Algorithmic Design of Wind Instrument Shape via 3D FDTD and Deep Learning

by

Larry Wang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by .....
Justin Solomon
X-Consortium Career Development Assistant Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurtis
Chair, Master of Engineering Thesis Committee

Algorithmic Design of Wind Instrument Shape via 3D FDTD and Deep Learning

by

Larry Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Traditional design of wind instruments centers around simple shapes such as tubes and cones, whose acoustic properties are well understood and are easily fabricated with traditional manufacturing methods. The advent of additive manufacturing enables the realization of highly complex geometries and new wind instruments with unique sound qualities. While simulation software exists to predict the sound of wind instruments given their shape, the inverse problem of generating a shape that creates a desired sound is challenging given the computational cost of 3D acoustic simulations. In this work we create a fast 3D acoustic wind instrument simulator using GPU acceleration. In addition, we use deep learning to solve the inverse problem of generating a 3D shape that roughly approximates a desired sound when played as a single-reed instrument. Finally we develop an automatic method for determining pitch hole locations for a given shape to generate playable instruments.

Thesis Supervisor: Justin Solomon
Title: X-Consortium Career Development Assistant Professor

Acknowledgments

I would like to thank my advisor Dr. Justin Solomon for helping make this dream project of mine become a reality. I am very grateful for his mentorship, guidance, and expertise throughout both my master's and undergraduate here at MIT.

I would also like to thank my friend Peter Godart, a PhD student in Mechanical Engineering at MIT, for assisting in the 3D printing and fabrication process.

Contents

1 Introduction **13**

2 Related Works **15**

3 Problem Statement **17**

4 Review of FDTD-based Instrument Simulation **19**

5 Methods **23**

5.1 Fast 3D FDTD Audio Simulation through CUDA 23

5.1.1 Extending 2D to 3D 24

5.1.2 Overview of Parallel Programming with CUDA 25

5.1.3 Reducing global memory reads via shared memory 25

5.1.4 Reducing global memory reads via bit packing 28

5.1.5 Reducing GPU - CPU synchronization 28

5.1.6 Minimizing Branching 29

5.1.7 Aside: Low Pass Filtering 29

5.2 Interactive GUI for shape exploration 31

5.2.1 3D visualization 31

5.2.2 Controlling Simulation Constants 32

5.2.3 Shape Exploration 32

5.2.4 Saving Configurations 32

5.3 Audio Feature Extraction 32

5.3.1 Preprocessing 33

5.3.2 Peak Detection 34
5.3.3 Overtone Detection 35
5.4 Random Shape Generation 35
5.4.1 Generating spline curve 35
5.4.2 Discretization 36
5.4.3 Conversion to 3D 37
5.5 Deep Learning Model 37
5.5.1 Input and Output 38
5.5.2 Generating Training Data 38
5.5.3 Network Architecture 39
5.5.4 Hyper Parameter Tuning 39
5.6 Automatic Instrument Designer 40

6 Results **41**
6.1 Physical Accuracy of 3D FDTD Simulation 41
6.2 Speed of 3D FDTD Simulation 42
6.3 Performance of Audio to Shape Prediction 43
6.4 Accuracy of Tone Hole Placement 45
6.5 Example Printed Instrument 46

7 Conclusion and Future work **49**

A Tables **51**

B Figures **53**

List of Figures

4-1 FDTD Unit Cell 20

5-1 Row major ordering 27
5-2 Interactive Simulation GUI 31
5-3 Audio Feature Extraction Pipeline 34
5-4 Random Spline Generation 36
5-5 Discretized Spline 36
5-6 Neural Network Architecture 39

6-1 Spectrogram of Simulated Tube Instrument 42
6-2 Error Plots of Audio to Shape Neural Network 43
6-3 Error Plots of Tone Hole Placement Neural Network 45
6-4 CAD model of prototype instrument 46
6-5 Printed 3D Instrument 47

B-1 Effect of Number of Neurons in Hidden Layers 54
B-2 Effect of Dropout Rate 55
B-3 Effect of Learning Rate 56
B-4 Effect of L2 Regularization rate 57

List of Tables

5.1 Constants for Audio Feature Extraction Algorithm 35
5.2 Constants for Spline Generation Algorithm 37
5.3 Neural Network Hyperparameters 40
6.1 Errors for Predicted Pitch and Overtones 44
A.1 3D FDTD Simulation Constants 52

Chapter 1

Introduction

Timbre, the 'character' of a sound that allows the ear to distinguish a trumpet from a clarinet, is crucial to how we appreciate music. The desire for new sounds has driven the invention of a plethora of instruments from all cultures around the world. Traditional instrument design in the pre-digital age was made possible by understanding wave propagation in fundamental structures such as strings, pipes, and metal bars, along with expert craftsmanship and trial and error. However limitations in manufacturing abilities and high lead times make the iterative process of producing new acoustic instruments quite slow. The combination of modern computing power as well as additive manufacturing has the promise of accelerating the invention of a new class of computationally designed instruments.

While many aspects contribute to the timbre of an instrument, an especially salient feature is its overtone series. For pitched instruments, although the human ear may interpret each pitch as a single frequency (the fundamental frequency), in reality multiple frequencies, typically integer multiples of the fundamental (overtones), also sound at reduced but significant amplitudes. The ratio of these overtones contribute to the sonic 'signature' of a sound. The shape of a wind instrument has a large effect on not only the fundamental pitch (the longer the instrument the lower the frequency typically), but also the overtone series. As a classic example, the clarinet being a 'closed pipe' only resonates strongly on the odd harmonics, while the flute, an 'open pipe', resonates on all the harmonics. The shape of an instrument can be

thought of as a filter that only lets the resonant frequencies through, producing what we perceive as pitch and timbre. For complex shapes, modelling which frequencies will be let through is analytically intractable, but within the capabilities of modern machines.

In the search for designing new instruments and sounds, recent innovations have been primarily made in the realm of digital synthesizers. Digital synthesizers make it possible to generate arbitrary sound, introducing a wide range of new sounds that are physically impossible to produce with acoustic instruments. However, while digital synthesizers may be able to create a wider range of timbres than traditional acoustic instruments, musicians have less realtime expressive control over them. Brass players can adjust the subtleties of mouth pressure, air flow, and more while synthesizer players are typically limited to a few knobs controlling basic parameters such as volume or filter frequency. Adequately simulating the complex aerodynamics effects of wind pressure and mouth pressure in realtime is currently beyond most machines. The use of digital modelling to produce novel sounds not just through software instruments, but also through the design of new physical instruments is a largely untapped domain in the world of musical instrument design.

Chapter 2

Related Works

The problem of going from shape to sound is a famous one, popularized by the article "Can One Hear the Shape of a Drum?", which discusses whether just given the frequency spectrum of a sound produced by some hypothetical drum-head, the shape of the drum-head can be predicted. The answer for the 2D drum-head is that while a unique shape cannot be uniquely determined, certain properties of the shape can be inferred. Indeed [Cosmo et al., 2018] present an algorithm for shape correspondence, style transfer, and other challenging geometry problems inspired by this idea that *in practice*, shape reconstruction from its spectrum can be quite powerful. In this work we attempt to 'hear' the 3D shape of a wind instrument.

In regards to the simulation of wind instruments, the first fast wind instrument synthesis was developed by [McIntyre et al., 1983], with custom designed filters and feedback loops to emulate the essential characteristics of wind instruments. [Smith, 1986] later introduced the digital waveguide technique, which simulated the actual wave equation but in one dimension only. The first realtime 2D wave equation-based simulation was introduced by [Allen and Raghuvanshi, 2015], which cleverly takes advantage of GPU acceleration to create performable digital instruments. As GPU power increases, a natural extension to this is to bring simulation to full 3D, which we do in this work.

Several papers have tackled 3D FDTD computations, including [Webb and Bilbao, 2011] which uses CUDA acceleration to simulate 3D room acoustics and [Takemoto et al., 2010]

which performs 3D FDTD analysis of vowel production in the human vocal tract. [Arnela and Guasch, 2014] attempts to approximate 3D simulation with a 2D one. A 3D GPU-optimized FDTD implementation for wind instruments however has to the best of the author's knowledge not yet been presented in the literature.

The inverse problem of optimizing shapes to produce desired sounds has also been recently demonstrated for several types of instruments. [Bharaj et al., 2015] create a system which given an input shape and desired frequency spectrum, generates a metallophone that produces a specified frequency spectrum when struck, and attempts to retain as much as possible the shape of the original object. For wind instruments, [Li et al., 2016] take the approach of a modular set of acoustic filter embedded in a shape that can yield a desired filter. [Umetani et al., 2016] introduce a program called Printone that given an 3D mesh input, semi-automatically finds the holes to place in the mesh to generate the desired pitches . Both system only optimize for the fundamental frequency however, precluding optimization for timbre. We have designed a system that optimizes the shape of a wind instrument not just for fundamental pitch, but also its overtone series. Furthermore it automatically determines hole placement to produce a desired set of pitches, enabling the rapid design of new playable instruments.

Chapter 3

Problem Statement

In this work, we tackle two main problems.

1. Perform 3D acoustic simulation of clarinet-like single reed wind instrument in reasonable time.

2. Perform inverse problem of generating shape that produces sound matching requirements of the user

   (a) Generate shape that produces user's desired fundamental frequency and overtone ratios as closely as possible

   (b) Determine where to place tone holes for the user's desired pitches

In this work we assume the simulation results to be the ground truth, and do not perform extensive analysis on whether the simulation results accurately reflect experimental results from a real physical system. Nevertheless, a more physically accurate simulation can be swapped in with fairly little modification to our overall framework.

Chapter 4

Review of FDTD-based Instrument Simulation

There are several methods of performing audio simulation, the two most common being the Finite Element Method (FEM) and Finite Difference Time Domain (FDTD). We choose to use FDTD due its simplicity as well its ability to be parallelized easily. In particular we base our simulation off the model proposed in [Allen and Raghuvanshi, 2015], which uses FDTD to simulate a variety of wind instruments. Below is a review of their FDTD algorithm.

The radiation of sound waves in an instrument can be approximated using coupled wave equations.

$$\frac{\partial p}{\partial t} = -\rho C_s^2 \nabla \cdot \mathbf{v} \tag{4.1}$$

$$\frac{\partial \mathbf{v}}{\partial t} = -\frac{1}{\rho} \nabla p \tag{4.2}$$

p is the pressure, $\mathbf{v}$ is the velocity vector, ρ is the mean density, and C_s is the speed of sound.

FDTD is a method for solving Equations 4.1 and 4.2. The FDTD employs a 'staggered grid', where pressure and velocity values are conceptually staggered in the time and spatial dimensions as seen in Figure 4-1. This allows second order accuracy

despite using only what seems like a first order accurate stencil.

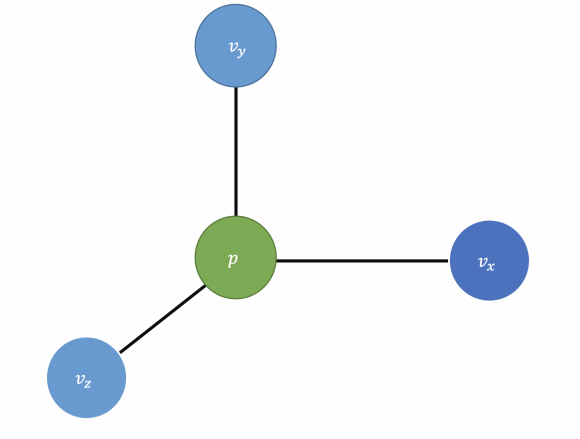

Figure 4-1: FDTD Unit Cell: Pressure and velocity are staggered on a grid

As the domain for FDTD is finite, and larger grid sizes take longer to simulate, there needs to be an absorbing layer at the edges of the domain to prevent the waves from bouncing back at the edges. This is achieved with a Perfectly-Matched-Layer PML region, where cells near the edge of the boundary have a non zero σ value that controls absorption. The σ values linearly decrease from $0.5/\Delta_t$ at the edge of the domain, to zero over the thickness of the PML region. Incorporating the PML, the updated equations become:

$$\frac{\partial p}{\partial t} + \sigma p = -\rho C_s^2 \nabla \cdot \mathbf{v} \tag{4.3}$$

$$\frac{\partial \mathbf{v}}{\partial t} + \sigma \mathbf{v} = -\frac{1}{\rho} \nabla p \tag{4.4}$$

The boundary conditions in the PDE come from both the walls of the instrument and the excitation mechanism. Boundary conditions are incorporated by forcing 'virtual velocities' on cells in the grid that are boundary cells. This can be expressed by the variable β, where $\beta = 1$ indicates an 'air' cell and $\beta = 0$ indicates a boundary condition with velocity set to $\mathbf{v_b}$. Suppose $\sigma' = 1 - \beta + \sigma$. We can express the boundary conditions elegantly as:

$$\frac{\partial p}{\partial t} + \sigma' p = -\rho C_s^2 \nabla \cdot \mathbf{v} \tag{4.5}$$

$$\beta \frac{\partial \mathbf{v}}{\partial t} + \sigma' \mathbf{v} = -\beta^2 \frac{\nabla p}{\rho} + \sigma' \mathbf{v_b} \tag{4.6}$$

In the case of wall cells, we simply set $\mathbf{v}_b$ to zero. For the exciter cells, we set $\mathbf{v}_b$ to the velocity determined by the excitation mechanism, which attempts to model the highly nonlinear behaviour of the exciter. For the clarinet-like mechanism for example, we have

$$|\mathbf{v}_b| = \frac{w_j h_r}{(\Delta_s^2 N)} \frac{1 - \Delta p}{\Delta p_{max}} \sqrt{\frac{2\Delta p}{\rho}} \tag{4.7}$$

where $\Delta p = p_{mouth} - p_{bore}$. p_{mouth} is the pressure on the reed, a constant, and p_{bore} is the current pressure at a chosen location inside the bore of the instrument, typically right after the excitation location. The meaning behind the remainder of the variables can be found in [Allen and Raghuvanshi, 2015].

In discrete form, the p and $\mathbf{v}$ values are staggered on a grid. Suppose the pressure locations are at $p[x, y, z, t]$. The velocity values then are placed at $\mathbf{v}_x[x+1/2, y, z, t+1/2]$,$\mathbf{v}_y[x, y+1/2, z, t+1/2]$, and $\mathbf{v}_z[x, y, z+1/2, t+1/2]$. Expressing the data with integer indices, we can collapse the half steps into the same index, i.e. a particular cell location $D[x, y, z, t]$ can be thought of as a unit cell (see Figure 4-1) containing the values p, $\mathbf{v}_x$, $\mathbf{v}_y$, $\mathbf{v}_z$ where $D[x, y, z, t].\mathbf{v}_x = \mathbf{v}_x[x + 1/2, y, z, t]$ and so forth.

The discretized forms of Equations 4.5 and 4.6 are

$$p[n+1] = \frac{p[n] - \rho C_s^2 \Delta_t \tilde{\nabla} \cdot \mathbf{v}[n]}{1 + \sigma' \Delta_t} \tag{4.8}$$

$$v[n+1] = \frac{\beta \mathbf{v}[n] - \beta^2 \Delta_t \tilde{\nabla} p[n+1]/\rho + \sigma' \Delta_t \mathbf{v_b}[n+1]}{\beta + \sigma' \Delta_t} \tag{4.9}$$

$\tilde{\nabla}$ is the discrete spatial derivatives. With the unit cell as defined above,

$$\begin{aligned}\tilde{\nabla} \cdot \mathbf{v}[x,y,z,n] = &\frac{\mathbf{v}_x[x,y,z,n] - \mathbf{v}_x[x-1,y,z,n]}{\Delta_s} \\ &+ \frac{\mathbf{v}_y[x,y,z,n] - \mathbf{v}_y[x,y-1,z,n]}{\Delta_s} \\ &+ \frac{\mathbf{v}_z[x,y,z,n] - \mathbf{v}_z[x,y,z-1,n]}{\Delta_s}\end{aligned} \tag{4.10}$$

Although this seems like a first order stencil, in actuality these indices map to

$$\begin{aligned}\tilde{\nabla} \cdot \mathbf{v}[x,y,z,n] = &\frac{\mathbf{v}_x[x+1/2,y,z,n] - \mathbf{v}_x[x-1/2,y,z,n]}{\Delta_s} \\ &+ \frac{\mathbf{v}_y[x,y+1/2,z,n] - \mathbf{v}_y[x,y-1/2,z,n]}{\Delta_s} \\ &+ \frac{\mathbf{v}_z[x,y,z+1/2,n] - \mathbf{v}_z[x,y,z-1/2,n]}{\Delta_s}\end{aligned} \tag{4.11}$$

, which is a second order centered difference stencil.

Similarly,

$$\begin{aligned}\tilde{\nabla} p[x,y,z,n] = &\frac{p[x+1,y,z,n] - p[x,y,z,n]}{\Delta_s} \\ &+ \frac{p[x,y+1,z,n] - p[x,y,z,n]}{\Delta_s} \\ &+ \frac{p[x,y,z+1,n] - p[x,y,z,n]}{\Delta_s}\end{aligned} \tag{4.12}$$

Chapter 5

Methods

5.1 Fast 3D FDTD Audio Simulation through CUDA

In order to solve the inverse problem of finding the shape that produces a desired sound, the forward problem of simulating the sound given a shape should be as fast as possible. Full 3D simulation is desired for its higher accuracy compared to 1D and 2D methods as well as its straightforward translation to fabrication using 3D printers.

While [Allen and Raghuvanshi, 2015] developed the first real-time FDTD-based wind instrument simulation using GPU acceleration, their algorithm was implemented in 2D only. Moreover their implementation utilized programmable GLSL shaders, which do not have the flexibility and optimization opportunities of CUDA, NVIDIA's dedicated platform for parallel computing. In this work we extend [Allen and Raghuvanshi, 2015] and implement a 3D aerophone simulation in CUDA. Some modifications have been made to the original implementation. Only the clarinet-like single reed excitation model is implemented and the wall loss computations are not included in the final implementation due to their additional computational complexity and the absence of the 'parasitic resonance' problem observed by the authors in [Allen and Raghuvanshi, 2015] with the settings we use. We employ different constant values from the original, see Table A.1 for the values used.

5.1.1 Extending 2D to 3D

As mentioned by [Allen and Raghuvanshi, 2015], extending their framework to 3D is fairly straightforward conceptually. Two dimensional difference stencils become three dimensional and the excitation mechanism remains the same. The primary challenge lies in the added computational complexity of the additional dimension. Not only does the number of cells increase dramatically, the cost of computing the updated pressure and velocity values for each cell increases as well due to additional directions that must be accounted for. Nevertheless, the FDTD algorithm is still highly parallelizable as updating each cell still only depends on a small local region around it.

At its core, the update equations for the 3D FDTD can be expressed in the form:

$$\begin{aligned} p[x,y,z,t+1] = \; & f(\; p[x,y,z,t], \mathbf{v}_x[x,y,z,t], \mathbf{v}_x[x-1,y,z,t], \\ & \mathbf{v}_y[x,y,z,t], \mathbf{v}_y[x,y-1,z,t], \\ & \mathbf{v}_z[x,y,z,t], \mathbf{v}_z[x,y,z-1,t]) \end{aligned} \tag{5.1}$$

$$\begin{aligned} \mathbf{v}_x[x,y,z,t+1] = \; & g(\; \mathbf{v}_x[x,y,z,t], \\ & p[x+1,y,z,t+1], p[x,y,z,t+1]) \end{aligned} \tag{5.2}$$

$$\begin{aligned} \mathbf{v}_y[x,y,z,t+1] = \; & g(\; \mathbf{v}_y[x,y,z,t], \\ & p[x,y+1,z,t+1], p[x,y,z,t+1]) \end{aligned} \tag{5.3}$$

$$\begin{aligned} \mathbf{v}_z[x,y,z,t+1] = \; & g(\; \mathbf{v}_z[x,y,z,t], \\ & p[x,y,z+1,t+1], p[x,y,z,t+1]) \end{aligned} \tag{5.4}$$

x, y, and z are the spatial dimensions, t is the time dimension, p is the pressure and $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$ are the velocities in their respective dimension. As seen in the update equations f and g, updating the values for each cell only requires access to a small neighborhood of cells around it. Within each time step, the update equations for each cell can be run independently, as each cell only needs read access to already computed values and there are no overlapping writes. Note however, the velocity update steps

must happen after all the pressure steps have finished due to the staggered nature of the FDTD. The simplest way to parallelize this FDTD equation would be to assign each thread to a cell and compute all the pressure update equations. Once all the pressure updates have finished, the same can be done for the velocity steps and so forth. While this works, several performance optimization can be made to take the most advantage of current GPU technology.

5.1.2 Overview of Parallel Programming with CUDA

Here we give a brief overview of performance considerations when working with GPU parallel programming through CUDA. GPUs excel at Single Instruction Multiple Data, SIMD, computations where the same operation is done many times but on different sets of data. This conveniently maps to the FDTD algorithm where the update step for each cell in the grid is essentially identical. However to achieve good performance, certain optimizations must be made including:

- Reducing reads from global GPU memory
- Reducing GPU - CPU synchronization
- Minimize branching

Below is a selection of performance optimization we have implemented, many inspired by [Allen and Raghuvanshi, 2015].

5.1.3 Reducing global memory reads via shared memory

As is the case in CPU programming, reading from memory is expensive. On NVIDIA GPUs, there is a memory hierarchy, similar to that on a CPU. The slowest but largest memory is global memory, accessible from all streaming multiprocessors (SMs) on the GPU. An SM is a processor responsible for running a group of threads in parallel, and every NVIDIA GPU has several of them. Workloads are divided by the programmer into thread blocks, which are assigned by the GPU to SMs to run.

In addition to global memory, there are faster but smaller memory caches that store data more likely to be accessed soon, as in the CPU. Unique to GPUs is shared memory, dedicated memory for each SM colocated on each SM chip. Shared memory can be 100 times faster than global memory, but can only be accessed by the threads running on the SM owning the shared memory. A key optimization strategy is to reduce reads from global memory as much as possible by taking advantage of these faster memory caches.

Another way to significantly reduce reads from global memory is memory coalescing. Memory coalescing occurs when sequential threads in a thread group access sequential memory addresses, i.e. thread 1 accesses memory address 1004, thread 2 address 1008, thread 3 address 1012, etc. When this condition is met, the GPU can read an entire range of memory addresses in one go for the cost of one.

Unfortunately for computations in 2D and 3D, this breaks down due to issue of 'striding'. Consider the 2D case and suppose the 2D grid is stored as an array in row-major order, where each row is laid out contiguously in memory as seen in Figure 5-1. If a group of threads is each assigned to read from a row of the grid, then the memory addresses will be coalesced into one read as they are contiguous. On the other hand, if the group of threads is assigned to read a column, the memory addresses will be offset by the 'stride' of the y dimension and the memory reads will not be coalesced. There is a large penalty for reading consecutive values in one dimension versus the other due to the fact that fundamentally the grid is stored in memory as a 1D array. In the 3D FDTD, each cell must access neighboring locations in the x, y, and z direction, but unfortunately adjacent cells in the y and z directions are not adjacent in memory.

Additionally, we see that in the update equations 5.1 through 5.4, there is read redundancy between different cells. For example computing $p[x, y, z, t]$ and $p[x+1, z, t]$ both require values at location (x, y, z).

In order to reduce reads from global memory as much as possible, we follow the suggestions of [Micikevicius, 2009], which provides a recommendation for computing 3D finite difference stencils on NVIDIA GPUs. The key idea is the use of shared memory as an explicit cache. Each thread block, which is assigned to a subsection of

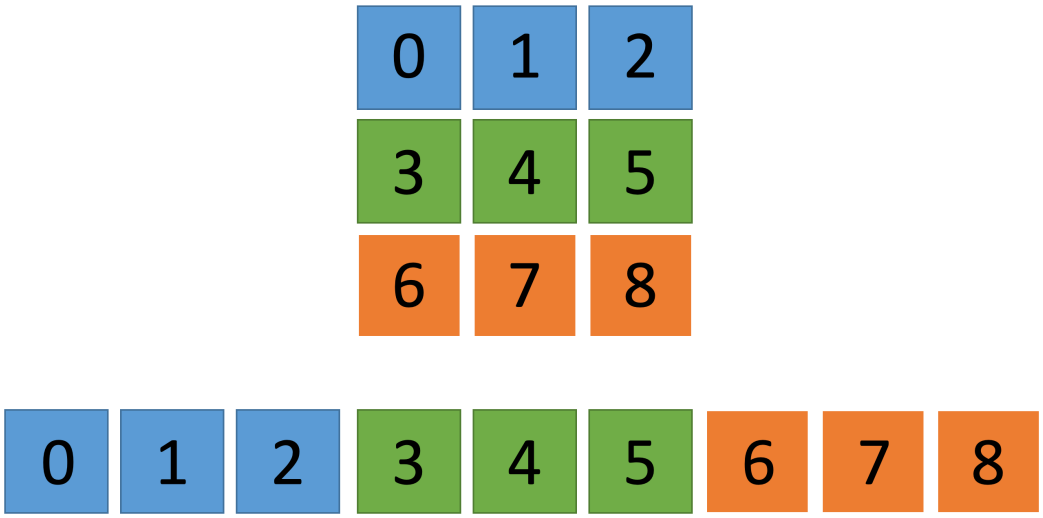

Figure 5-1: Row major ordering

the 3D domain, first copies all the data necessary to complete all the computations in the subsection from global memory to shared memory. This requires loading a slightly larger subsection of the 3D grid than just the cells within the subsection as the cells on the border require access to memory regions outside. Each thread is responsible for transferring some region of the required memory from global to shared memory. Once all the required memory is loaded into shared memory, then the update functions downstream can read from the shared memory instead of global memory. However due to the parallel nature of threads, we must specifically ensure that all the threads have completed writing to shared memory before continuing, so a block synchronization is inserted before the update step begins.

While having each thread block handle a 3D subsection of the grid seems the most natural, the amount of shared memory is limited and we are typically unable to store all of the memory for a 3D block of reasonable size into shared memory. Instead, as suggested by [Micikevicius, 2009], each thread block is assigned to a 2D subsection in the xy plane. Each thread group then travels along the z axis, completing the updates for one xy layer at a time. Each time the thread group advances to the next layer, it loads into shared memory the data necessary for that layer. Each thread

also keeps track of the forward and behind as it moves along the z axis, minimizing rereads along the z axis.

5.1.4 Reducing global memory reads via bit packing

Another way to reduce reads from global memory is to compact the data itself. In [Allen and Raghuvanshi, 2015], the authors pack all the data necessary for a cell into one RGBA32 pixel, with the pressure and two velocity fields (2D) occupying the first three channels and 'auxiliary' data in the last channel. The auxiliary data consists of several pieces of data that each occupy some subregion of bits within the 32 bits available.

Using CUDA an analogous approach would be using the four components of a `float4` datastructure, a vector of four floats. Each `float4` is fetched in one go due to memory coalescing. Unfortunately in the 3D case, as there are three velocity values per cell instead of two, the pressure and velocity fields take up all the room in the `float4`. Additional memory is required to store the auxiliary data. To minimize the cost of fetching the auxiliary data, we also use this bit packing idea. The fields we store are whether the cell is a wall (1 bit), whether it is an exciter (1 bit), and the σ value of the cell (3 bits), which is nonzero within the PML layer. While the σ value is technically a real number, if the PML layer is n layers thick there are only $n + 1$ unique values (including zero). For a fixed PML layer size, in our case 7, only 8 distinct σ values exist, which is expressible with 3 bits. The actual sigma value can be easily recomputed by each thread as it just requires a multiplication. The remaining bits are used to store information regarding the direction of wall normals for incorporating wall losses as described in [Allen and Raghuvanshi, 2015], but were not included in the final implementation.

5.1.5 Reducing GPU - CPU synchronization

Another major cost in GPU computing is synchronization between the GPU and CPU. Often times, the GPU or CPU may be forced to stall waiting for the other,

leaving GPU cores idle as they wait. As mentioned above, the velocity steps must wait for the pressure steps to finish and vice versa, requiring blocking to ensure that all threads have finished before advancing. Ideally we could combine the pressure and velocity steps into one step, as done in [Allen and Raghuvanshi, 2015]. This is doable at the expense of additional computation. For each time step, rather than having each cell just compute its own future pressure value, it also computes the future pressure values for all cells the future velocity values at the cell depend on. In the 3D case, for cell location (x, y, z), the future pressure cells for locations $(x+1, y, z)$, $(x, y+1, z)$, and $(x, y, z+1)$ must also be computed. In order to compute the pressure values for these locations, additional memory locations must also be fetched in the forward and backward planes. Whereas previously we only needed $(x, y, z-1)$ and $(x, y, z+1)$ in the forward and backward planes, now $(x+1, y, z-1)$, $(x, y+1, z-1)$, etc. are required as well. In this work, these additional values are not cached in shared memory but this optimization certainly can be made.

5.1.6 Minimizing Branching

While branching `if`, `else` statements etc. are supported in CUDA, they often come with a performance hit. In the hardware, threads are grouped into warps and within each warp, all threads run the same instructions in lockstep. When a branch is encountered and threads in the warp must follow different paths, the GPU must essentially run each of those different paths for each thread, reducing throughput. Fortunately, not many conditionals are needed as the different behaviour of the wall, exciter, and air cells is already incorporated in the Equations 4.8 and 4.9.

5.1.7 Aside: Low Pass Filtering

One detail left out of [Allen and Raghuvanshi, 2015] is how the discrete low pass filters are implemented. In this work we use low pass filters to lowpass both the output of the excitation mechanisms to reduce spurious noise, as done in [Allen and Raghuvanshi, 2015], and in order to down sample the generated audio signal to the standard 44100 Hz.

We choose to implement the lowpass filter as a second order biquad filter using the transformed direct form 2, which requires two delay lines, i.e. two numbers must be stored for the next iteration. On the GPU we achieve this by reserving 4 floats of GPU memory. We reserve 4 rather than 2 because we need to prevent threads which are running in parallel from reading and writing to the same locations. We use the iteration number to determine which 2 floats of the 4 are being written to during this iteration, as all threads share the same iteration number during one step of FDTD. We ping pong back and forth between which two floats are read and which two are write based on the parity of the iteration number. We choose to lowpass the output excitation mechanisms on the GPU rather than the CPU as lowpass filtering on the CPU would force a GPU-CPU synchronization between every iteration. Filtering on the CPU would require reading memory from the GPU to the CPU and then writing the low passed value back to the GPU, greatly reducing throughput.

For downsampling the final audio signal to 44100 HZ, we use a CPU implementation as the GPU has already finished its iterations. Downsampling is required because the FDTD necessitates a very small time step Δ_t to achieve stability for our desired spatial resolution Δ_s seen in Table A.1, much smaller than that required by audio. The condition for stability is the Courant Friedrichs Lewy (CFL) condition which in the 3D case is

$$dt < ds/(C_s\sqrt{3}) \tag{5.5}$$

The output of our FDTD algorithm thus has a sample rate of 573300 Hz, exactly 13 times 44100 Hz. Taking every 13 samples would bring the sample rate down to the desired 44100 Hz but could lead to aliasing if there are high frequency components greater than the Nyquist frequency of 44100 Hz, 22050 Hz. In order to minimize aliasing, we first perform a lowpass with 10khz cutoff before picking every 13 samples.

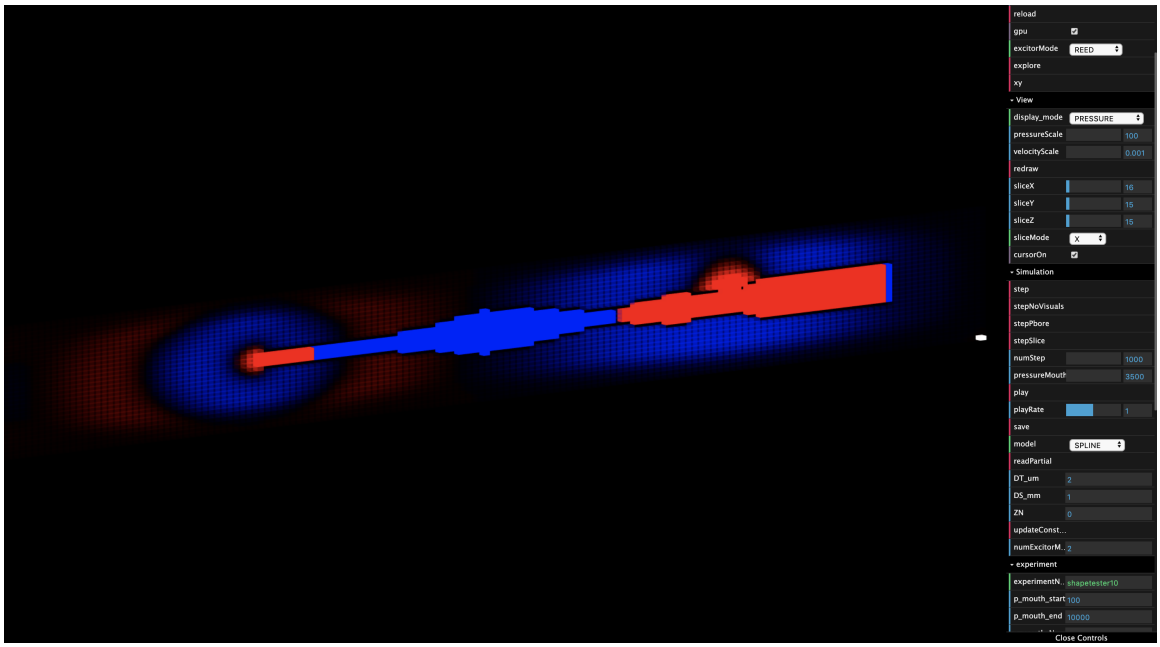

Figure 5-2: Interactive Simulation GUI. The pressure values of a cross section of the 3D domain are being visualized

5.2 Interactive GUI for shape exploration

In addition to the algorithm, we have implemented a 3D visualization tool to aid in visualizing and debugging the simulation as well as helping users explore and design instruments. Our tool runs in the browser using javascript and uses the three.js library for 3D rendering. Below is an overview of the high level features of the tool.

5.2.1 3D visualization

Our tool allows the user to inspect the 3D voxel grid of the simulation, with the orbiting controls found in typical 3D software. This allows to user to visually see the progress of the simulation and also aids greatly in debugging. The user can choose to inspect several aspects of the state of the simulation, from pressure and velocity values, to auxiliary states such as which cells are wall cells, exciter cells, part of the PML layer, etc. The user can step through the simulation at their desired step size and see a snapshot of the simulation at any state. Slicing options in all axes are available for the user to see a 2D cross section of the simulation state.

5.2.2 Controlling Simulation Constants

From the GUI, the user can interactively change the constants of the model, including dimensions of the domain, spatial and time resolution, mouth pressure on the reed, etc. Given the importance of these constants in the simulation, it is very helpful to be able to change these constants on the fly without having to recompile.

5.2.3 Shape Exploration

To aid the exploration of new shapes, the GUI can automatically generate and visualize new instrument shapes via the process explained in section 5.4, as well as other modes such as a generator for simple flared bell shapes that are parameterized by intuitive settings. Given the complexity of 3D modelling, this aids the user in quickly generating new candidate shapes. The user also has the option of generating a voxel dataset in an external program and loading it in the GUI to simulate arbitrary 3D shapes.

5.2.4 Saving Configurations

The user has the ability to load and save different configurations of the simulation, including associated constants, visualization modes, etc. This makes it very easy to experiment with different settings and switch back and forth without worrying about losing good configurations.

5.3 Audio Feature Extraction

In order to analyze a large batch of audio files, it is necessary to derive analytic and intuitive features that capture the 'character' of the sound. This is also important for the machine learning algorithm described in Section 5.5 as a reduced data dimensionality will help in training speed and effectiveness. Moreover, having a limited set of features to describe audio aids the user in defining the type of audio output they desire.

In this work, the features we choose to extract from a given sound are the fundamental frequency and the relative strength of the first N_o overtones. While there are theoretically infinite overtones, the most prominent overtones are typically the lower ones so we ignore the higher overtones in this work. The threshold can certainly be increased at will. While there are many aspects that affect the perceived 'character' of a sound, the strength of the overtones is a key discriminator. In summary, the 'audio signature' used to describe each snippet of audio consists of a size $N_o + 1$ vector, the first element being the fundamental frequency in hertz and the remaining elements the relative strength of the harmonics compared to the fundamental frequency.

In order to robustly detect pitch and overtone strength, the following algorithm is used, see Figure 5-3.

1. Preprocessing
   (a) Apply Hanning window to audio signal
   (b) Zero pad signal
2. Perform DFT, take the magnitude
3. Detect primary peak to determine fundamental frequency
4. Determine strength of integer multiples of fundamental frequency

Details on various steps of the algorithm are provided below

5.3.1 Preprocessing

When instruments generate notes, there is often a strong attack preceding the steady state waveform. As the scope of this work is restricted to the steady state behaviour, we multiply a Hanning window to smooth out the beginning and the end. We then zero pad our signal by Z_N samples to increase the frequency resolution of the DFT. The frequency resolution of the DFT is restricted by the number of bins, which is directly proportional to the input size of the signal.

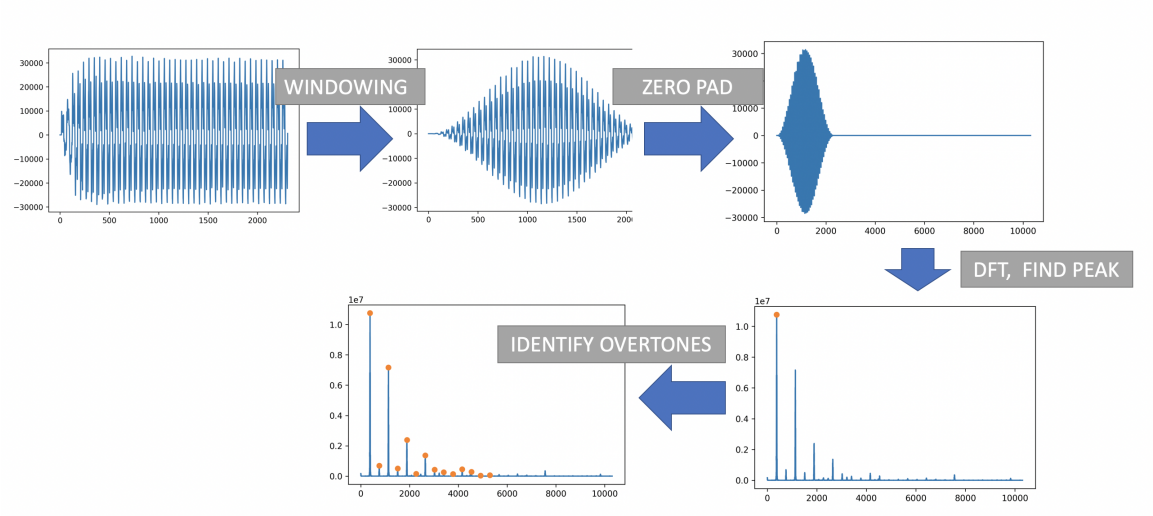

Figure 5-3: Audio Feature Extraction Pipeline

5.3.2 Peak Detection

Let X be the magnitude of the DFT. Typically the highest peak of the magnitude of the DFT indicates the fundamental pitch of the sound. This is not always the case as in some instruments, like the trumpet, an overtone might actually be stronger than what is perceived as the fundamental frequency. In order to account for this, we determine a threshold θ such that all candidate peaks must be greater than the $\theta \times \max(X)$.

For peak detection, we define peaks to be any location where the signal is greater than both its neighbors. However this will lead to many false positives if the signal is noisy. We impose the additional condition that peaks must be a certain width W_p away from each other. In the peak finding process, first the highest peak is chosen, and than any peaks within width W_p away from the peak are removed from consideration. Finally, we choose the lowest peak that satisfies both the threshold and width condition, assuming the lower peak is the true fundamental.

5.3.3 Overtone Detection

Once the fundamental pitch is assumed, the overtones can be found as the integer multiples of the frequencies of the fundamental pitch. This is under the assumption of the absence of inharmonicity, which is the presence of overtones not integer multiples of the base frequency. Unfortunately simply querying the frequency bins that are integer multiples of the fundamental fails due to discretization errors of the DFT and noise. To resolve this, for each overtone of interest, we instead query a small neighborhood of width W_o around the supposed overtone location, and pick the max amplitude. Once the amplitudes of the overtones are determined, we normalize by dividing all the amplitudes by the amplitude of the fundamental. See Table 5.1 for a list of constants chosen in our implementation.

Symbol	**Meaning**	**Range**
θ	Peak height threshold	0.6
W_p	Peak width	100
Z_N	Zero padding amount	8000
N_o	Number of overtones to query	13
W_o	Overtone search width	100

Table 5.1: Constants for Audio Feature Extraction Algorithm

5.4 Random Shape Generation

In order to search the design space of possible shapes and the sounds they produce, in this work we restrict ourselves to a limited subset of possible shapes. We fix the length of our instrument to L spatial units and random shapes are generated via the following process:

5.4.1 Generating spline curve

First we generate a 1D BSpline curve consisting of S_N points. For each of the S_N points, we generate a random number between R_{min} and R_{max}. We choose to use a

spline curve as it varies smoothly, which seems natural for an instrument design. See Figure 5-4.

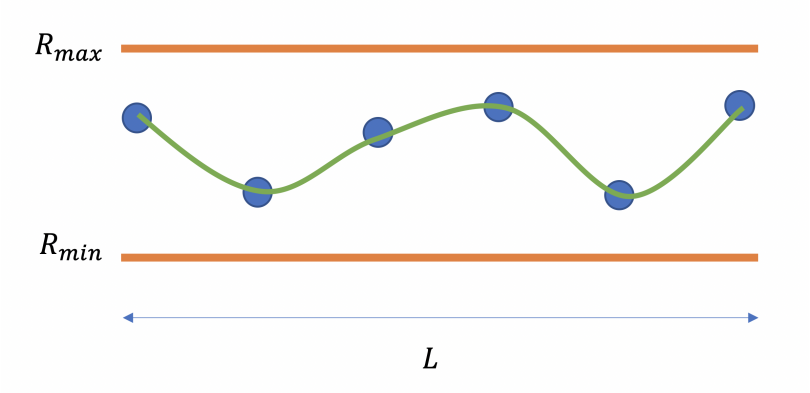

Figure 5-4: Random Spline Generation

5.4.2 Discretization

We discretize our spline curve by computing for each step $n \in [0, L]$ the value of the BSpline at $t = n/L$, assuming the BSpline starts at $t = 0$ and ends at $t = 1$. We then round the result to the nearest integer to end up with a length L vector of integers we will call V_r. Each integer represents the 'radius' of the cross section of the instrument at that position along the length of the instrument. Essentially it defines a discrete 1D profile of the instrument. Figure 5-5 is an example of a randomly generated and discretized spline curve (mirrored).

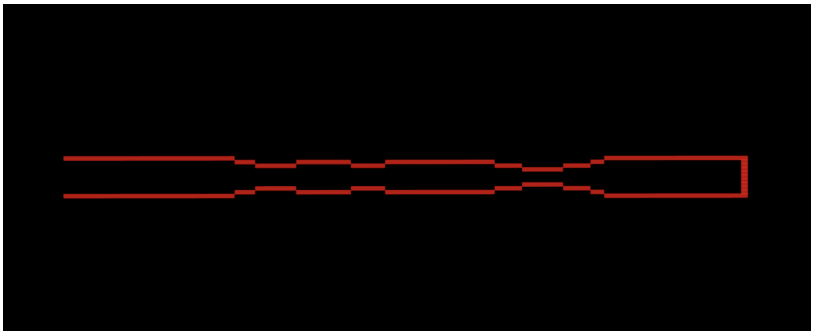

Figure 5-5: Discretized Spline, mirrored across axis

Symbol	Meaning	Range
L	Length of bore	100
S_N	Number of spline points	12
R_{min}	Minimum height of spline curve	2
R_{max}	Maximum height of spline curve	6

Table 5.2: Constants for Spline Generation Algorithm

5.4.3 Conversion to 3D

In order to convert the 1D profile into a 3D shape we use the following algorithm: For each n along L, we create a square hoop with thickness one and half-width $V_r[n]$, centered along a chosen axis. Essentially we are revolving the profile around the axis, but using a square rather than a circle. Then we designate the interior of the square found at the beginning of the instrument as the exciter cells. We now have a 3D voxel grid representation of a tube like instrument with perturbations. See Table 5.2 for constants used.

5.5 Deep Learning Model

With a forward model for going from shape to sound, we now attempt the inverse problem of going from sound to shape. Specifically we solve two different problems:

1. Given desired audio features, predict the geometry that would generate the desired sound

2. Given the geometry of the bore and a desired pitch, predict where the tone holes should be placed to achieve the desired pitch

Given the complexity of the inverse problem and the fact that we have a working simulation and data generation is cheap, we choose to use a machine learning approach.

5.5.1 Input and Output

Audio to Shape: Rather than expressing the audio data and geometry voxel data directly, we represent the audio using the feature vector described in Section 5.3 and the geometry as a list of spline points as described in Section 5.4.

- **INPUT**: Size $N_o + 1$ vector, first component is desired pitch, remaining N_o components are the relative strength of the overtones (Table 5.1)
- **OUTPUT**: Size S_N vector of the heights of the spline curve (Table 5.2)

For a frequency f, rather than represent it in terms of Hz, we find the corresponding MIDI pitch m number corresponding to the Hz and scale by 127.

$$m = \frac{69 + 12\log_2(f/440)}{127} \tag{5.6}$$

This more accurately represents how the human ear experiences pitches and also expresses most pitches within human hearing as a scalar between 0 and 1.

Shape and Pitch to Tone Hole location

- **INPUT**: Size $S_N + 1$ vector, first component is desired pitch, remaining S_N components are the heights of the spline curve (Table 5.1)
- **OUTPUT**: Scalar indicating location along bore to insert tone hole

5.5.2 Generating Training Data

In order to generate training data, we generate many random spline curves as described in 5.4 and run the FDTD simulation for 30000 steps, which is enough to capture the steady state region of the audio. For the pitch hole problem, we also randomly select a hole position along the bore from $[0, L)$ (Table 5.2) and remove a 3x3 rectangular patch centered on the hole position from the bore wall. We generate data in parallel on a machine with four GeForce GTX 1080 Ti GPUs, 130 gigabytes of memory, and a 4.5GHz processor. Approximately 3 sound files are generated every second.

5.5.3 Network Architecture

For both cases, we use essentially the same neural network architecture, just with different input shape and output shape. We use a fully connected neural network with an input layer, two hidden layers, and output layer (see Figure 5-6). To prevent over-fitting, we have a dropout layer after each hidden layer, L2 regularization, and early stopping with a patience of 10. We use ADAM as the optimizer and mean squared error for loss.

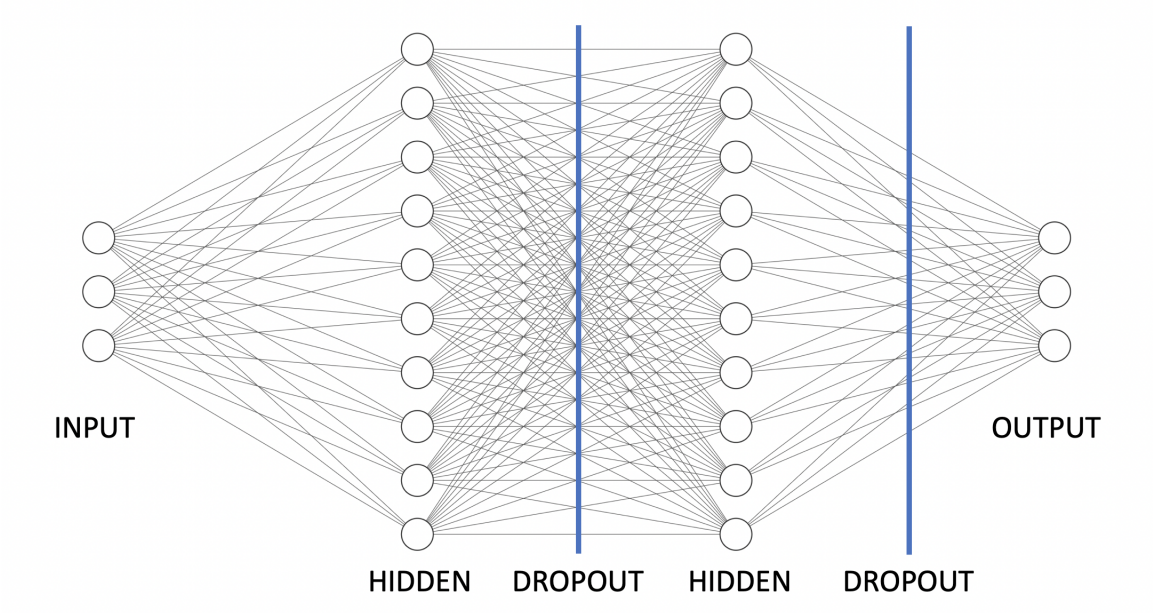

Figure 5-6: Neural Network Architecture

5.5.4 Hyper Parameter Tuning

To choose the best parameters, we run a grid search with the parameters in table to determine the best performing network on the hole prediction problem. The grid search was performed with a training size of 31396 and validation size of 7849 with parameter values listed in Table 5.3. The results of the grid search across various parameters can be seen in Figures B-1 through B-4. The best performing parameters based on validation mean absolute error can be found in Table 5.3 as well.

Symbol	Meaning	Range	Chosen value
N	Neurons in hidden	[8, 16, 32, 64]	64
d	Dropout rate	[0.1, 0.3, 0.5, 0.7]	0.1
λ	Regularization rate	[0.1, 0.01, 0.001, 0.0001]	0.0001
μ	Learning rate	[0.0001, 0.001, 0.01]	0.0001

Table 5.3: Neural Network Hyperparameters

5.6 Automatic Instrument Designer

With the trained models described in Section 5.5, we now have the ability to automatically generate instruments with the user's sound preferences much faster than an exhaustive search with simulation could. The user provides a desired base pitch (all tone holes closed) and overtone spectrum profile. The trained network for predicting shape from audio then outputs the suggested spline curve to use. The user can also supply a list of desired pitches and the network will predict suggested hole locations. The hole locations may give just an estimate, so the hole location can be refined with a brute force search by perturbing the hole location and running the actual FDTD simulation. From the spline curve and hole locations, a 3D voxel grid can be generated, which can then be processed by 3D modelling programs for 3D printing.

Chapter 6

Results

6.1 Physical Accuracy of 3D FDTD Simulation

While we do not perform physical experiments to determine the accuracy of the 3D FDTD code, we do perform a sanity check calculation based on theoretical results. Given a tube closed on one end, the fundamental frequency can be approximated as:

$$f = \frac{C_s}{4(L + 0.61r)} \tag{6.1}$$

where C_s is the speed of sound, L is the length of the instrument, and r is the radius of the bore. We simulate an instrument with a rectangular prism shape of length 0.105 and radius 3. The simulated audio has a pitch of 810 Hz while the theoretical result reports 812 Hz, a deviation of only 4 cents. Moreover the spectrogram of the sound as see in Figure 6-1 indicates that the odd harmonics are prominent, as predicted by the theoretical for tubes with one closed end.

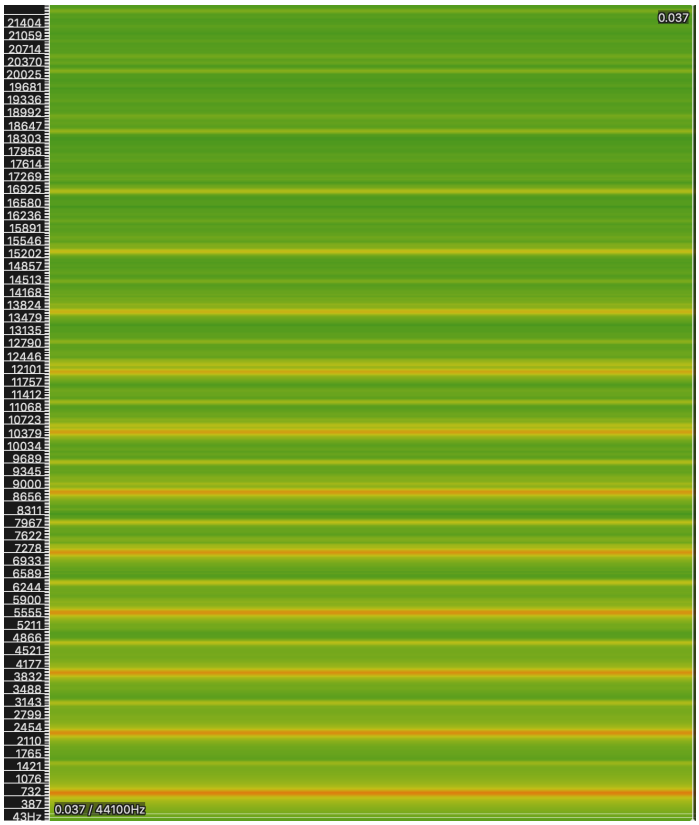

Figure 6-1: Spectrogram of Simulated Tube Instrument. Notice that the odd harmonics are prominent, as expected from a closed tube instrument.

6.2 Speed of 3D FDTD Simulation

For a simulation grid of 32 x 32 x 128 and 30000 iterations, which generates 0.05s of audio, our CUDA implementation of FDTD completes in roughly 12 seconds on a GTX 1080 Ti. In comparison, our CPU based implementation, albeit not optimized, takes around 1 hour. Our CUDA based method yields around a 300x speedup from a naive CPU implementation. 0.05s is enough audio to get a fairly good idea of what the sound will be like as a steady state will likely be already reached. 12 seconds is short enough that testing different shapes can feel somewhat interactive to the user.

6.3 Performance of Audio to Shape Prediction

We assess the performance of our neural network that predicts the spline points given an audio feature vector in two ways. First we compute the mean absolute error of the predicted spline points against the actual spline points that generated that audio vector. The mean absolute error of the training and validation sets over epochs can be seen in Figure 6-2. The settings for the network are the same as those chosen in Table 5.3 and the training set and test set are of size 26653 and 6664 respectively.

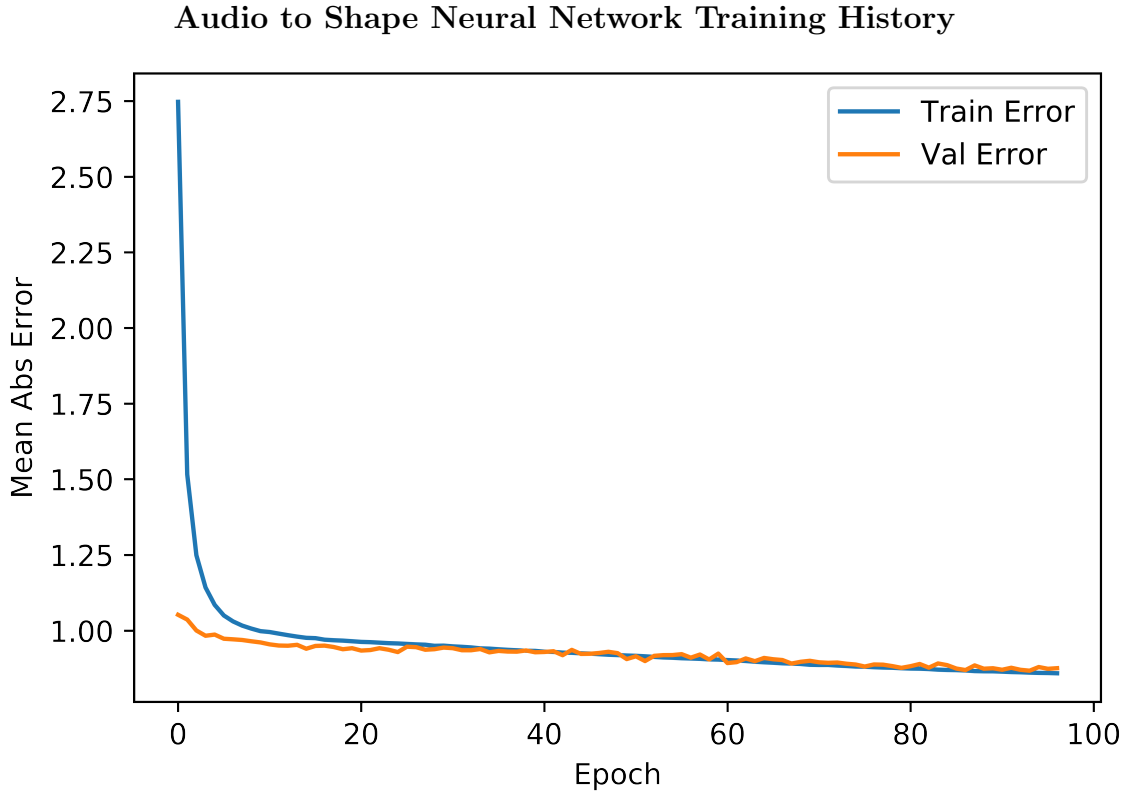

Figure 6-2: Error Plots of Audio to Shape Neural Network

The final validation error is 0.8757, which can be interpreted as the average height difference for each spline point being 0.8757. Given that the radius spans from R_{min} to R_{max}, in our case a range of 5, this error is quite high. However, we primarily care not that the shapes match, but that the audio they generate match. To evaluate whether or not the actual sounds generated are similar, we take our predicted spline points and run our FDTD algorithm to generate the ground truth audio. We compare

the audio features extracted from these audio files with the desired audio features. Table 6.1 shows the mean absolute error for the pitch in cents, and for the first N_o overtones. The mean absolute error is also shown for a random guess where values are randomly sampled from the training test distribution.

Symbol	**Meaning**	**MAE**	**MAE Guess**
p_0	Pitch in MIDI	1.11	3.39
h_1	Overtone 1 Rel. Amp	0.011	0.023
h_2	Overtone 2 Rel. Amp	0.113	0.241
h_3	Overtone 3 Rel. Amp	0.064	0.076
h_4	Overtone 4 Rel. Amp	0.098	0.167
h_5	Overtone 5 Rel. Amp	0.030	0.040
h_6	Overtone 6 Rel. Amp	0.073	0.089
h_7	Overtone 7 Rel. Amp	0.042	0.052
h_8	Overtone 8 Rel. Amp	0.041	0.049
h_9	Overtone 9 Rel. Amp	0.045	0.047
h_10	Overtone 10 Rel. Amp	0.039	0.044
h_11	Overtone 11 Rel. Amp	0.033	0.039
h_12	Overtone 12 Rel. Amp	0.026	0.023
h_13	Overtone 13 Rel. Amp	0.024	0.023

Table 6.1: Errors for Predicted Pitch and Overtones, compared with error of random guess. MAE is mean average error

As seen in Table 6.1, the mean absolute error in fundamental pitch is around 1.11 semitones, which is not great but reasonable. For the overtones, we see that for the first few, our network produces half the error of that of a random guess. As the overtone number increases however, past the 6th overtone, our method is no better than a random guess. The lower overtones are more prominent making this in some way more acceptable. Given the highly nonlinear relationship between shape and sound, it is not surprising that our relatively simple network cannot learn the relationship with high precision. Nevertheless it can provide a reasonable starting point for the user.

6.4 Accuracy of Tone Hole Placement

For tone hole placement, we similarly assess accuracy by comparing the predicted tone hole location and the actual tone hole location in our test set. The settings for the network are also the same as those chosen in Table 5.3. The training and validation error over epochs can be seen in Fig 6-3. The settings for the network are the same as those chosen in Table 5.3 and the training set and test set are of size 55873 and 13969 respectively.

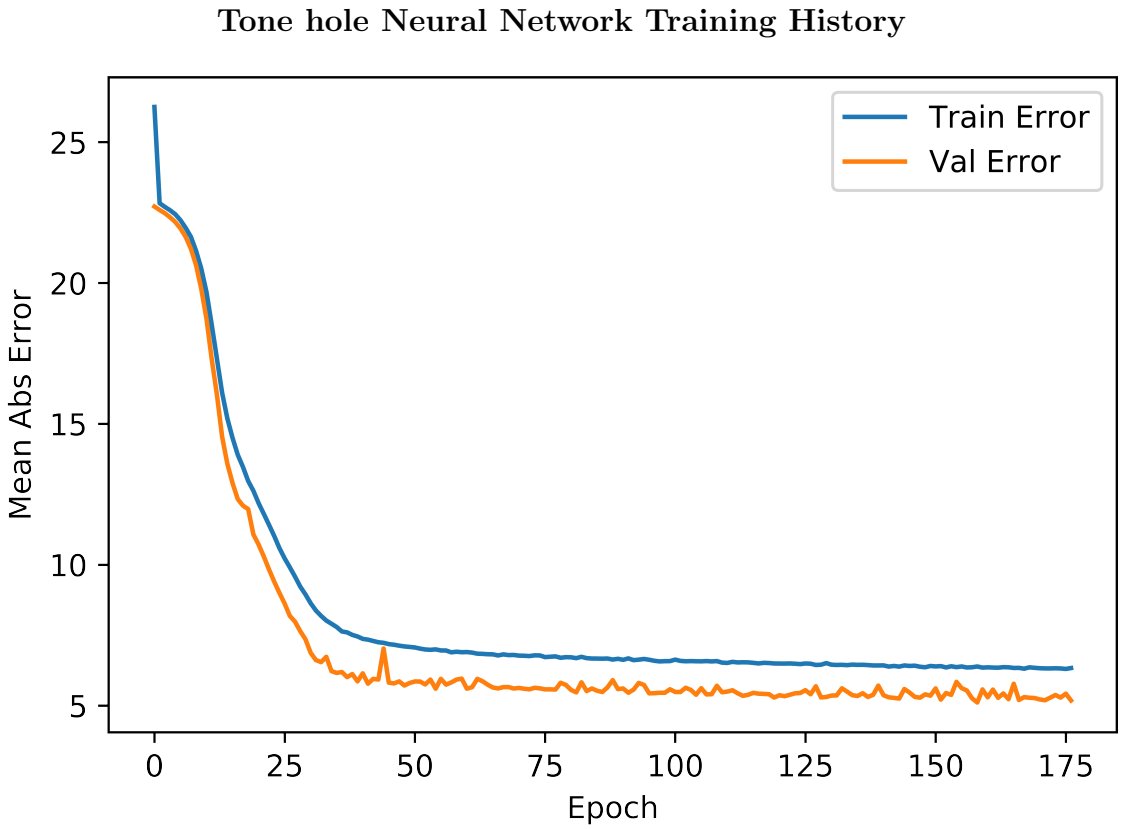

Figure 6-3: Error Plots of Tone Hole Placement Neural Network

The mean absolute error in tone hole position is 5.18. For a bore length $L = 100$, this is much better than chance, but certainly could be improved. However even if the tone hole predictor is not super accurate, it does provide a good initial approximation. A brute force depth first search starting from the initial guess tone hole location should be able to quickly find the optimal hole location.

6.5 Example Printed Instrument

Although we did not perform extensive physical testing of printed 3D models, we did print an example prototype. Using the GUI and random spline generator, we generated a shape and then determined the placement of the holes for the pitches of the pentatonic scale. From the voxel grid, we generate a STL file. In order to facilitate connection with a mouthpiece, we replace the beginning section with a cylinder of matching length.

We 3D print the instrument in three parts: the mouthpiece connector, the top half of the bore, and the bottom half of the bore, see Figure 6-4. Once printed we use epoxy and superglue to attach the parts. See Figure 6-5 for the printed part.

Figure 6-4: CAD model of prototype instrument

We attach a clarinet mouthpiece to the 3D printed bore. Unfortunately, as the clarinet mouthpiece itself is almost the same size of the bore, which only spans 10cm, the effective tube length of the instrument changes significantly, making our simulation results invalid. Nevertheless, we end up with a quite pleasing sounding instrument, recordings can be found at `http://www.larryoatmeal.com/clarinet.wav`.

With further time we would print prototypes generated by our automatic instrument designer and test their performance.

Figure 6-5: Printed 3D Instrument, clarinet mouthpiece attached.

Chapter 7

Conclusion and Future work

In conclusion, our main contributions in this work are a fast implementation of a 3D single reed instrument simulator, a tool for visualizing and exploring shapes, as well as machine learning models for generating instrument bore shapes and tone holes that attempt to match the user's desired sound. There are many possibilities for further extending this work, including

Performing physical experiments to verify simulation: By 3D printing several shapes and playing them, the difference between the FDTD simulation and reality can be better understood. Moreover, given enough examples, it might be possible to learn the discrepancies between the simulated and actual audio.

Further optimizing CUDA to get closer to real-time: There are several additional optimizations that can be made to our CUDA code, including experimenting with different thread block sizes, caching more values in shared memory, and multi GPU parallelism.

Exploring more complex shapes than simple spline-based extrusions: In this work, we limit ourselves to essentially a 1D design domain. We are not taking full advantage of all the additional degrees of freedom 3D provides.

Using more sophisticated network models: More sophisticated networks than our simple fully connected network can be experimented with, especially if our shape representation moves from 1D to higher dimensions. Image based networks such as CNNs could be explored. The performance of our models can certainly be improved,

and better models plus more data should help immensely.

Optimize for combinatorial pitch holes: In real instruments, the desired pitches are achieved through a combination of open and closed holes. Solving for example how to assign 12 pitches to five holes and a fingering guide for each pitch would be a much harder problem.

Appendix A

Tables

Symbol	**Meaning**	**Range**
Δ_s	Spatial resolution	0.00105m
Δ_s	Temporal resolution	1.74428μs
ρ	Mean density	1.1760 kg/m^3
C_s	Speed of sound	347.23 m/s
W_j	effective jet width	1.2×10^{-2} m
H_r	max reed displacement	6×10^{-4} m
K_r	reed stiffness	8×10^6 N/m
p_{mouth}	Pressure from mouth on reed	3500P

Table A.1: 3D FDTD Simulation Constants, see [Allen and Raghuvanshi, 2015] for details

Appendix B

Figures

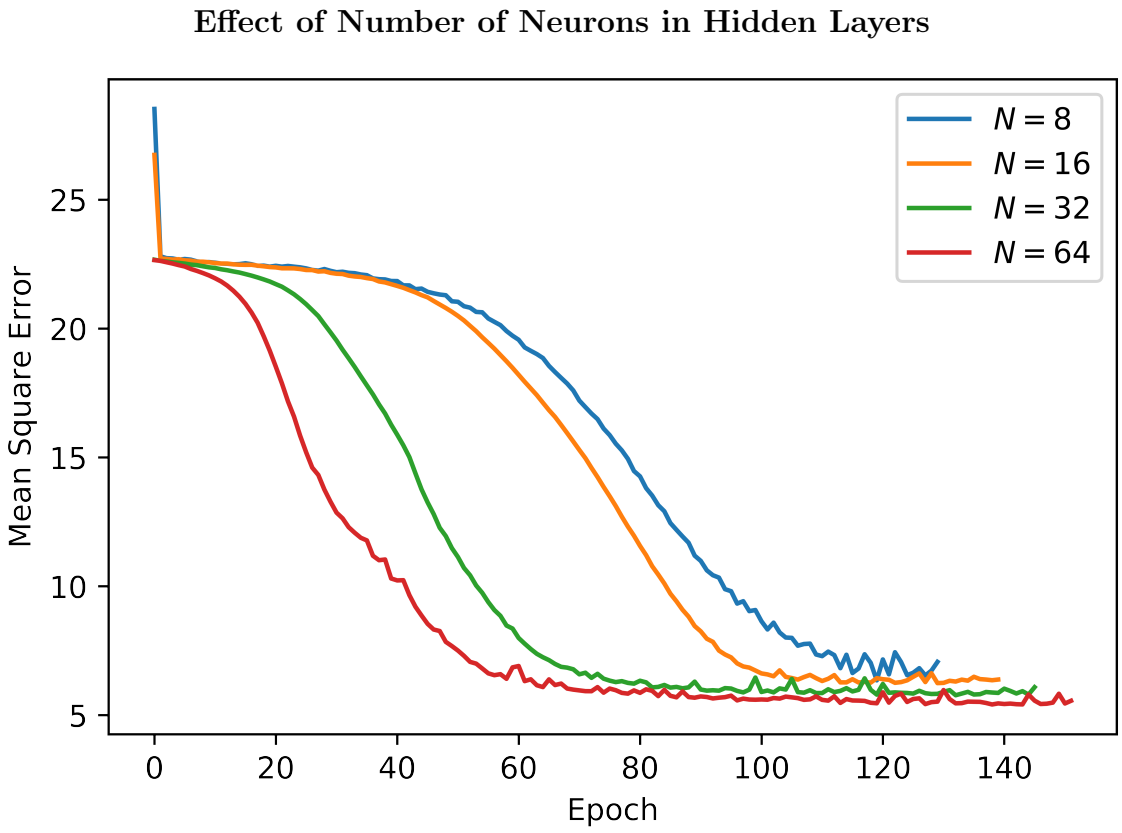

Figure B-1: Validation mean square error training history with settings as described in Section 5.5.4, with the number of neurons in hidden layers varied.

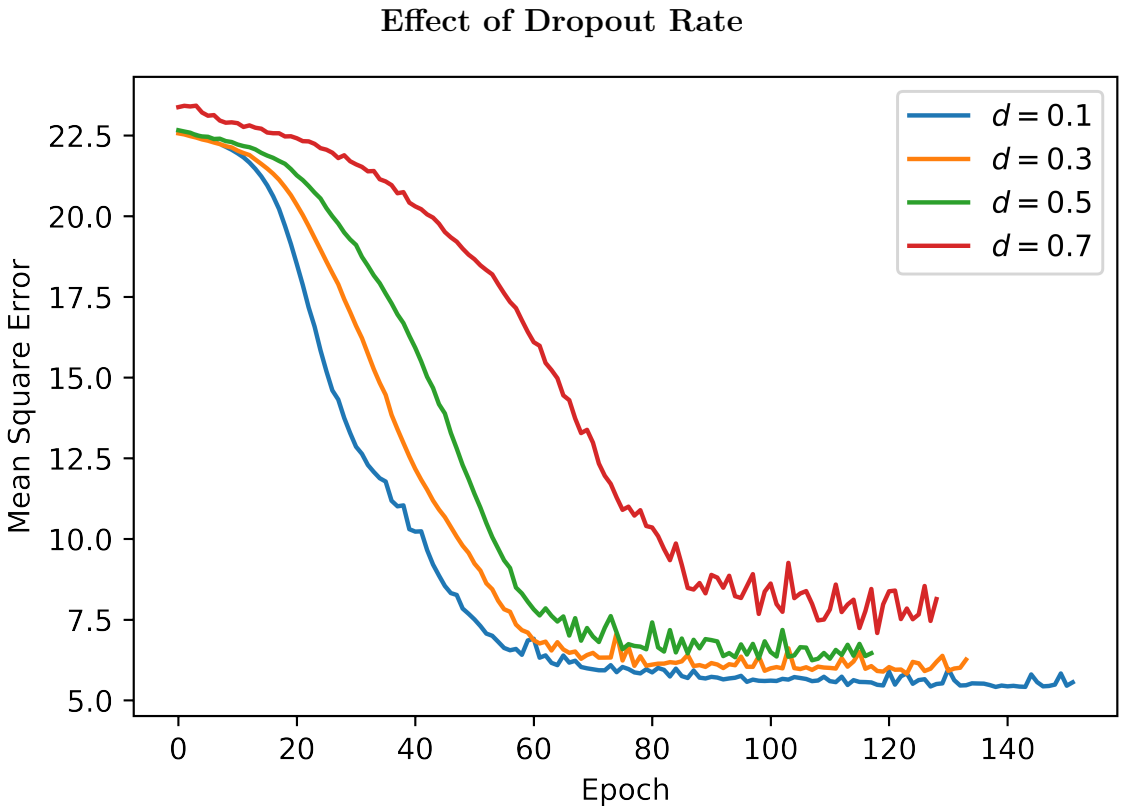

Figure B-2: Validation mean square error training history with settings as described in Section 5.5.4, with dropout rate varied.

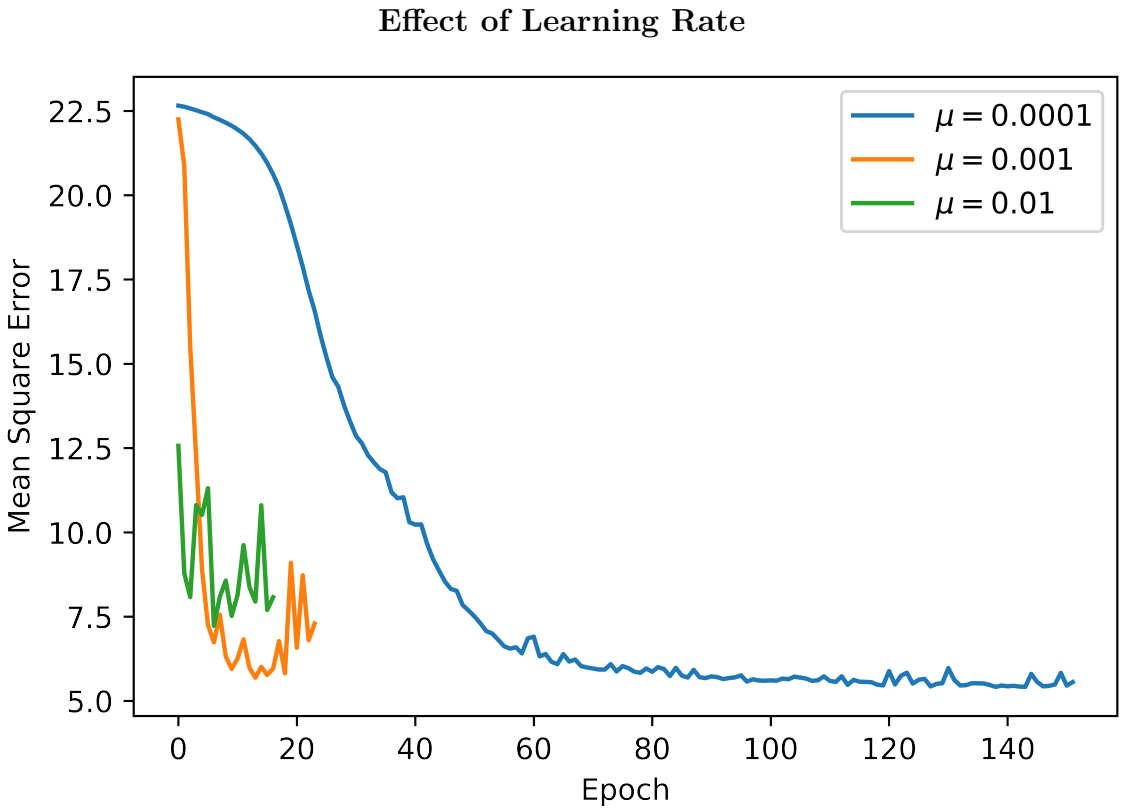

Figure B-3: Validation mean square error training history with settings as described in Section 5.5.4, with learning rate varied.

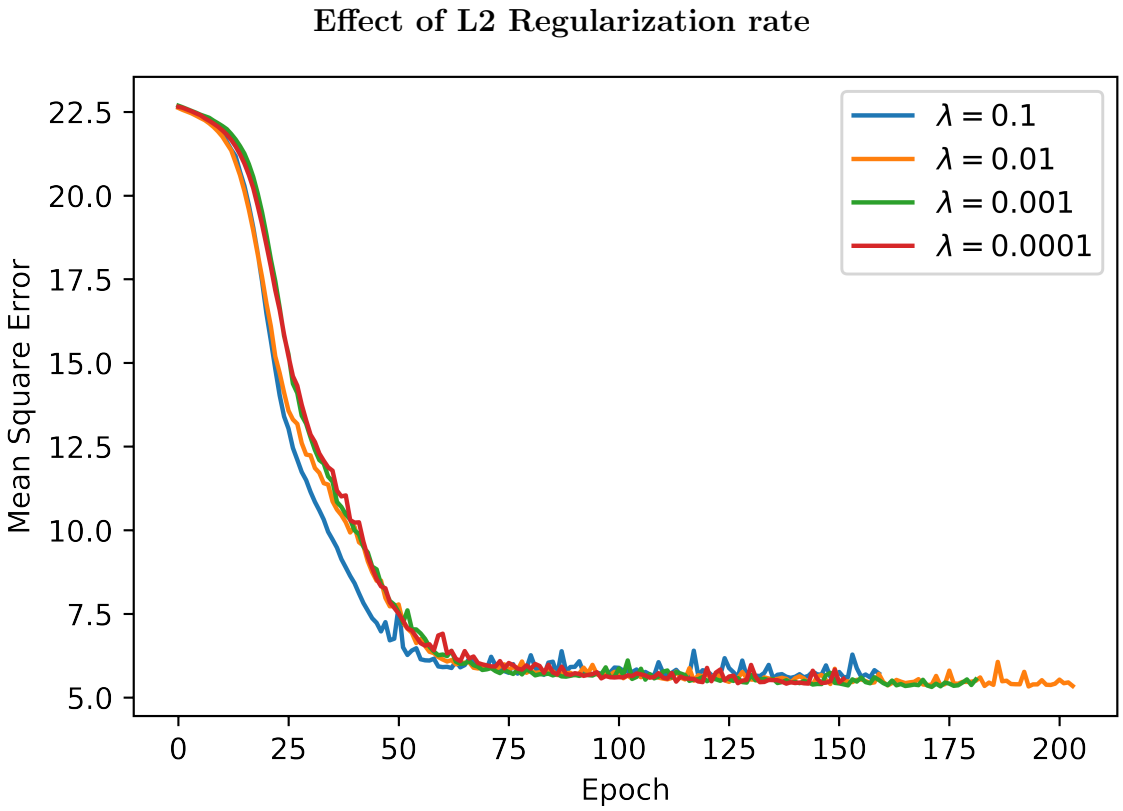

Figure B-4: Validation mean square error training history with settings as described in Section 5.5.4, with L2 regularization rate varied.

Bibliography

[Allen and Raghuvanshi, 2015] Allen, A. and Raghuvanshi, N. (2015). Aerophones in flatland: Interactive wave simulation of wind instruments. *ACM Transactions on Graphics (TOG)*, 34(4):134.

[Arnela and Guasch, 2014] Arnela, M. and Guasch, O. (2014). Two-dimensional vocal tracts with three-dimensional behavior in the numerical generation of vowels. *The Journal of the Acoustical Society of America*, 135(1):369–379.

[Bharaj et al., 2015] Bharaj, G., Levin, D. I., Tompkin, J., Fei, Y., Pfister, H., Matusik, W., and Zheng, C. (2015). Computational design of metallophone contact sounds. *ACM Transactions on Graphics (TOG)*, 34(6):223.

[Cosmo et al., 2018] Cosmo, L., Panine, M., Rampini, A., Ovsjanikov, M., Bronstein, M. M., and Rodolà, E. (2018). Isospectralization, or how to hear shape, style, and correspondence. *arXiv preprint arXiv:1811.11465*.

[Li et al., 2016] Li, D., Levin, D. I., Matusik, W., and Zheng, C. (2016). Acoustic voxels: computational optimization of modular acoustic filters. *ACM Transactions on Graphics (TOG)*, 35(4):88.

[McIntyre et al., 1983] McIntyre, M. E., Schumacher, R. T., and Woodhouse, J. (1983). On the oscillations of musical instruments. *The Journal of the Acoustical Society of America*, 74(5):1325–1345.

[Micikevicius, 2009] Micikevicius, P. (2009). 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 79–84. ACM.

[Smith, 1986] Smith, J. O. (1986). Efficient simulation of the reed-bore and bowstring mechanisms.

[Takemoto et al., 2010] Takemoto, H., Mokhtari, P., and Kitamura, T. (2010). Acoustic analysis of the vocal tract during vowel production by finite-difference time-domain method. *The Journal of the Acoustical Society of America*, 128(6):3724–3738.

[Umetani et al., 2016] Umetani, N., Panotopoulou, A., Schmidt, R., and Whiting, E. (2016). Printone: interactive resonance simulation for free-form print-wind instrument design. *ACM Transactions on Graphics (TOG)*, 35(6):184.

[Webb and Bilbao, 2011] Webb, C. J. and Bilbao, S. (2011). Computing room acoustics with cuda-3d fdtd schemes with boundary losses and viscosity. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 317–320. IEEE.