Dss Mult           ||  882000 |       35860650 |  0.010887 |  2.437% |  2.437%
 Ad Solve           ||  882000 |      607228600 |  0.184346 | 41.265% | 41.265%
 Update             ||  882000 |       39160000 |  0.011888 |  2.661% |  2.661%
 Circulate Vectors  ||  882000 |        2471645 |  0.000750 |  0.168% |  0.168%

Solver: Cyclic Reduction with 8 lanes
Total Time: 0.343 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves||  882000 |       56802930 |  0.017244 |  5.033% | 95.644%
 Rd Mult            || 1764000 |      116587384 |  0.035393 | 10.330% | 10.330%
 Dd Solve           ||  882000 |      433871740 |  0.131711 | 38.441% | 38.441%
 Dss Mult           ||  882000 |       36371530 |  0.011041 |  3.222% |  3.222%
 Ad Solve           ||  882000 |      435877660 |  0.132320 | 38.618% | 38.618%
 Update             ||  882000 |       39093692 |  0.011868 |  3.464% |  3.464%
 Circulate Vectors  ||  882000 |        2447891 |  0.000743 |  0.217% |  0.217%
-----------
Fs=176400
N=2200
L=2\.5
-----------

Solver: LAPACK
```

```
Total Time: 3.201 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 -------------------------------------------------------------------------------
 Multiply and Solves|| 1764000 |      148013360 |  0.044935 |  1.404% | 98.600%
 Rd Mult            || 3528000 |      318245060 |  0.096615 |  3.018% |  3.018%
 Dd Solve           || 1764000 |     4917929500 |  1.493020 | 46.637% | 46.637%
 Dss Mult           || 1764000 |       96937930 |  0.029429 |  0.919% |  0.919%
 Ad Solve           || 1764000 |     4916341000 |  1.492538 | 46.622% | 46.622%
 Update             || 1764000 |      127288750 |  0.038643 |  1.207% |  1.207%
 Circulate Vectors  || 1764000 |        4925465 |  0.001495 |  0.047% |  0.047%

Solver: Jacobi Iteration
Total Time: 17.545 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 -------------------------------------------------------------------------------
 Multiply and Solves|| 1764000 |      156156380 |  0.047407 |  0.270% | 99.763%
 Rd Mult            || 3528000 |      322496860 |  0.097905 |  0.558% |  0.558%
 Dd Solve           || 1764000 |    20106779000 |  6.104091 | 34.792% | 34.792%
 Dss Mult           || 1764000 |      106012680 |  0.032184 |  0.183% |  0.183%
 Ad Solve           || 1764000 |    36963240000 | 11.221439 | 63.960% | 63.960%
 Update             || 1764000 |      115747816 |  0.035139 |  0.200% |  0.200%
 Circulate Vectors  || 1764000 |        5058330 |  0.001536 |  0.009% |  0.009%

Solver: Thomas, no precomputation
Total Time: 3.150 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 -------------------------------------------------------------------------------
 Multiply and Solves|| 1764000 |      188850770 |  0.057331 |  1.820% | 98.377%
 Rd Mult            || 3528000 |      410011260 |  0.124471 |  3.952% |  3.952%
 Dd Solve           || 1764000 |     4735222000 |  1.437518 | 45.640% | 45.640%
 Dss Mult           || 1764000 |      131017960 |  0.039774 |  1.263% |  1.263%
 Ad Solve           || 1764000 |     4741746700 |  1.439498 | 45.703% | 45.703%
 Update             || 1764000 |      143156450 |  0.043459 |  1.380% |  1.380%
 Circulate Vectors  || 1764000 |        5175574 |  0.001571 |  0.050% |  0.050%

Solver: Thomas, with precomputation
Total Time: 1.582 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 -------------------------------------------------------------------------------
 Multiply and Solves|| 1764000 |      164363660 |  0.049895 |  3.154% | 96.867%
```

```
 Rd Mult            || 3528000 |      349775600 |  0.106179 |  6.711% |  6.711%
 Dd Solve           || 1764000 |     2213526800 |  0.671942 | 42.469% | 42.469%
 Dss Mult           || 1764000 |      105537090 |  0.032037 |  2.025% |  2.025%
 Ad Solve           || 1764000 |     2215604200 |  0.672573 | 42.509% | 42.509%
 Update             || 1764000 |      139116270 |  0.042230 |  2.669% |  2.669%
 Circulate Vectors  || 1764000 |        5103488 |  0.001549 |  0.098% |  0.098%

Solver: Cyclic Reduction with 2 lanes
Total Time: 1.148 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves|| 1764000 |      160712560 |  0.048791 |  4.249% | 95.696%
 Rd Mult            || 3528000 |      350984640 |  0.106555 |  9.279% |  9.279%
 Dd Solve           || 1764000 |     1497241600 |  0.454546 | 39.584% | 39.584%
 Dss Mult           || 1764000 |      112233110 |  0.034073 |  2.967% |  2.967%
 Ad Solve           || 1764000 |     1498419500 |  0.454904 | 39.616% | 39.616%
 Update             || 1764000 |      140715300 |  0.042720 |  3.720% |  3.720%
 Circulate Vectors  || 1764000 |        5390816 |  0.001637 |  0.143% |  0.143%

Solver: Cyclic Reduction with 4 lanes
Total Time: 1.294 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves|| 1764000 |      158342340 |  0.048071 |  3.714% | 96.283%
 Rd Mult            || 3528000 |      344411600 |  0.104559 |  8.079% |  8.079%
 Dd Solve           || 1764000 |     1746911500 |  0.530340 | 40.976% | 40.976%
 Dss Mult           || 1764000 |      105777140 |  0.032113 |  2.481% |  2.481%
 Ad Solve           || 1764000 |     1749385100 |  0.531091 | 41.034% | 41.034%
 Update             || 1764000 |      136620340 |  0.041476 |  3.205% |  3.205%
 Circulate Vectors  || 1764000 |        5297000 |  0.001608 |  0.124% |  0.124%

Solver: Cyclic Reduction with 8 lanes
Total Time: 0.997 seconds
 Block Name         ||  hits   |     clocks     |    time   |  % excl |  % incl
 ------------------------------------------------------------------------------
 Multiply and Solves|| 1764000 |      158185340 |  0.048023 |  4.816% | 95.183%
 Rd Mult            || 3528000 |      347948450 |  0.105632 | 10.594% | 10.594%
 Dd Solve           || 1764000 |     1250768500 |  0.379713 | 38.084% | 38.084%
 Dss Mult           || 1764000 |      112561230 |  0.034172 |  3.427% |  3.427%
 Ad Solve           || 1764000 |     1256583600 |  0.381479 | 38.261% | 38.261%
```

Update	1764000	136512540	0.041443	4.157%	4.157%
Circulate Vectors	1764000	5376189	0.001632	0.164%	0.164%

Appendix B

Final Project Proposal

In this appendix, the final project proposal submitted as part of the completion of this course is included.

Open-Tank Spring Reverberation FDTD Simulation - Final Project Proposal

Oliver Frank

August 18, 2024

1 Project Description

The last decade has seen a rise in popularity of open-tank spring reverberation units, which are essentially traditional spring reverbs, but with the springs exposed, allowing the musician to mess with them to create novel timbres. Some examples of this hardware are the Ekdahl Moisturizer created by Knas[1], and the Intellijel Springray [2]. The last few decades have also seen a large amount of interest in virtual analog effects. However, to the author's knowledge, there are currently no models of spring reverberation units that allow for the modes of inputs available with open-tank designs. The primary aim of this project will be to leverage existing finite difference time domain (FDTD) models of helical spring reverb units to provide these modes of interaction.

2 Background

There are a few different genres of spring reverb simulations, one such being those using allpass filter networks [3]. However, the only really acceptable approach for this use case is some sort of direct physical approach. I am opting to base my approach of Stefan Bilbao's 2013 paper, "Numerical Simulation Of Spring Reverberation,"[4] as it provides a model that physically simulates the geometry of the spring, and is not so computationally expensive as to rule out real time simulation. Additionally, I am familiar with FDTD techniques, having taken Stefan's "Physics Based Modelling of Musical Instruments" course.

3 Implementation

To create the simulation, I plan on beginning by creating a simulation in MATLAB, so that I am able to quickly create a prototype of the simulation, as well

[1] Knas Ekdahl Moisturizer
[2] Intellijell Springray[2]
[3] Spring Reverb Emulation Using Dispersive Allpass Filters in a Waveguide Structure (2006)
[4] Numerical Simulation of Spring Reverberation (2013)

as verify the stability of the system. Then, I will move to a C/C++ (or perhaps Zig, another systems programming language) to create a realtime plugin implementation that can be used inside of a DAW. If time permits, I would like to be able to create some sort of the real-time 3D visualization of the system in the plugin, as I have some graphics programming experience that would allow me to do so. I also plan on acquiring a spring reverb tank of some sort so that I can check that my simulation sounds close to the real thing.

The following is a provisional timeline for the project:

- **Weeks 1-2** - Research, planning, and initial MATLAB spring simulation implementation.
- **Weeks 3-4** - Add strike/pluck inputs to MATLAB simulation
- **Weeks 5-6** - Compare results with real spring tank and adjust model accordingly
- **Weeks 7-8** - Move model to C/C++/Zig plugin.
- **Weeks 10-12** - Create UI input mechanism.
- **Weeks 13-14** - User testing and adjustment of plugin.
- **Weeks 15-16** - Report writing, and 3D visualization.
- **Week 17** - Finalize report, tie up loose ends.

4 Equipment / Software

- MATLAB
- Digital Audio Workstation
- Spring Reverb Tank (£30-£50)

5 Supervisor

I believe Stefan Bilbao would be an ideal supervisor for this project, as I will be relying heavily on his work to complete the project.

Appendix C

Archive Listing

In this appendix, the README for the archive is given, which includes an accounting of the files in the archive, as well as instructions for how to run the various programs.

```
Included in this directory is:
- The dissertation report
- The Zig program with the various solver methods for the scheme (in
code/)
- The MATLAB prototype of the scheme
- The MATLAB dispersion relation programs
- Some demos of the effect

-- MATLAB PROGRAMS --
The MATLAB programs can be run by simply running them in the standard
manner within MATLAB.

-- ZIG PROGRAM --
NOTE: Due to the solver using Windows specific code, this program is
Windows only

For the Zig program, the user must obtain a copy of Zig 0.13:
https://ziglang.org/download/#release-0.13.0

Then, one can run the program from the 'code' directory, using the
following:
/path/to/zig build run -Doptimize=ReleaseFast -Dsolver="<solver name>" \
-Dsamplerate="<sample rate>" -DN="<grid points>" -DL="<spring length>"

solver should be one of:
```

```
lapack
jacobi
thomas
thomasprecomp
cr2
cr4
cr8

samplerate should be one of:
44100
88200
176400

L can be chosen freely.
N can be chosen freely, but should obey stability for the choice of L.

optimize should be set to ReleaseFast to ensure that the optimizer is
running on the most aggressive setting

An example program run would be (assuming zig is on the system path):
zig build run -Doptimize=ReleaseFast -Dsolver="cr8" \
-Dsamplerate="44100" -DN="1200" -DL="5.0"

The program will output a file called "zig_test.wav" that contains the
audio generated by the test.

-- DEMOS --
Some demos of the effect have been included in the demos/ folder:
DI_GuitarChords_44100.wav ................... Input for the other files
PROCESSED_GuitarChords_<sample_rate>.wav .... Simulation out using L=5
                                              for the given sample rate
EFFECTS_GuitarChords.wav .................... Effect run through an
                                              amplifier and some other
                                              effects to give a "surf
                                              rock" sound
```

Bibliography

[1] schroeder manfred r., "natural sounding artificial reverberation," *journal of the audio engineering society*, vol. 10, pp. 219–223, july 1962.

[2] J. O. Smith, "Physical modeling using digital waveguides," *Computer music journal*, vol. 16, no. 4, pp. 74–91, 1992.

[3] abel jonathan s., berners david p., costello sean, and smith julius o. iii, "spring reverb emulation using dispersive allpass filters in a waveguide structure," *journal of the audio engineering society*, no. 6954, october 2006.

[4] J. McQuillan and M. van Walstijn, "Modal spring reverb based on discretisation of the thin helical spring model," in *2021 24th International Conference on Digital Audio Effects (DAFx)*, 2021, pp. 191–198.

[5] W. Wittrick, "On elastic wave propagation in helical springs," *International Journal of Mechanical Sciences*, vol. 8, no. 1, pp. 25–47, 1966. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020740366900610

[6] S. Bilbao, "Numerical simulation of spring reverberation," in *DAFX Conference Proceedings*, 01 2013.

[7] A. Fog, "Optimizing software in c++," 2006. [Online]. Available: http://www.agner.org/optimize/optimizing_cpp.pdf

[8] ——, "The microarchitecture of intel, amd and via cpus," 2024. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

[9] C. Muratori, "Instrumentation-based profiling," *Computer Enhance*, 2023. [Online]. Available: https://www.computerenhance.com/p/instrumentation-based-profiling

[10] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference*, 2016.

[11] W. Ford, "Chapter 20 - basic iterative methods," in *Numerical Linear Algebra with Applications*, W. Ford, Ed. Boston: Academic Press, 2015, pp. 469–490. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B978012394435100020X

[12] W. Lee, "Tridiagonal matrices: Thomas algorithm," *MS6021, Scientific Computation, University of Limerick*, 2011.

[13] W. Gander[1] and G. H. Golub, "Cyclic reduction-history and applications," in *Scientific Computing: Proceedings of the Workshop, 10-12 March 1997, Hong Kong.* Springer Science & Business Media, 1998, p. 73.

[14] S. JOHNSSON, "Solving tridiagonal systems on ensemble architectures," *SIAM journal on scientific and statistical computing*, vol. 8, no. 3, pp. 354–392, 1987.

[15] M. Heath, "Lecture notes for cs554 parallel numerical algorithms," October 2015.