MONASH University
Engineering

Standalone Physical Modelling Synthesizer

Project Report

Valerian McCaskill
vmmcc2@student.monash.edu
26024608

Supervisor: Prof. Arthur Lowery
ECE4095
20/10/2018

Contents

1. Significant Contributions

- Implemented 5 different physical models on an embedded device, capable of real-time performance at 12 note polyphony including:
    - 1D waveguide model
    - Karplus Strong
    - Banded waveguide model
    - Woodwind model
    - 2D waveguide mesh model
- Improved upon previously existing models in literature:
    - Designing and implementing a modified linear interpolation method for fractional delay lines
    - Designed and implemented downsampling functions that allow for longer delay lines, overcoming memory restrictions
- While implementing these models, I also:
    - Implemented a wavetable synthesizer
    - Designed commuted impulse responses from audio recordings of real instruments
    - Implemented an all-pass filter, state variable filter, DC blocker filter, and multiple waveguide filters
    - Created lookup tables to make faster versions of standard C functions
- Implemented a midi optocoupler circuit and interfaced a midi device with the STM32F4
- Designed and built a basic enclosure, power supply circuit and Veroboard interfacing circuits, and created graphics and interface for the attached screen

2. Poster

MONASH University Engineering

Department of Electrical and Computer Systems Engineering

ECE4095 Final Year Project 2018

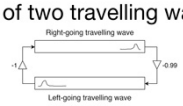
Valerian McCaskill

Standalone Physical Modelling Synthesizer

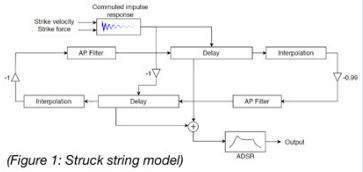
Supervisor: Prof. Arthur Lowery

A playable instrument capable of simulating strings, bars, tubes and woodwind instruments with up to 12 note polyphony in real-time

A string can be modelled as a combination of two travelling waves

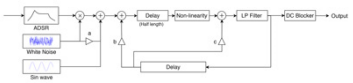


These travelling waves can be simulated as delay lines in software



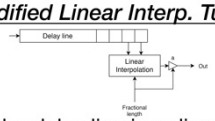
(Figure 1: Struck string model)

The same approach can be used for any instrument involving waves



(Figure 2: Flute model)

Modified Linear Interp. Tuning

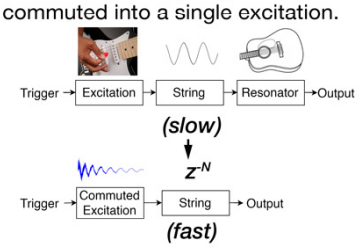


As the delay line is a discrete length, a variation of linear interpolation can be used to fine tune the string length, while retaining the amplitude of the note.

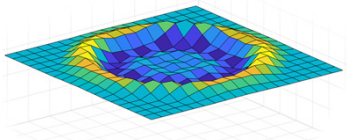
1D Commuted Waveguide model

(for strings, tubes and bars)

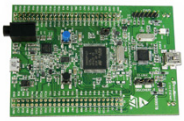
As the string and body of the instrument are approximately linear and time-invariant, they can be commuted into a single excitation.



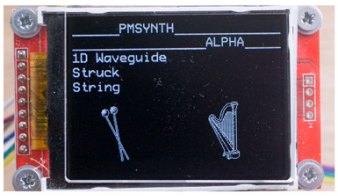
2D Transmission Line model findings



Large meshes were not fast enough to compute in real time, so a ***banded waveguide*** approach was used to model plates and bars.



Built on the low cost 168MHz STM32F4-DISC1 ARM board



Interactive display allows the user to select which model is active, and change options.

Modify parameters such as stiffness and strike position in real-time.

Listen to sound demos and view source code at:
github.com/valmcc/pmsynth

3. Executive Summary

This report details the creation and development of a physical modelling synthesizer, and the models implemented on it.

I approached this task by researching papers about physical modelling and implementing aspects of these papers into my models. I also researched real time audio processing and DSP to implement these models on the microcontroller. I then had to optimise each model to the point that it was real-time and playable as a musical instrument.

The main outcome of this project is that I now have a working physical modelling synthesizer that is usable in a performance context. The synthesizer is capable of the following models:

- Commuted 1D Waveguide (Strings and tubes)
- Karplus-Strong (Plucked string)
- Flute (variation of the 1-D waveguide model)
- Banded Waveguide (Bars – xylophone and marimbas)

By creating this synthesizer, I have discovered that a 2D mesh is not viable as a tonal musical instrument on embedded hardware at this time. However, many existing physical models with improvements can be implemented in real time, at up to 12 note polyphony.

4. Introduction

4.1 Aims and Objectives

This project aims to design and create a playable synthesizer capable of synthesizing physical models in real-time, with the option to modify parameters of the models as the musician plays.

Physical models will then be developed and implemented on the device. This project intends to bridge the gap between prior hardware physical modelling synths and todays software synths.

This will allow the musician to dynamically alter the sound with the use of adjustable parameters in ways that are not possible in more traditional forms of sound synthesis.

An additional goal is to attempt to create a playable 2D mesh-based waveguide model based on transmission line modelling, to synthesise the sounds of a metal plate. This will be the first implementation of a two-dimensional waveguide mesh on embedded hardware, to the best of my knowledge.

The project also aims to achieve this on an embedded capable of being used ‘standalone’ – that is without being connected to an external device or computer.

4.2 Motivation

My motivation for attempting this project stems from my love of audio engineering and synthesis. I personally produce a lot of electronic music in my spare time and have amassed a large collection of software synthesizers (I have no room in my tiny apartment for any more than one hardware synth!).

If you are interested in hearing some of my music (a lot of it integrates these physical modelling algorithms) you can listen to it here: https://soundcloud.com/valer

Figure 4.1 Chromaphone 2 by Applied Acoustics Systems

I was first exposed to physical modelling by a software synthesizer called *Chromaphone 2* which is known for modelling xylophones and marimbas accurately. Despite the appearance of a few physical modelling software synths in the recent years, physical modelling has nowhere near the popularity of other forms of synthesis such as wavetable synths, and I wanted to see why this was the case.

I also play synthesizer in a band called Sleeping Lessons, and plan on using the synthesizer in live shows.

4.3 Physical Modelling Synthesis

Physical modelling synthesis is a form of sound synthesis that generates waveforms using mathematical models designed to simulate real life musical instruments. This typically is achieved by a recursive algorithm that applies a modification process to a delay line of samples [1].

Digital waveguide modelling was developed in the 1980s by Julius Smith [2]. It has a high level of popularity in the past as it allows for the physical modelling of various musical instruments at relatively low computational cost.

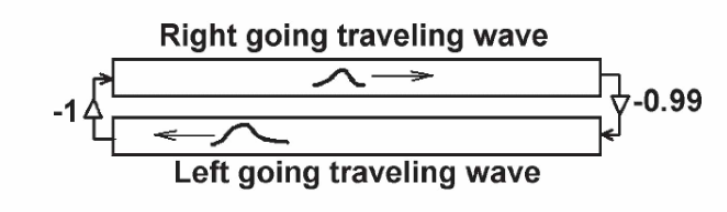


Figure 4.2 Waveguide string [3]

For instruments that can be modelled by the one-dimensional wave equation, such as strings or wind instruments, a digital waveguide approach is suitable. The wave equation can discretise into a form of delay line, which simulates the propagation of a travelling wave.

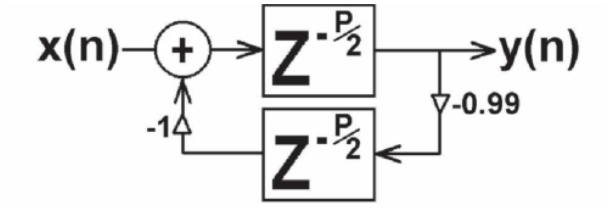


Figure 4.3 Filter view of waveguide string [3]

The characteristics of the sound produced by a physical modelling algorithm are greatly affected by the initial contents of the delay line, and the modification processes that are applied to the delay line. These must be designed in a way as to reproduce the acoustic properties of the desired instrument as closely as possible.

Further expanding upon this idea, two-dimensional waveguides can be created, where delays from each 'junction' extend in four directions instead of just two.

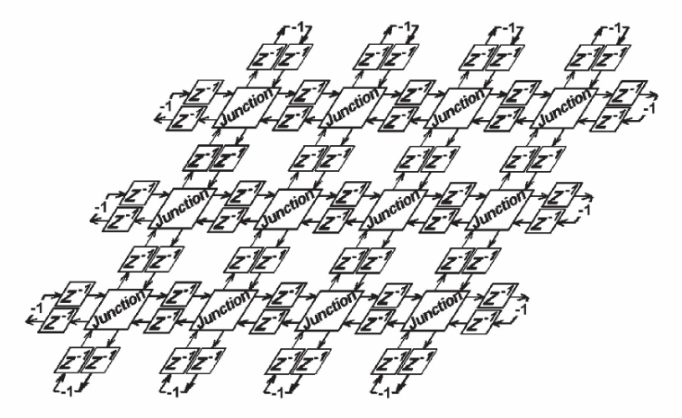


Figure 4.4 Two-dimensional waveguide mesh [3]

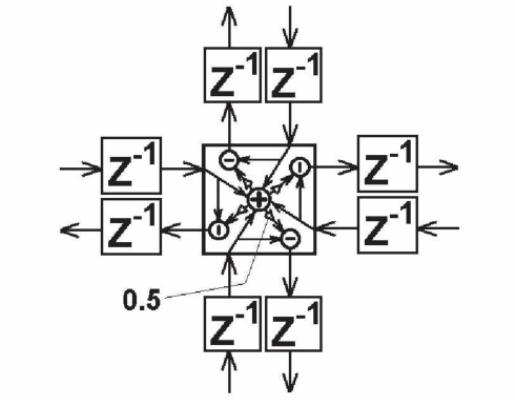


Figure 4.5 Close-up of mesh junction [3]

Upon exciting a section of the delay line with an input (a strike, pluck or blow), the waveguide resonates at a particular frequency. The output audio can be extracted from the mesh by recording the values of the waveguide at certain points of the mesh, much like a pickup on a guitar.

In this project, both models have been implemented with improvements. Some of these improvements include scattering junctions, linear interpolation for the output, different boundary conditions and alternative configurations of these delay lines.

5. Prior work

This section aims to demonstrate the background of prior work in the field of physical modelling, both on the commercial and academic side.

5.1 Standalone Physical Modelling Synthesizers

Standalone refers to a piece of hardware or software able to operate independently of other hardware of software. In the case of synthesizers, it refers to a synth that does not need an external computer to be present to create its sound.

Waveguide synthesis has been seen in a few synths in the past, with one of the most well-known being the Yamaha VL-1 released in 1993.

Figure 5.5 Yamaha VL-1 hardware synth

While not being considered a commercial success, the VL-1 contained impressive simulations of wind, brass and strings instruments with the use of one-dimensional waveguide technology. After the release of the VL-1, most of the synth development intended to recreate real-life instruments expanded towards sample-based methods.

Figure 5.6 Korg OASYS synth workstation

Later endeavours such as the Korg OASYS in 2005 attempted to improve on prior physical modelling synthesizers. This device was however very expensive; at $8000 USD, and bulky, requiring a Pentium 4 processor to run it. It still had not extended past one-dimensional waveguides.

Figure 5.7 AAS StringStudio software synth

Since then, developments in physical modelling by means of waveguides have been sparse. Only recently has more work been done in the form of software synthesizers such as Applied Acoustic System's recent products such as StringStudio.

This project intends to bridge the gap between prior hardware physical modelling synths and todays software synths, with the use of a powerful 32-bit ARM processor. This will be to my knowledge, the first implementation of a two-dimensional waveguide mesh on embedded hardware.

Despite all the developments in the past, physical modelling synthesis has yet to reach the level of success of more traditional wavetable and FM synthesis. However, with the increasing level of computing power seen today, it can be expected to make a comeback.

5.2 Review of Previous Literature

5.2.1 Introduction

Physical modelling is a form of synthesis that generates musical tones with the use of mathematical models that simulate the properties of real-world instruments.

Over the years, physical modelling has captured the minds and imaginations of physicists and musicians alike. In the recent years, with the invention of faster computers, physical modelling has become viable to perform in real-time.

Digital waveguides are a form of physical modelling synthesis that utilises delay lines to simulate the propagation of waves through musical instruments, including strings and woodwind. This method was initially discovered by discretising the wave equation, a method of modelling waves in strings mathematically.

Over time, additional improvements were added to the waveguide model, adding more dimensions, more material properties to modify, and simplifications to allow it to run in real-time.

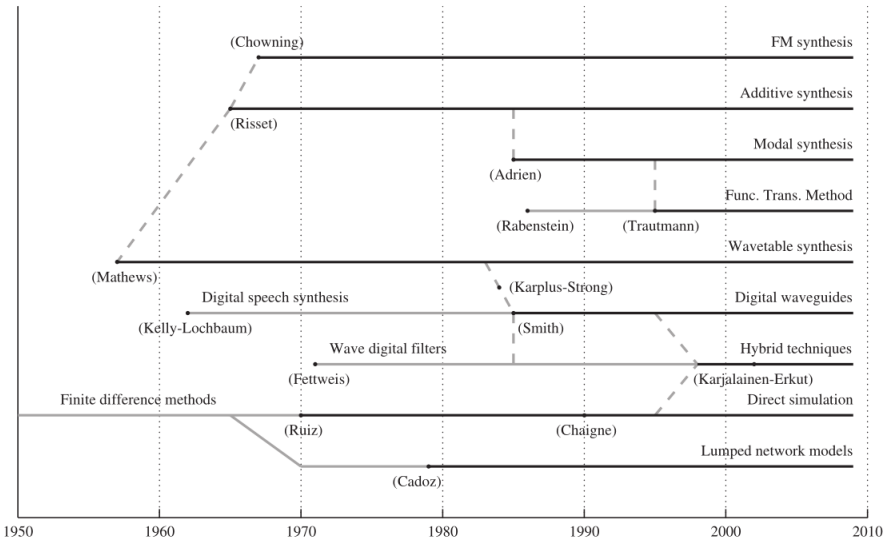


Figure 5.1 History of various forms of audio synthesis [4]

5.2.2 Early Virtualisation of Strings

In the past, the vibrating string was always a point of contention between musicians and physicists.

Marsenne, in 1636, back when the theory of sound was only time domain, noted that he could hear overtones in a string despite only visibly seeing the fundamental frequency [5].

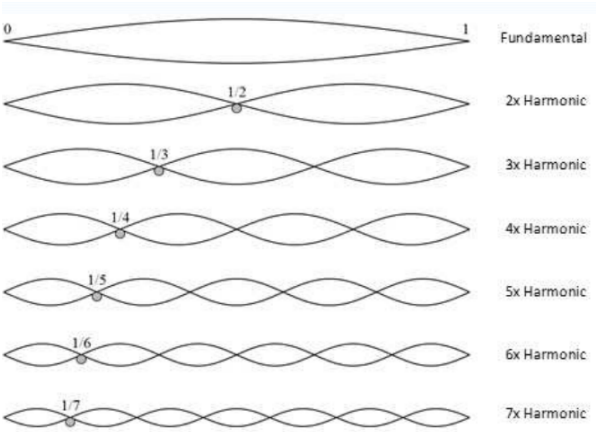


Figure 5.2 Illustration of nodes on a string

Later, in 1701, Sauveur coins the term “harmonics”, based on string playing musicians being able to play “harmonious” sounds by lightly dampening the fundamental [6]. He coins the term “node” which refers to a point on the string that doesn’t move.

Taylor, famous for the Taylor series, derives an equation for the fundamental frequency of the string in 1713:

$$f = \frac{\sqrt{\frac{K}{\epsilon}}}{2L} \tag{5.1}$$

Where K = string tension, ϵ = mass density, L = string length. He incorrectly states that restoring force on the string is proportional to string curvature. Furthermore, he incorrectly thought everything, but the fundamental was evanescent [7].

Rameau in 1726, a composer interested in overtones anticipated a spectrum analyser model of hearing [8]. Later, in 1727, Bernoulli studied the mass-loaded ideal massless string and also makes the same mistakes as Taylor, and overlooked higher modes of vibration, considering only the fundamental [7].

Bernoulli's son in 1742 studies elastic bands with inharmonic modes, and comes up with the idea of superposition, with the following quote:

"Both sounds exist at once and are very distinctly perceived.... This is no wonder, since neither oscillation helps or hinder the other; indeed, when the band is curved by reason of one oscillation, it may always be considered as straight in respect to another, oscillation, since the oscillations are virtually infinitely small. Therefore, oscillations of any kind may occur, whether the band be destitute of all other oscillation or executing others at the same time. In free bands, whose oscillations we shall now examine, I have often perceived three or four sounds at the same time. [9]"

This observation is then turned into an equation (the wave equation for a string) by D'Alembert in 1747 [10].

$$y_{D'AI} = y(x,t) = y_l\left(t + \frac{x}{c}\right) + y_r(t - \frac{x}{c}) \tag{5.2}$$

This equation states that any vibration of a string can be expressed as a combination of two travelling waves; one travelling in the left direction (y_l), and one travelling in the right direction (y_r). The equation also states that the rate of the propagation of this wave is at the velocity c along the string.

This discovery was "obvious" to string playing musicians like Bernoulli, but not Taylor and Euler who were less familiar with musical instruments.

The wave equation (5.2) stated above is then used by Karplus and Strong hundreds of years later to develop the digital waveguide model for a string. This same equation is the basis for all of the models on the synthesizer.

5.2.3 The Digital Waveguide

5.2.3.1 Delay Lines for Strings and Tubes

Digital waveguide synthesis; a form of modelling strings and other instruments with the use of delay lines, was born in 1983 by Karplus and Strong [11]. They noted that based on a discretised form if the wave equation by D'Alembert (Equation 5.2), they could model a string as a combination of white noise and a delay line, attached to an averaging filter.

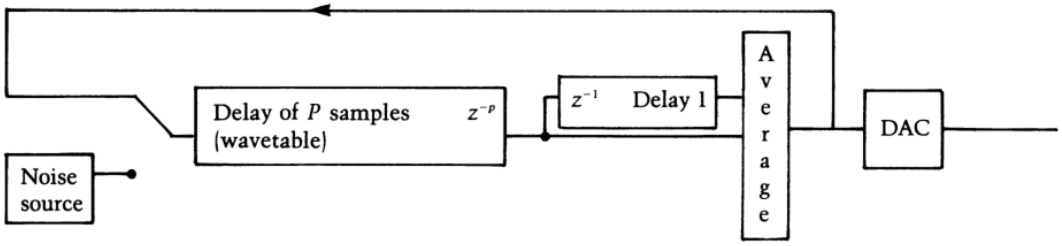


Figure 5.3 Block diagram of Karplus-Strong Plucked String [11]

The model they proposed sounded like a plucked string and could be computed! It was not in tune however and lacked modulation options.

The extended Karplus-Strong model was proposed in 1983 by Jaffe and Smith [12]. This model aimed to make the Karplus-Strong model more 'musical' by tuning the string, modifying the decay of the string, simulating a moving pick, and proposing using all-pass filters for stiffness. Over the next years, multiple improvements to this model were proposed, usually adapting findings by Morse [13], who wrote about the underlying physics of musical instruments.

One of the first documented real-time implementations was in 1991 with the "real-time guitar" on a floating point signal processor [14]. While still rudimentary, the model hints at the possibilities of digital waveguide techniques. More real-time implementations appear around this time also, with clarinets [15] and more stringed instruments [16-18].

A tube model was proposed by Cook in 1991, to be used in modelling a vocal tract. It uses the same equation as the other models, but was adapted for air pressure and not string vibration [19].

Rapid developments towards commuted synthesis were made in 1995 which led to models that sounded highly realistic [20, 21]. This was one of the biggest improvements to the models to date and it allowed realistic models to be played in real-time.

Many of these techniques would be patented by Yamaha, starting at around the year 1993, which could have been a contributing factor into why interest into development of new models began to wane after 1993.

5.2.3.2 The 2D Waveguide Mesh

The 2D waveguide mesh was proposed by Smith in 1993, to simulate plates and membranes, however is nowhere near real-time. This method utilises the same delay lines as in previous instrument models, however is constructed like a transmission line matrix [22]. Digital waveguide networks, or combinations of waveguide meshes are developed by Smith later also, however is also not in real-time [23].

More testing on the digital waveguide mesh is performed in 2001 by Murphy, who states that computational power is still not fast enough for a real-time mesh [24]. Some smaller sized mesh models to simulate drums are created, to demonstrate that the mesh is viable for non-tonal applications in real-time [25-27].

5.2.4 Recent Model Improvements

Duyne discusses methods for tuning delay lines, however dismisses linear interpolation due to the attenuation that varies due to fractional lengths [28]. Linear interpolation is the method I intend to improve upon with my work.

More practical information is released as the Yamaha patents expire; Cook publishes [3] which gives a practical overview of physical modelling and extensions in 2002. More advanced versions of the speech synthesizer based off work the vocal tract model are created [29].

Furthermore, in 2012, major extensions of the guitar model are published, which make use of increased computational power [30]. Similar extensions are made to piano models [31].

More recently, fully 3D meshes are approaching real-time with the use of modal techniques. This is beyond the scope of this project however [32]. Other sources state that real-time meshes are still not viable without a lot of extra research [33].

5.2.5 Digital Audio Improvements

Digital audio has come a long way, and now frameworks and development platforms for PC based development are relatively mature and well documented [34, 35]. Embedded audio development is, however not well documented in literature.

5.2.6 Discussion

Since 1993, not a lot of practical implementations of physical modelling algorithms have been developed, with most papers discussing improvements to models that lack real-time application. I aim to develop real-time models to remedy this. Furthermore, there seems to be a lack of consensus on the best form of tuning method for waveguides, so I intend to investigate this further. Commuted synthesis appears to be the best suited for my purposes, being a real-time, simplified approach.

As embedded development related to audio is less developed, I aim to implement these models on an embedded device. Embedded devices include more limitations, but also advantages with cost and portability [36].

5.2.7 Conclusion

Since its conception, digital waveguides have seen a few practical implementations, however lag other forms of audio synthesis. Physical modelling remains a field of synthesis that shows huge potential, however this potential is marred by a lack of real-time practical implementations. Therefore, I aim to provide implementations and document the solutions that I come up with to get the models playable in real-time.

6. Theory behind the Discrete Waveguide Model

6.1 The wave equation

The 1D wave equation describes the movement of transverse waves on an ideal string. It has the form:

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2} \tag{6.1}$$

Where $y(x,t)$ represents the transverse displacement of the string at time t and position x [13]. This equation and its extensions form the basis for all the models on the synthesizer.

6.2 The Ideal String

The equation above can be obtained by imagining a string under tension, with a length density of ρ grams/meter. If the string is displaced by a small amount *y* from its rest position at *y = 0*, we can compute the force on each little section of string.

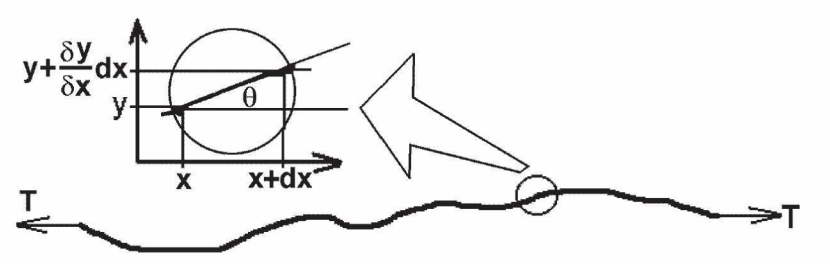


Figure 6.1 An ideal string under tension T

Ignoring gravity, the force on the string in the y direction depends on the tension component, which acts either upward or downward. This component is given by $T \sin\theta$, where θ is the angle shown in Figure 7.1.

The difference in force between two adjacent points x and $x + dx$ as a function of the force along the string is therefore:

$$df_y = (T \sin\theta)_{x+dx} - (T \sin\theta)_x \tag{6.2}$$

The Taylor's series for expressing the force at point $x + dx$ as a function of the force at point x on the string is given by:

$$f(x + dx) = f(x) + \left(\frac{\partial f}{\partial x}\right) dx + \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^2}\right) dx^2 + \cdots \tag{6.3}$$

If we assume that dx is very small, we can keep only the first two terms of the Taylor's series in equation, giving us:

$$f(x + dx) - f(x) = \left(\frac{\partial f}{\partial x}\right) dx \tag{6.4}$$

$$or, df_y = \left(\frac{\partial f}{\partial x}\right) dx \tag{6.5}$$

Combining Equations 7.5 and 7.2 gives us:

$$df_y = \left(\frac{\partial (T \sin\theta)}{\partial x}\right) dx \tag{6.6}$$

If θ is small, we can replace it with the approximation $\left(\frac{\partial y}{\partial x}\right)$ (change in y divided by length in x direction as per Figure 7.1)

Now we can obtain:

$$df_y = T\left(\frac{\partial^2 y}{\partial x^2}\right) dx \tag{6.7}$$

Now we can relate this back to physics. Let's use Newton's second law:

$$F = ma, \quad or\ alternatively, \quad df_y = (\rho\, dx)\left(\frac{d^2 y}{dt^2}\right) \tag{6.8}$$

We can set the right sides of Equations 7.8 and 7.7 equal as their right sides are both equal. After rearranging, this gives us:

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2} \tag{6.1}$$

Which is the wave equation for a 1-dimensional string. We have defined c as the speed of wave motion along the string where $c = \left(\frac{T}{\rho}\right)^{1/2}$ (proportional to the square root of string tension divided by the square root of linear string mass density).

6.3 D'Alembert Solution Proof

The D'Alembert solution to Equation 7.1 is:

$$y_{D'AI} = y(x,t) = y_l\left(t + \frac{x}{c}\right) + y_r\left(t - \frac{x}{c}\right) \tag{6.9}$$

To show that this is a solution, we can take the derivative with respect to x twice:

$$d(y_{D'AI})/dx = \left(\frac{1}{c}\right) y_l\left(t + \frac{x}{c}\right) - \left(\frac{1}{c}\right) y_r\left(t - \frac{x}{c}\right)$$

$$d^2(y_{D'AI})/dx^2 = \left(\frac{1}{c^2}\right) y_l\left(t + \frac{x}{c}\right) - \left(\frac{1}{c^2}\right) y_r\left(t - \frac{x}{c}\right)$$

$$d^2(y_{D'AI})/dx^2 = \left(\frac{1}{c^2}\right)\left[y_l\left(t+\frac{x}{c}\right) - y_r\left(t-\frac{x}{c}\right)\right]$$

$$d^2(y_{D'AI})/dx^2 = \frac{y_{D'AI}}{c^2} \tag{6.10}$$

And the derivative with respect to t twice:

$$d(y_{D'AI})/dt = y_l\left(t+\frac{x}{c}\right) - y_r\left(t-\frac{x}{c}\right)$$

$$d^2(y_{D'AI})/dt^2 = y_l\left(t+\frac{x}{c}\right) - y_r\left(t-\frac{x}{c}\right)$$

$$d^2(y_{D'AI})/dt^2 = y_{D'AI} \tag{6.11}$$

Now we can substitute Equation 7.11 into 7.10, giving us:

$$d^2(y_{D'AI})/dx^2 = \frac{d^2(y_{D'AI})/dt^2}{c^2} \tag{6.12}$$

Which is identical to Equation 7.8.

6.4 A Discrete-Length String

Continuing from our solution:

$$y_{D'AI} = y(x,t) = y_l\left(t+\frac{x}{c}\right) + y_r(t-\frac{x}{c}) \tag{6.9}$$

We have an equation that shows that any vibration of a string can be expressed as a combination of two travelling waves; one travelling in the left direction (y_l), and one travelling in the right direction (y_r). The equation also states that the rate of the propagation of this wave is at the velocity c along the string.

We can implement these two travelling waves as two delay lines, one representing the propagation of the wave in the left direction and one representing the propagation of the wave in the right direction.

7. Implemented Physical Models

The synthesiser currently supports the following physical models:

- Commuted 1D Waveguide (Strings and tubes)
- Flute (variation of the 1-D waveguide model)
- Banded Waveguide (Bars – xylophone and marimbas)
- Karplus-Strong (Plucked string)

The synthesiser can also support a 2D waveguide mesh model, however is not able to calculate large enough meshes to be considered tonal. It is locked to a size of 8x8 and larger sizes are incapable of being processed in real time.

7.1 Struck String

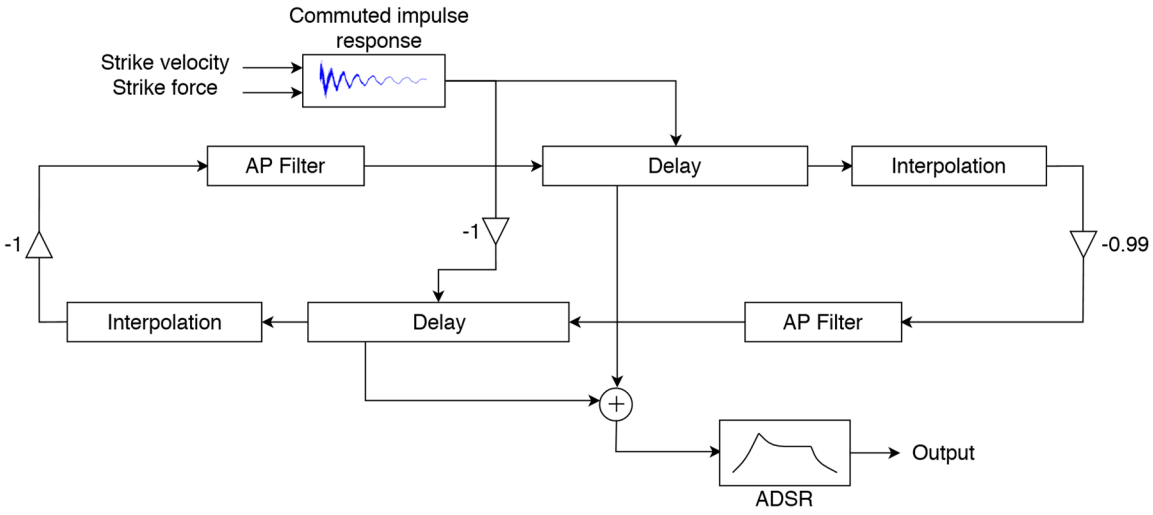


Figure 7.1 Block diagram of the string model

A struck string can be modelled using two 1-D waveguides to simulate the propagation of a wave on a string. To pluck the string, we load an initial shape into the upper and lower delay lines. This effect is achieved by inputting the samples from the output of a wavetable synthesizer loaded with a commuted impulse response.

On either end of the string, which could be referred to as the nut or bridge, the wave is reflected, and its polarity is inverted. A lossy termination of the string would result in a reflection coefficient of less than 1.

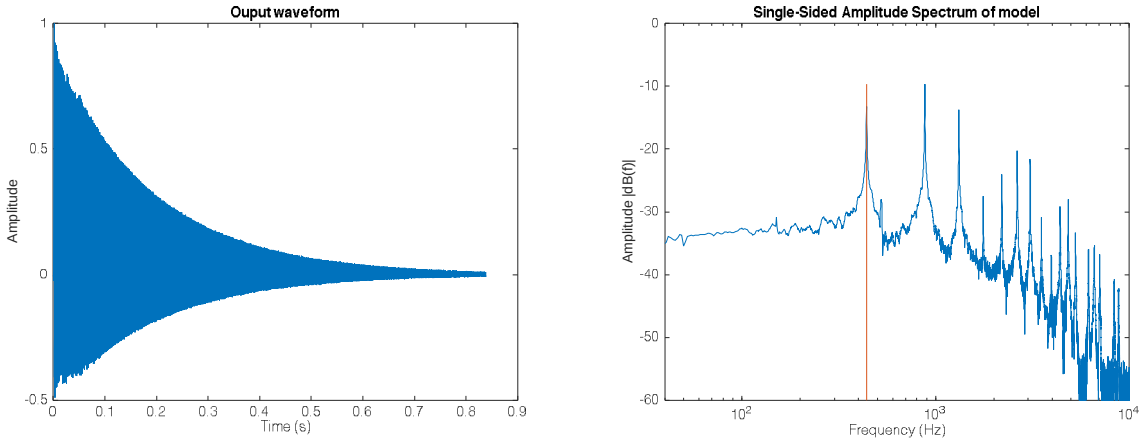


Figure 7.2 Waveform and frequency plot of modelled struck string at 440Hz

These two delay lines together are considered a “waveguide filter”, and as per our solution to the wave equation (7.9), the output waveform of the model can be considered to be a linear combination of the left-travelling delay line and the right-travelling delay line.

The sum of the contents of the two delay lines is the displacement of the string, and the difference between the two delay lines is the string velocity.

As the system is approximately linear and time invariant, we can move the ADSR stage to the end of the model, which allows us to make the reflections completely lossless and observe the effects of the harmonics involved when adding extra impulse responses (or plucks) while the string is already vibrating.

There were numerous additions made to the original model to make it more robust and expressive. These are detailed in Section 8.

7.1.1 Tuning the string

In the digital domain, the delay line representing the string must have an integer length due to arrays only being able to be discrete sizes. An integer delay length is not ideal for a musical instrument, as delay length is directly related to the pitch of the note played. The relationship is given by:

```
1   osc->delay_len_total = (AUDIO_FS/freq/2.0f/osc-
    >downsample_amt)+1;
```

Figure 7.3 String tuning C code for the struck string model

Where *AUDIO_FS* is the sample rate (44.1kHz), and *freq* is the frequency of the required note.

By using an integer delay length, we are effectively trimming off the decimal places after this calculation, leading to a delay line that is shorter than is required for a specified pitch.

This problem becomes more pronounced as the pitch of the note is increased, and the delay lengths get smaller, as a small change in delay length could correspond to a large change in pitch.

There are a few solutions to this problem that have been proposed:

7.1.1.1 Linear interpolation

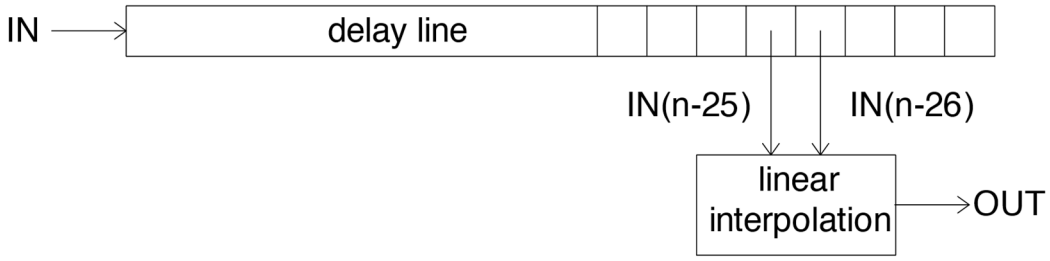


Figure 7.4 Linear interpolation on a delay line

Linear interpolation makes the delay line a fraction longer by taking a weighted average of the two closest delay lengths. For the figure above (7.4), if we wanted a delay length of 25.3 samples we can do:

$$OUT(n) = 0.7 \times IN(n - 25) + 0.3 \times IN(n - 26)$$

The disadvantage of linear interpolation is that it acts as a form of averaging filter, and attenuates the higher frequencies. This leads to some notes sounding *louder* than other notes, depending on the value of the coefficient.

My solution to this problem is to apply make-up gain to each output sample depending on the intensity of linear interpolation that was being applied – which is directly proportional to the excess fraction of the delay line.

I found this value by running the output of the synthesizer through a signal analyser and comparing the amplitudes of each of the notes as I changed the value of the make-up gain. I did this for the string model at no stiffness.

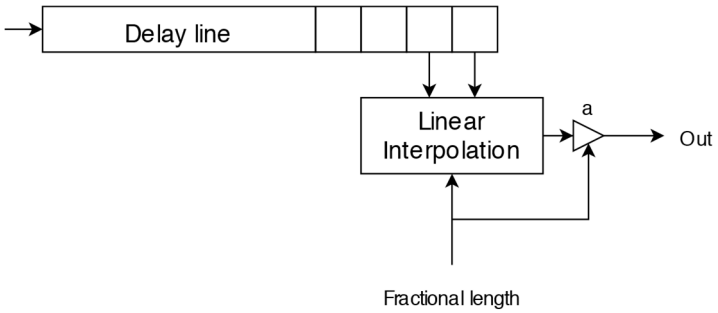


Figure 7.5 Modified linear interpolation with make-up gain

This approach works well most of the time and is a CPU efficient way of solving linear interpolation's biggest drawbacks (only requiring one multiplication). There are caveats however –

the frequency response of the note varies based on the fractional delay line length, even though the amplitude is the same. Furthermore, I didn't perform this testing at different levels of stiffness, so when the all pass filter value is increased, sometimes the amplitude differences between notes become more noticeable.

```
1  out[i] = ((osc->velocity) / 0.8f + 0.2f) * 0.75f * (frac) *
   (osc->delay_l[osc->x_pos_l] +
2      osc->delay_r[osc->x_pos_r]);
```

Figure 7.6 Output code for synth - notice that the make-up gain is set to 0.75

7.1.1.2 All-pass filter

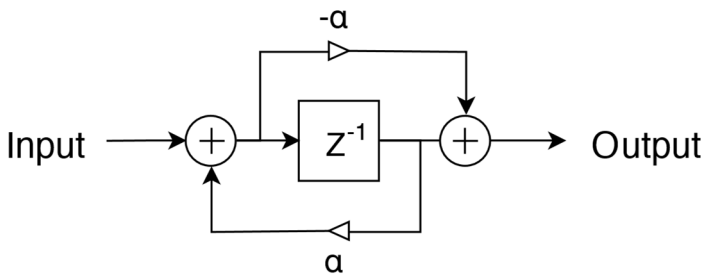


Figure 7.7 Block diagram of an all-pass filter

Using an all-pass filter solves the issue of the high frequency energy loss prevalent with linear interpolation. It instead introduces errors in the phase response, which causes incidental de-tuning of the highest partials.

Jaffe and Smith [12] proposed that the all-pass coefficient can be estimated as:

$$\alpha \approx \frac{1-d}{1+d}$$

The disadvantage to this method is that the phase response of the signal is altered. This is fine for models that don't have noticeable transients such as flutes, however as the majority of our models are percussive, this model causes a variation in the perceived stiffness of each note. This is due to stiffness also being implemented via an all-pass filter.

7.1.1.3 Stub

This method, proposed by Lowery [37], affects the tuning by placing a waveguide stub on one of the ends of the delay line, affecting the level of reflection.

This disadvantage to this method is that it affects the level of reflection on one of the ends of the delay line, also varying the volume of the reflection of different notes. The benefit of this method over linear interpolation is that for models that have a constant input (such as a wood-wind model), the effect is less noticeable, as the magnitude of the input signal (in this case white noise simulating breath) remains unchanged.

7.1.1.4 Conclusion of tuning methods

This model uses the modified linear interpolation method of tuning the waveguide, as the all-pass method affects the transients of the model. The stub method was not attempted on the waveguide string, however could be the focus of further work.

7.2 Tube Model

If you look at Equation 6.9 (the D'Alembert solution to the ideal string), the solution to the ideal tube is the same, except that string displacement y is replaced with air pressure P [3].

The physical equation for air pressure in an ideal cylindrical tube is given by:

$$\frac{\partial^2 P}{\partial t^2} = c^2 \frac{\partial^2 P}{\partial x^2} \tag{7.1}$$

And the solution follows:

$$P_{D'AI} = P(x,t) = P_l\left(t + \frac{x}{c}\right) + P_r(t - \frac{x}{c}) \tag{7.2}$$

If you find a tube and hit the end of it, you end up with a sound that sounds like a simple damped plucked string. This has been achieved with the waveguide model by swapping the reflection of one of the ends to no longer invert, as this effectively makes it a long continuous delay line.

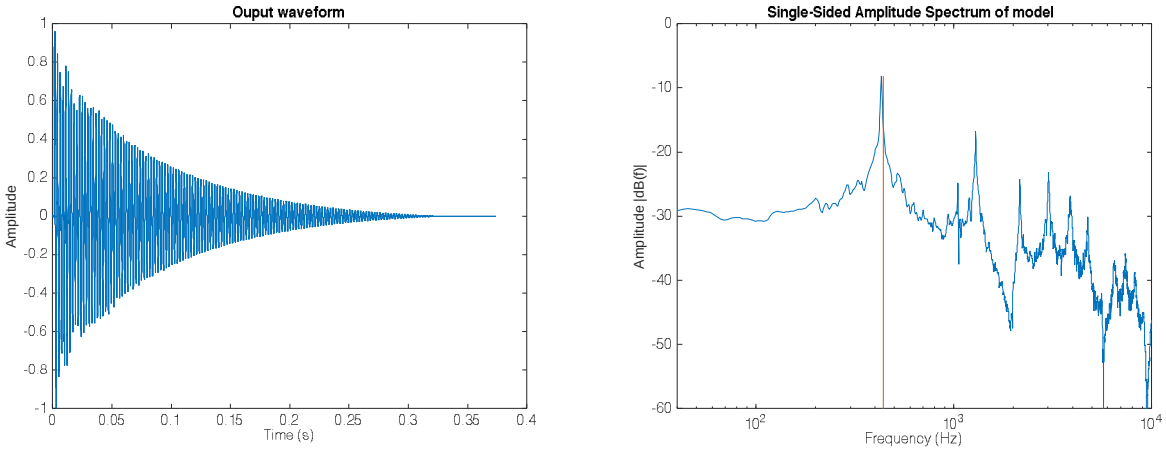


Figure 7.8 Waveform and frequency plot of modelled tube at 440Hz

7.3 Flute Model

This model aims to simulate a woodwind instrument based loosely on the physics of jet-reed based instruments. The model presented is based off Perry R. Cook's slide flute model.

The non-linearity of the jet of air travelling through the flute is modelled by the polynomial $x^3 - x$, while the half length delay line models the propagation time of the jet reed [15], which has been implemented by Karjalainen [38].

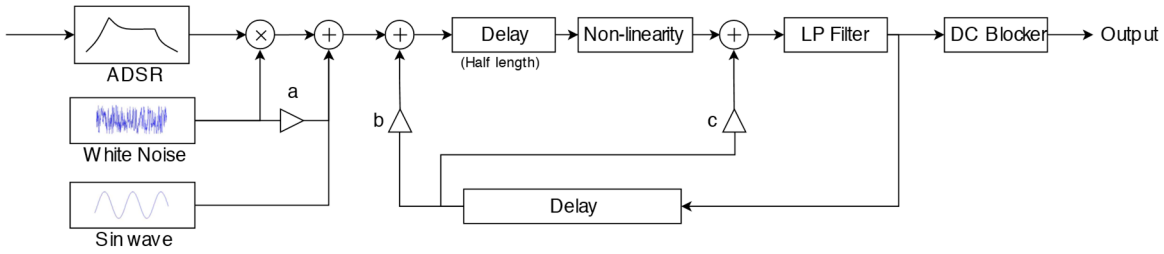


Figure 7.9 Block diagram of the flute model

The low-pass filter models the refection at the end of the instrument, while the variable a models how much white noise is mixed into the breath excitation function, which is modelled by an ADSR envelope.

The variable b represents the level of reflection of the output of the flute back to the mouth-piece, while the variable c represents the level of reflection from the output of the flute back up the tube, not including the mouthpiece.

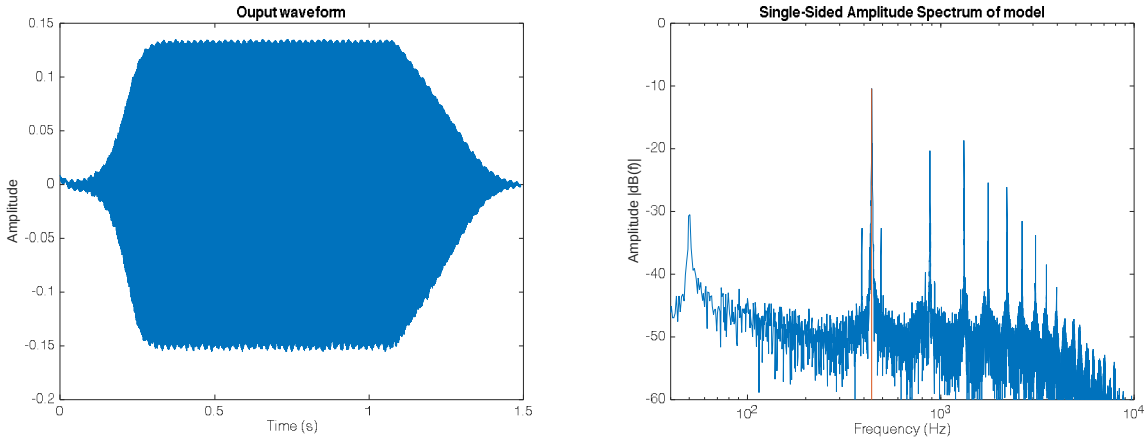


Figure 7.10 Waveform and frequency plot of modelled flute at 440Hz

7.3.1 DC Blocking

The model has a few problems, which involve a significant DC offset that is fixed by a DC blocker on the output. This is essentially a high pass filter:

```
1  // dc block
2  #define DC_FILT_GAIN 0.99f
3  osc->dc_filt_out = flute_out - osc->dc_filt_in + (DC_FILT_GAIN *
   osc->dc_filt_out);
4  osc->dc_filt_in = flute_out;
5  out[i] = osc->dc_filt_out;
```

Figure 7.11 C Code for output DC blocker

Furthermore, the model is difficult to tune, as both delay lines are of different lengths. Adding linear interpolation to both delay lines would be a possible solution, however the stub matching added to the output of the lowpass filter also achieves this, with less noticeable attenuation. This was implemented in MATLAB; however, I ran out of time in adding this to the device. As a result, the model gets more out of tune as the pitch is increased, and the delay lines get smaller.

7.4 Banded Waveguide Model

The banded waveguide model is a method of simulating rigid instruments such as bars. This can be considered a hybrid form of modal and waveguide synthesis as it utilises the frequency response of the instrument to tune the system. It involves several bandpass filters tuned to the frequencies of the significant modes of the system to provide an accurate representation.

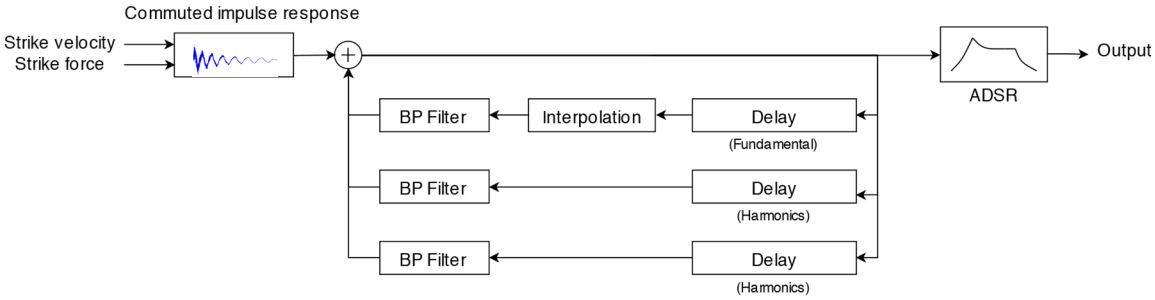


Figure 7.12 Block diagram of the implemented banded waveguide model

To tune the bandpass filters and delay lines, a frequency plot is taken of a marimba recording, and bandpass filters are placed on the peaks. A comparison of a marimba recording and the model simulating the marimba is detailed below.

In this case, only three delay lines are required. One for the fundamental and two for the dominant harmonics. It is worth noting that the length of the delay lengths does not contribute to the tuning of this system as the bandpass filter forces each delay line to conform to a specific frequency.

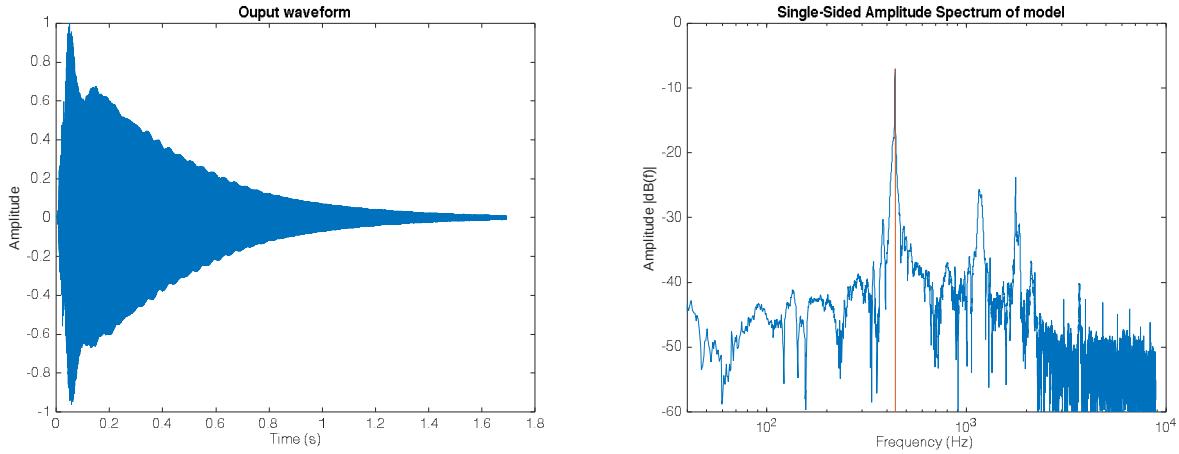


Figure 7.13 Waveform and frequency plot of recorded marimba at 440Hz

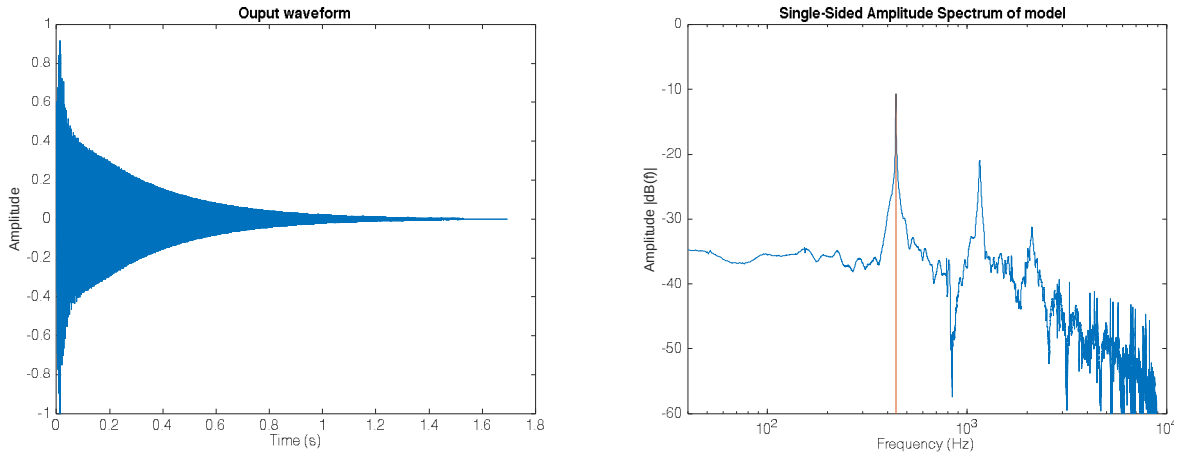


Figure 7.14 Waveform and frequency plot of modelled marimba at 440Hz

The same approach can be used for other bars such as xylophones. By using a similar method with the frequency plot of the square waveguide mesh, an approximation of it can be achieved also.

7.4.1 State Variable Filter Implementation

The band-pass filters were implemented with a state-variable filter. I am using the same filter as the output low-pass filter and has a resonance of 0.4999.

```
1  void svf2_gen(struct svf2 *f, float *out, const float *in,
   size_t n, uint16_t filter_type) {
2      float ic1eq = f->ic1eq;
3      float ic2eq = f->ic2eq;
4      float a1 = 1.f / (1.f + (f->g * (f->g + f->k)));
5      float a2 = f->g * a1;
6      float a3 = f->g * a2;
7
8      for (size_t i = 0; i < n; i++) {
9          float v0, v1, v2, v3;
10         v0 = in[i];
11         v3 = v0 - ic2eq;
12         v1 = (a1 * ic1eq) + (a2 * v3);
13         v2 = ic2eq + (a2 * ic1eq) + (a3 * v3);
14         ic1eq = (2.f * v1) - ic1eq;
15         ic2eq = (2.f * v2) - ic2eq;
16         out[i] = v1; // band-pass output
17     }
18
19     // update the state variables
20     f->ic1eq = ic1eq;
21     f->ic2eq = ic2eq;
22 }
```

Figure 7.15 State variable filter code (tapping band-pass output)

This filter was taken from Perth sound software company Cytomic, with the variables in the code above corresponding a circuit equivalent version of the filter.

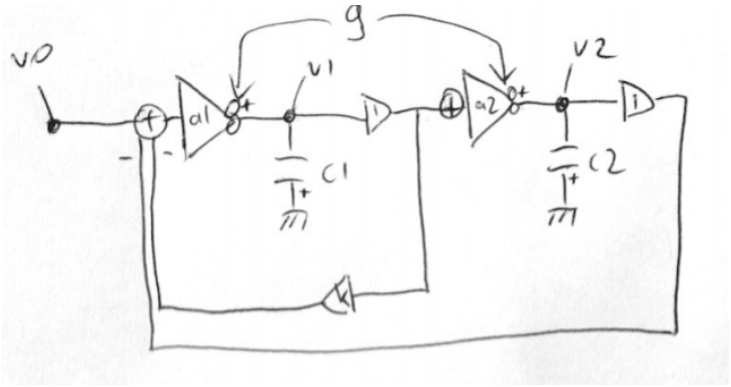


Figure 7.16 Circuit state variable filter (From Cytomic document) [39]

The resonance of 0.4999 means that the gain of the system is below 1, while being as close to 1 that the system doesn't stop resonating too quickly.

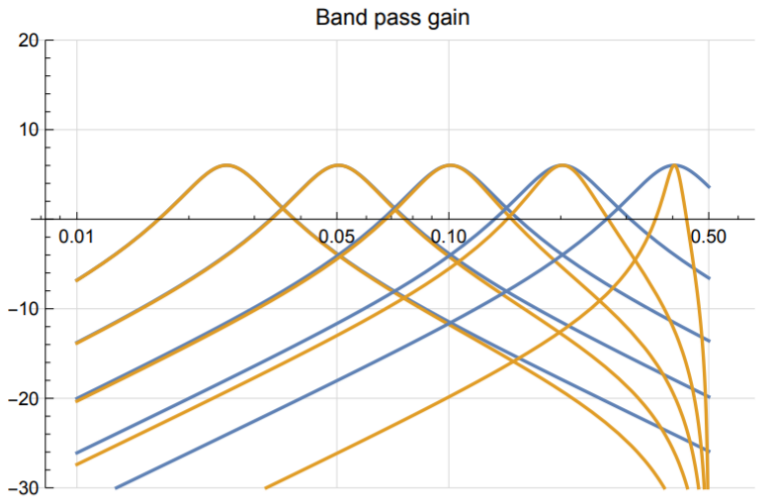


Figure 7.17 Transfer function for band pass filter (From Cytomic document) [39]

One of my biggest issues with this model was obtaining the correct frequency to feed into the bandpass filters. As all the calculations for the cut-off were done via a sinusoidal lookup table, the frequency is sometimes off, by up to 26 cents. Unfortunately, I ran out of time to fix this, and the regular method of linear interpolation like with the string model does not work as the bandpass filter is what is controlling the tuning.

A solution would be to create another lookup table of sorts to manually tune the frequency of each note, however I have yet to do this. I would estimate adding this would only take a few hours.

7.5 Karplus-Strong Plucked String

Karplus-Strong is a well-known method of physical modelling synthesis that loops a short noise burst through a filtered delay line to simulate the sound of a plucked string. Developed in 1983 by Kevin Karplus and Alex Strong, the method is well researched, and will not be covered in detail.

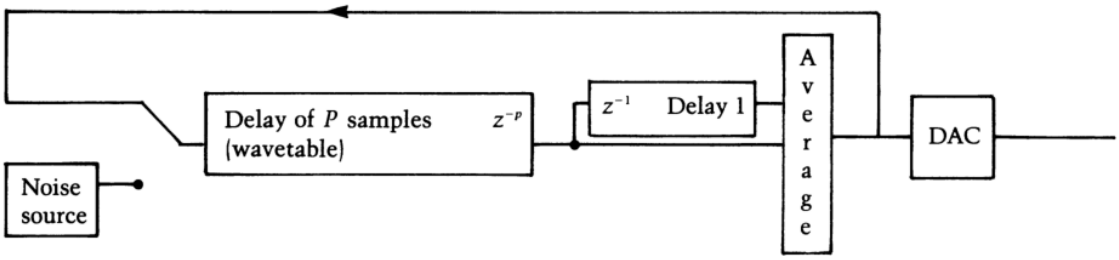


Figure 7.18 Block diagram of Karplus-Strong Plucked String [11]

This model was used to test the device initially, and for personal learning. The model is currently implemented on the synth, however does not have any additional modulation options.

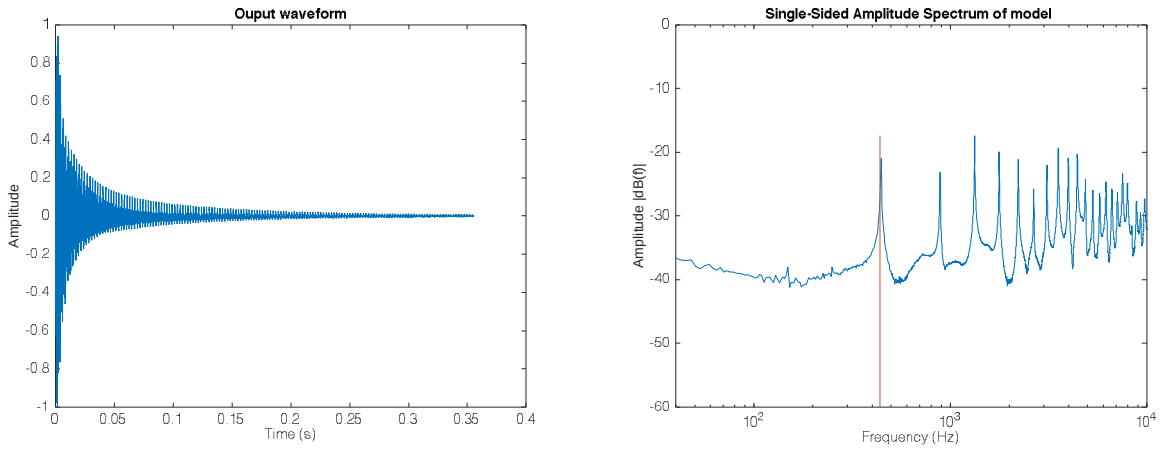


Figure 7.19 Waveform and frequency plot of Karplus-Strong Plucked String at 440Hz

7.6 2D Transmission Line Mesh Model

This model did not reach the state of being presentable on the synthesizer, due to it requiring more computing power than what was able to be provided by the STM32F4-DISC1 board.

2D and 3D waveguide meshes can be used to model wave propagation on membranes, plates and cavities [22], however time-domain implementations (such as my implementation) appear to be computationally prohibitive for real time performance [33].

Coming to this conclusion was almost to be expected, as it has been reported in the past that processors had yet to become fast enough to calculate large mesh sizes in real time. A recent paper in 2017 [33] explained that a waveguide mesh was still not viable for real time operation, and were aiming to find a frequency-domain representation instead.

Earlier experiments with a real time mesh in 2001 found that only meshes of 6x6 were achieved on a Pentium III 500MHz processor and a 10x10 mesh on an Athlon 1.3GHz system [24].

Considering that I had a 168MHz microcontroller and managed to achieve an 8x8 mesh shows that there is still room to optimise the waveguide mesh further.

On the device, I attempted a metal plate model and a membrane model based off prior work by [25].

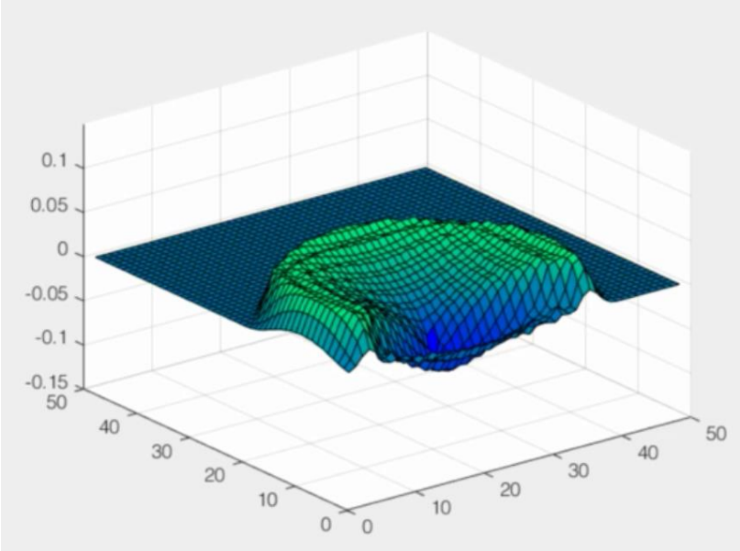


Figure 7.20 50x50 sized waveguide mesh after excitement

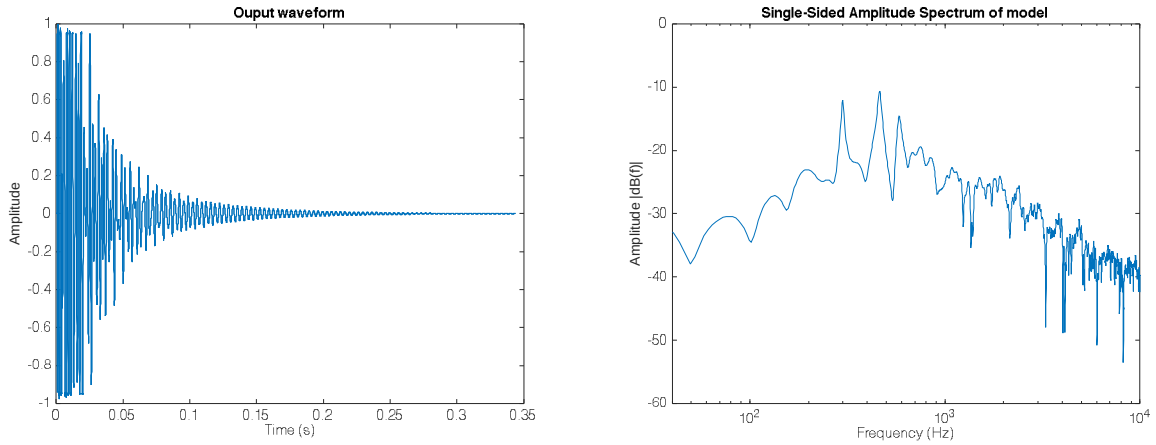


Figure 7.21 Waveform and frequency plot of 2D Waveguide mesh (membrane)

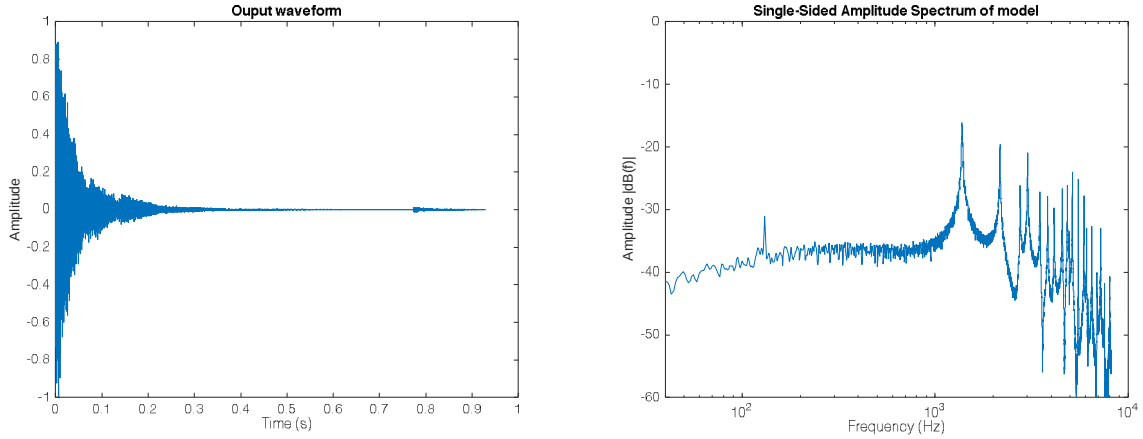


Figure 7.22 Waveform and frequency plot of 2D Waveguide mesh (metal plate)

8. Improvements made to the Waveguide Models

8.1 Active Downsampling

One of the major problems with delay line based models is that the delay lines get larger as the note gets lower. Eventually they reach a length that is too long to fit into memory. One solution to this problem is to use a delay line of a fixed length, and to simply slow down the speed of writing and reading from the delay line. With a lot of filters, reflections and modulation options, this approach becomes slow due to having to perform linear interpolation many times at different points along the delay line to extract the samples or insert them back into the delay line. This also leads to an additive low-pass effect.

The other solution is to downsample the voice whenever the delay length for that note gets too long. When the delay length is found to be longer than what can fit into memory, the sample rate is halved, and the delay length is also halved. In practice, this means that in the main processing loop, this delay line is then only processed every 2 *ticks* instead of every *tick*, halving the pitch (as halving the decay length doubles the pitch).

```c
// Code to determine downsampling amount (from waveguidebanded.c)
uint32_t delay_len = AUDIO_FS/(osc->mode[i].freq_coef * freq);
if (delay_len > WGB_DELAY_SIZE) {
    uint32_t rough_ds = (delay_len / WGB_DELAY_SIZE) + 1;
    osc->mode[i].downsample_amt = rough_ds + rough_ds % 2;
    ...
}
```

Figure 8.1 C Code determining downsampling amount

Linear interpolation is then performed between samples to filter out some of the aliasing.

This method is close to unnoticeable when downsampling by a factor of 2 (as this is still at the respectable sampling frequency of 22500 Hz), however once a factor of 4 is required, noticeable aliasing appears in the higher frequencies.

It is worth noting that when using filters, the downsampling amount has to be taken into consideration. As it is generating half the number of samples, the filter cutoff needs to be halved also.

The same applies to wavetable synths, in that the playback of the wavetable needs to be doubled.

Further improvements could be made to adjust the output lowpass filter as the level of downsampling increases too filter out some of the aliasing.

8.2 Commuted Synthesis

In a stringed instrument, for example a guitar, the string is attached to a *body* which radiates the sound to the outside air. In doing so, it imposes its own frequency response on the sound radiated. This is illustrated in Figure 8.2.

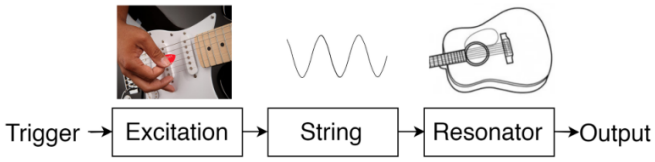


Figure 8.2 Block diagram of a typical stringed musical instrument

As the string and body are approximately linear and time-invariant, we can commute the string and the resonator.

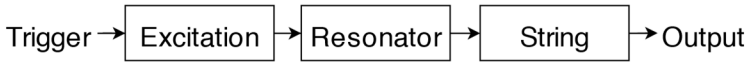


Figure 8.3 Equivalent diagram with commuted resonator

We can then combine the excitation and resonator into one block.

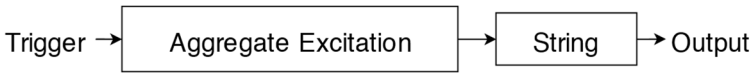


Figure 8.4 Equivalent diagram, with convolved excitation with resonator impulse response

In the case of a single impulse to pluck or strike a string, the aggregate excitation is simply the impulse response of the resonator [17] which can be extracted from a real instrument using a recording. This could be obtained by recording the output of a guitar with a microphone while striking the bridge with a force hammer. It is ideal to record as close to the instrument as possible to avoid also capturing the room response, and to keep the resonator response as short as possible [17].

8.3 Weak Stiffness

The ideal string equation derived in section 7 assumed that all the force on the string attempting to return it to its rest position was caused by the tension component. However, this would only be the case for string with next to no stiffness, such as cloth. A stiff string attempts to return to straightness even when there is no tension on the string.

To include stiffness in our model, a wave propagation speed that is dependent on frequency is required. We can adjust Equation (6.9) in section 6 to:

$$y(x,t) = y_l\left(t + \frac{x}{c(f)}\right) + y_r\left(t - \frac{x}{c(f)}\right) \tag{8.1}$$

Where f is frequency of the partials of the string. Implementing this to our delay line function isn't easy.

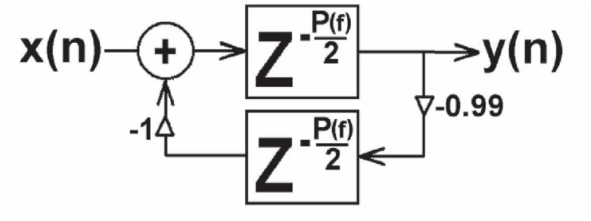


Figure 8.5 Filter view of waveguide string with stiffness [3]

One possible way to do this would be to have multiple delay lines for the different partials, which will be explored late with banded waveguide synthesis. We can predict the frequencies of the partials with an equation proposed by Morse (1948) [13]:

$$f_n = nf_0(1 + Bn^2)^{\frac{1}{2}} \tag{8.2}$$

Here, f_n is the frequency of the partial, f_0 is the fundamental frequency, and B is a number slightly greater than 0. [3] has predicted that B can typically be near 0.00001 for guitar strings or 0.004 for piano strings.

So how do we implement the $P(f)$ function? We could replace all the delay line units with an all-pass filter. An all pass filter allows us to implement a frequency dependent phase delay, while exhibiting a gain of 1 for all frequencies.

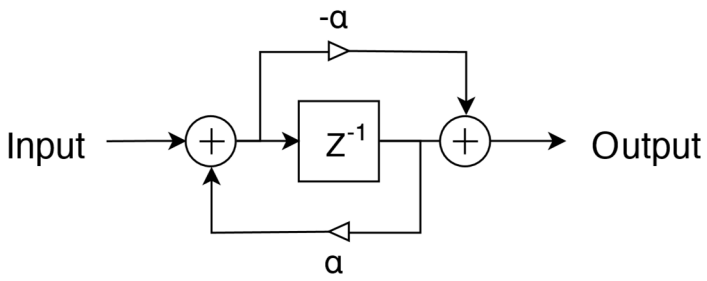


Figure 8.6 An all-pass filter

The variable α in the all-pass filter can have any value between -1 and 1. When $\alpha = 0$, the filter acts as a usual delay unit. When $\alpha > 0.0$, the filter exhibits delays longer than one sample, increasingly long for lower frequencies, and for $\alpha < 0.0$, the filter exhibits delays shorter than one sample, decreasingly so for high frequencies [40].

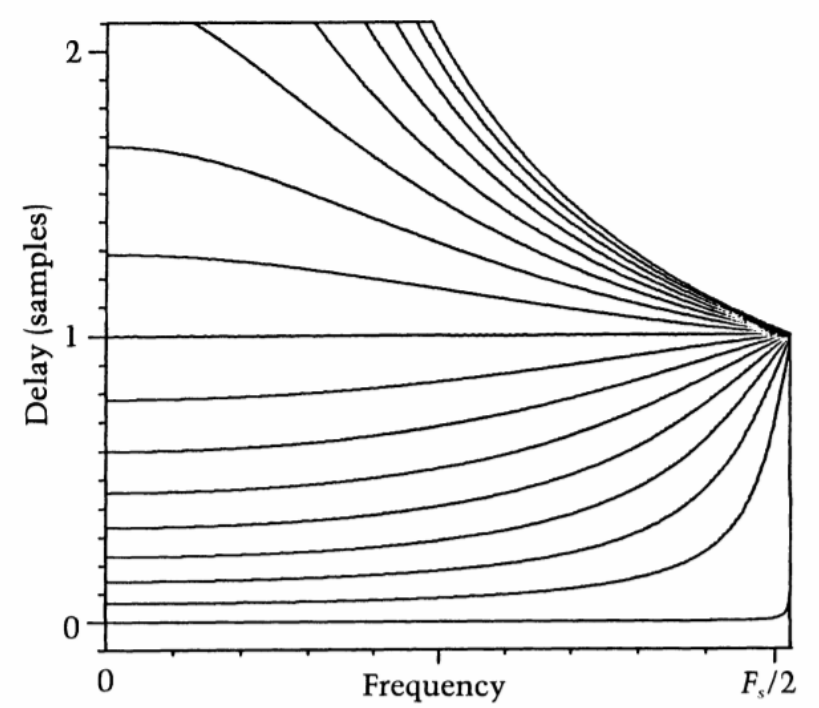


Figure 8.7 Sample delay of a first order all-pass filter at different frequencies, for varying α [12]

Implementing a chain of all-pass filters is more computationally intensive than a basic delay line, so for weak stiffness we can use only a few all-pass filters. In the string model there are only 2 used.

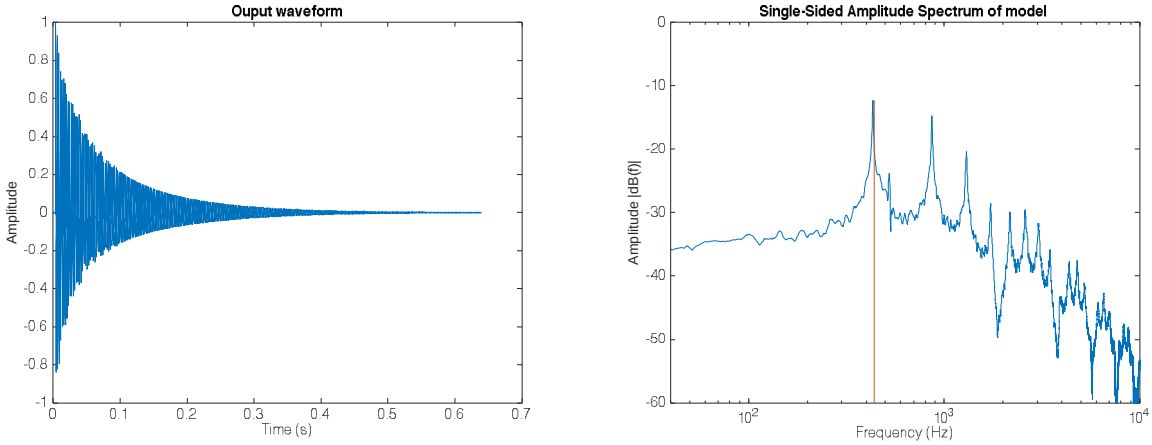


Figure 8.8 Waveform and frequency plot of modelled stiff struck string at 440Hz

8.4 Adjusting the pluck position

Adjusting the location of excitement along the string allows for different harmonics (otherwise referred to as modes) to be excited. These modes relate to the natural frequencies of the string. If we were, for example to excite the string a quarter of the way along it, we would strongly excite the second mode, but not the forth mode.

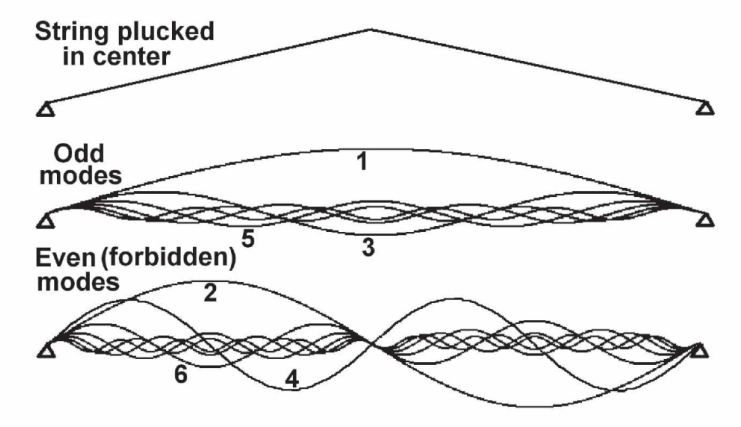


Figure 8.9 Top: Plucked string. Centre: Sinusoidal modes of vibration of a centre-plucked string. Bottom: Even modes, that would not be excited by the centre plucked excitation [3]

If you were to rest your finger on the centre of the string gently, and simultaneously pluck the string while removing your finger, you would excite some of the even modes of the string, but none of the odd modes. This is commonly referred to as playing *harmonics* by guitarists.

8.5 Adding modulation options

Physical modelling models vary from traditional synthesis methods (FM or subtractive) as they do not inherently provide the same level of modulation options in comparison.

In building this synthesizer, I intended to develop it to the stage that it was usable as an instrument capable of being played in a live setting. To further extend the synthesizer's usability, I adapted the state variable filter used in the banded waveguide model to be used as a low-pass filter for the output of the synthesizer which is mapped to the modulation wheel for all the instruments.

8.5.1 ADSR

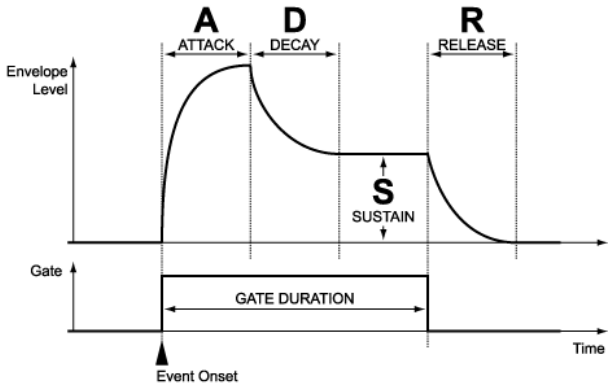


Figure 8.10 An example of an ADSR envelope as implemented

To provide the player with more knobs to tweak, I've added an ADSR (Attack, decay, sustain envelope to each voice that controls the amplitude of the note. A benefit of this is that it allows for the creation of interesting sounding pads by increasing the attack time and allows for very quick plucks and hits by turning down the release and decay.

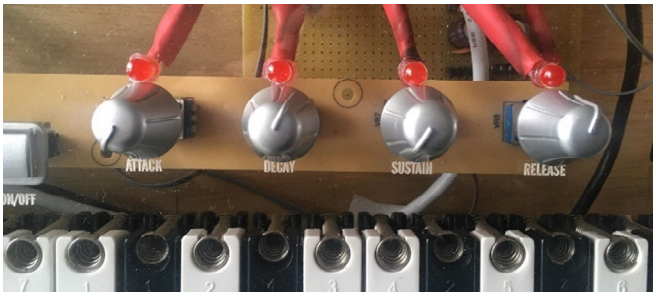


Figure 8.11 Implemented ADSR control knobs

The parameters for the ADSR are controllable by knobs on the synth frontplate. They can alter the attack, decay, and release time. The sustain knob determines sustain level.

I also experimented with linking the state variable filter cut-off frequency to the ADSR, however this was too computationally expensive and could only be sustained for monophonic operation. The included ADSR also has lights that indicate what state the ADSR is in.

8.5.2 Lowpass filter on output

Additionally, I placed a low pass filter on the output chain (Figure 8.12). The initial purpose of this was to allow for a filter envelope, however that proved too processor intensive. The main use for the filter now is to filter out some high harmonics that appear due to inaccuracies in the models (most noticeable on the banded waveguide model). Furthermore, some of the models had some harsh high frequency elements that required some filtering to become less tiresome to listen to (including Karplus-Strong, and the flute model).

8.5.3 Model control parameters

The left set of rotary encoders allows the player to change model parameters. These vary depending on the currently selected physical model.

Figure 8.12 Model control parameter knobs

The parameters listed are:

- Filter resonance
- Reflection amount
- Tone/stiffness
- Strike position

In all the models, filter resonance affects the resonance of the output filter. For the other parameters, they serve the following functions:

	String	Tube	Flute	Banded Waveguide	Karplus-Strong
Reflection amount	Level of reflection on the end of waveguide	Level of reflection on the end of waveguide	Adjusts internal reflection gain	Amplitude of partials	Alters attenuation of delay line
Tone/stiffness	All-pass filter coefficient (between 0 and 1)	All-pass filter coefficient (between 0 and 1)	Adjusts output lowpass filter coefficient	Level of mode mixing (makes metallic sound)	-
Strike position	Location of excitation along waveguide	Location of excitation along waveguide (however scope is limited)	Varies level of tremolo and white noise added to input	Adjusts frequencies of partials	-

8.5.4 Pitch Bend

A pitch bend wheel was also added. This alters the frequency that the delay line resonates at, and also the bandpass filter cutoff frequency for the banded waveguide model.

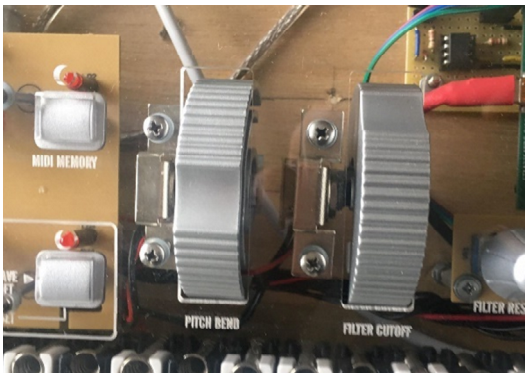


Figure 8.13 Pitch bend wheel (left) and Lowpass filter cutoff (right)

This pitch wheel introduces popping noises to some of the models. Due to how I implemented the string and tube models, if the pitch is changed, both the delay line for the left travelling wave and the right travelling wave are changed simultaneously. This leads to a “popping” sound due to delay line units being empty. For other models that don’t use linear interpolation, the pitch bend can cause popping as the delay line length snaps to another discrete length. The effect of this can be reduced by the lowpass filter.

8.5.5 Front panel buttons

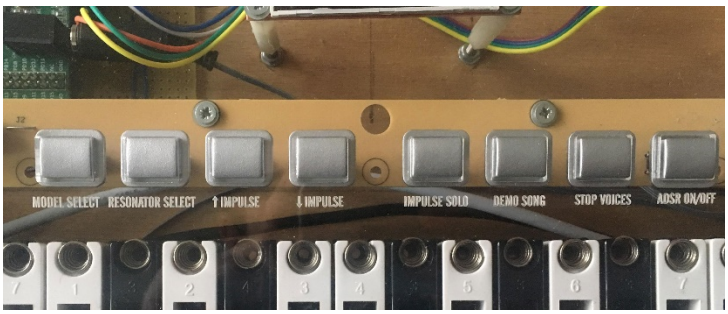


Figure 8.14 Front panel buttons

The buttons on the front of the synth allow the player to:

- Change between models
- Alter the resonator (string to tube, or for banded waveguide affect the frequency for the partials)
- Change between different instrument impulse responses to excite the models (currently 4 are included)
- Solo the impulse (to hear what the excitement sample sounds like)
- Demo song
- Stop voices (this is a way to stop extremely long sustained notes)
- ADSR on/off – disable or enable the ADSR envelope

9. Practical Considerations

9.1 Audio processing

Initially the physical model was developed for MiOS, based on FreeRTOS. MiOS provided an ideal framework for developing a synthesizer [41]. It included pre-written audio drivers that were easy to interface with, out of the box support for USB midi, and a schematic for a "MidiBOX Core" which would make adding external controls such as buttons and variable resistors an easy task.

Upon further research and experimentation, it appeared that MiOS was incapable of processing floats using the hardware FPU (Floating-Point Unit). The lack of being able to use the FPU would have led to a model that was more complex, due to having to convert everything into fixed point, and potentially more inaccurate. Delving into the MiOS code, the source of the lack of compatibility with FPU operations appeared to stem from the version of FreeRTOS in the repository not supporting the saving and restoring of vectored floating point registers during context switches [42]. After a lot of digging through the source code and contacting people on the forums, a fix was unable to be found.

Further testing showed that a floating-point model was faster than the fixed-point model. I attempted my synth code entirely in fixed point and found that after converting to floating point using the FPU, I was able to run 2 extra voices with my string model before any sort of buffer underruns.

9.2 Developing an understanding of digital audio

In developing a physical modelling synth utilizing waveguides, a general appreciation of digital audio is required. The book, well named "The Audio Programming Book" [34] was fundamental in developing an understanding of how digital audio worked, and the basis for FIFO (First In First Out) buffers worked, which were used extensively in the synth.

9.3 Connecting a keyboard

Finding a keyboard to attach synth involved searching for wholesalers that provide the keybeds for the larger synthesizer manufacturers. The industry standard for keybeds appear to be Fatar, an Italian company that sells the keybeds to many of the large keyboard manufacturers. I couldn't find any affordable places to purchase one of these keybeds so instead used an old MIDI keyboard I had lying around.

Figure 9.1 UMX49 Midi Keyboard

Behringer U-CONTROL UMX49 MIDI keyboard was deconstructed to be used as the keyboard input for the synthesizer. This was due to it being cheaper than buying a key bed, and

also due to the keys already being mapped to their appropriate midi output with its internal circuitry.

Using a MIDI keyboard saved the hassle of allocating each key on the keyboard to a defined note and having to calculate the velocity of the key. On most velocity sensitive keyboards, each key has two small buttons underneath it. Upon pressing a key, the two buttons are pressed in succession. This occurs faster when the key is pushed in a faster motion. Velocity is then calculated as a function of the time between the two button presses and translated to a value between 1 and 127 for use by the connected MIDI device.

I connected the keyboard to the STM32F4DISC-1 by connecting the MIDI output to a MIDI optocoupler circuit, which was then connected to one of the GPIO inputs of the STM32F4DISC-1.

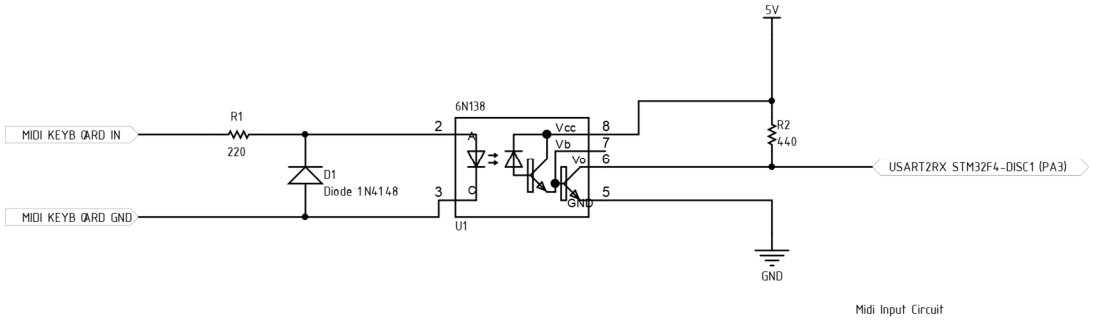


Figure 9.2 Midi optocoupler circuit

The MIDI output of the keyboard acts as a serial port running at a baud rate of 31250. MIDI uses current signalling to represent its serial communication so an optocoupler circuit is required by MIDI standard [43] to be used as the receiving circuit. This circuit then outputs a digital voltage output to be read by the microcontroller.

On the software side, a UART driver is required to interpret the incoming bytes, and to decode the MIDI messages a state machine is needed. The state machine is then used to call functions such as note on/off, pitch bend or control changes.

For the UART driver I opted to not use the manufacturer's standard libraries and instead opted for a third-party driver written by the github user *deadsy*. In my opinion this driver was much easier to use and better documented. I opted to use a lot of the drivers made by *deadsy* for the other components also.

9.4 Outputting audio

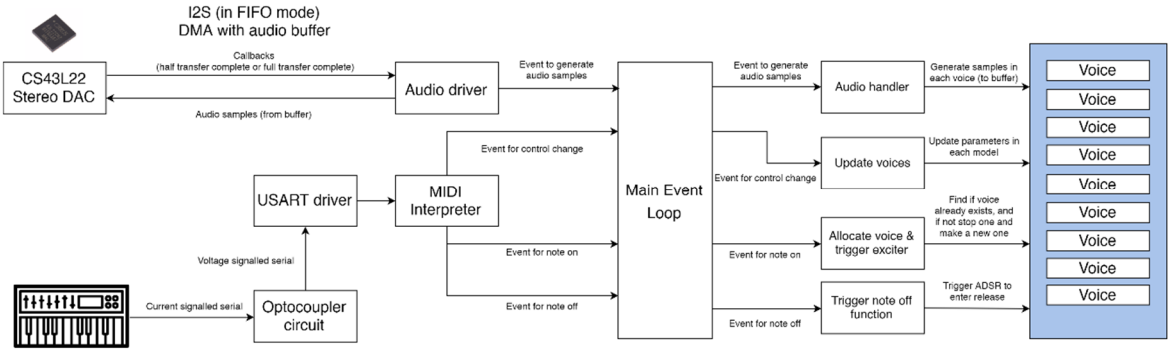


Figure 9.3 Overall block diagram of software

To output audio from the synthesiser, the device sends audio samples to a stereo digital to analog converter (DAC) which has an integrated headphone output.

This headphone output is then connected to the ¼ inch jack on the back of the synthesizer, which can be connected to a speaker or a set of headphones.

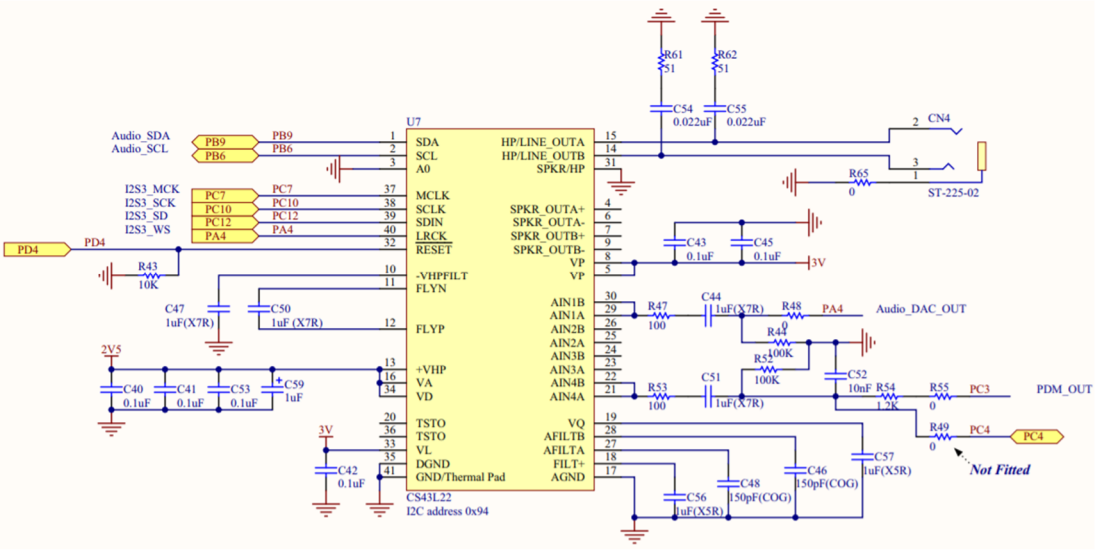


Figure 9.4 Schematic of the integrated DAC

Looking at the schematic to see how the integrated DAC (CS43L22) is connected to the rest of the board, it is evident that the Class-D amplifier outputs (SPKR_OUT) are not connected. This led to some confusion the initial time I tried to send audio out as I was attempting to play audio from these outputs instead of the headphone output.

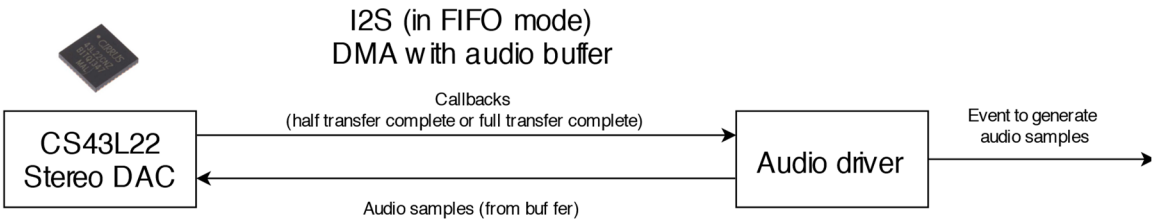


Figure 9.5 DAC to audio driver communication

On a software level, the microcontroller communicates with the Stereo DAC using I2S in FIFO mode. Here, direct memory access (DMA) is configured to constantly read data from the audio buffer in the device's memory.

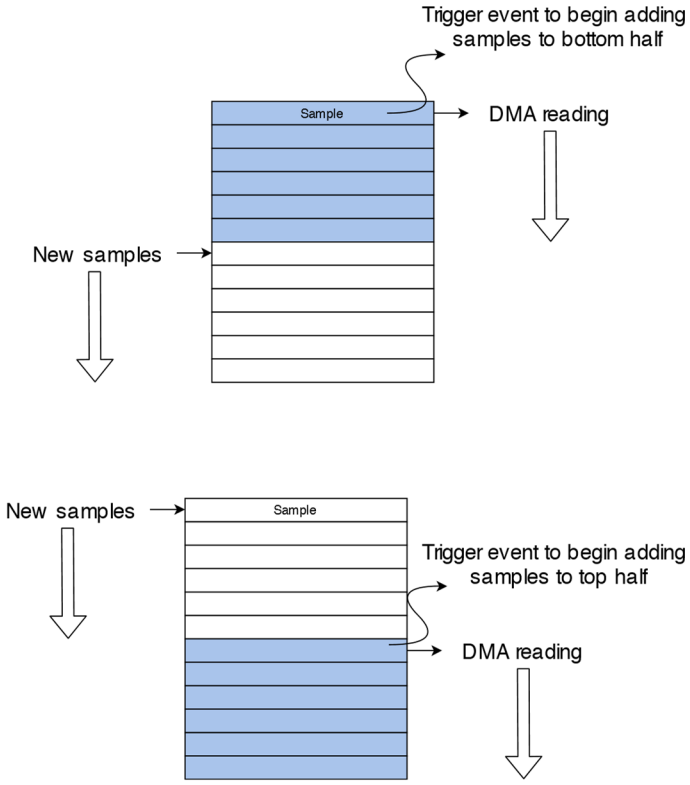


Figure 9.6 Writing to the audio buffer

The CS43L22 sends a callback to the audio driver when half of the buffer has transferred, telling the device to begin adding new audio samples to the top half of the audio buffer. When the full buffer has been sent, a callback is also sent, telling the device to begin adding audio samples to the bottom half of the audio buffer.

The device is constantly having to generate audio samples at a faster rate than they are being read by the DMA. The device constantly checks to ensure that when it finishes writing to a half of the buffer, that the DMA isn't also reading from that half of the buffer. If this occurs, it counts as a buffer underrun, and usually leads to glitches in the audio output. A lot of the work in developing the algorithms for this synth involved balancing functionality and modulation options with performance, to ensure that buffer underruns didn't occur.

9.5 Internal display

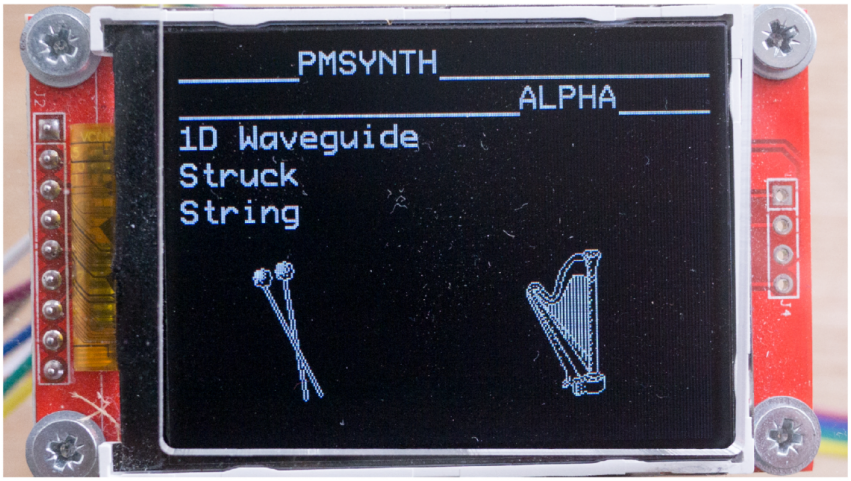


Figure 9.7 Screen and UI

The screen chosen for displaying the current patch is an ILI9341 2.2” LCD screen, connected to the STM32F4-DISC1 via SPI. In its current form, the screen displays the current patch, exciter and resonator model in use. It does not run via the event loop processor, which has yet to be fully implemented. As a result, if updates to the screen occur while an interrupt occurs, text on the screen can sometimes appear garbled.

The icons for the instruments were drawn by me in Photoshop and converted into mono-chrome byte arrays in Matlab with the script in the Appendices. It takes a monochrome .gif image and outputs a hexadecimal array to be added to the C code.

9.6 Supporting circuitry

The synthesizer has an on/off switch connected to the input of the power supply circuit. The circuit uses a 7805 linear regulator to provide 5V to the STM32F4 board and 4.3V to the MIDI keyboard. This is because I’m powering the keyboard via the battery input, which typically takes 3 AAA batteries.

The reason for adding the linear regulator is that I had trouble finding 5V wall plugs, and I would think that a musician would not go and check the voltage before plugging it into the synth. In the case of an input voltage that was too high, the linear regulator protects against overvoltage. Furthermore, a reverse voltage protection diode was also added in case the wall plug had an inverted polarity.

Four LEDs were also added to show the ADSR envelope state. The circuit for these is also in the appendices.

9.7 Polyphony

The software on the synth defines that each voice has a struct that describes it. Polyphony effectively states how many of these structs exist. Once a threshold of structs is reached, the processor cannot generate enough samples for each given voice and causes an audio buffer underrun. Likewise, each voice requires a certain amount of memory to store its settings, and

the contents of that voice's delay line. Therefore, there are two factors that determine maximum polyphony. In my case, the processor proved to be the bottleneck, limiting the polyphony to 12.

The current method of allocating voices on the device is "round robin". This is the easiest to implement as when a new voice is triggered, it will stop the first voice that was pressed. The downside to this method is that it neglects keys that are pressed down. Therefore, if the user is playing a chord and plays a lot of keys for a melody, while sustaining the chord, some of the lower notes of the chord will the stopped.

I fixed this in a late build of the code, however have chosen to leave it out for presenting, as it is still unstable (as there are multiple ways a voice can 'stop' – either note off, or voice snatching if the polyphony limit is reached).

9.8 Component breakdown

The following section describes the components and items that were used in construction of the synth:

9.8.1 Case

- Wood – plywood
- Wood – laminate
- Laser-cut 3mm Perspex
- Screws
- 3M bolts, nuts and washers
- Woodglue

9.8.2 Circuits

- 7805 linear regulator
- 4 x bypass capacitors
- 2 x reverse voltage protection diodes
- 1 x signalling diode
- 1 x 6N138 optocoupler
- 1 x 220 ohm resistor
- 1 x 440 ohm resistor
- 4 x 68 ohm resistor
- 8 x screwable terminal blocks
- Dupont connectors
- 4 x red LEDs
- 1 x STM32F4 discovery board
- 1 x 2.2 inch TFT screen

9.8.3 Other

- Power jack
- Power switch
- ¼ inch plug
- 2 x USB Type B connectors

- UMX49 Behringer Midi Controller
- 1 x MIDI cable
- 6V wall plug
- Multiple sheets of protoboard

9.9 Housing design

To design the housing for the synthesizer, I drafted up the design in AutoCAD. 4 different revisions were required before the final product. These were primarily due to me performing all the measuring with a ruler, and therefore various measurements were slightly inaccurate.

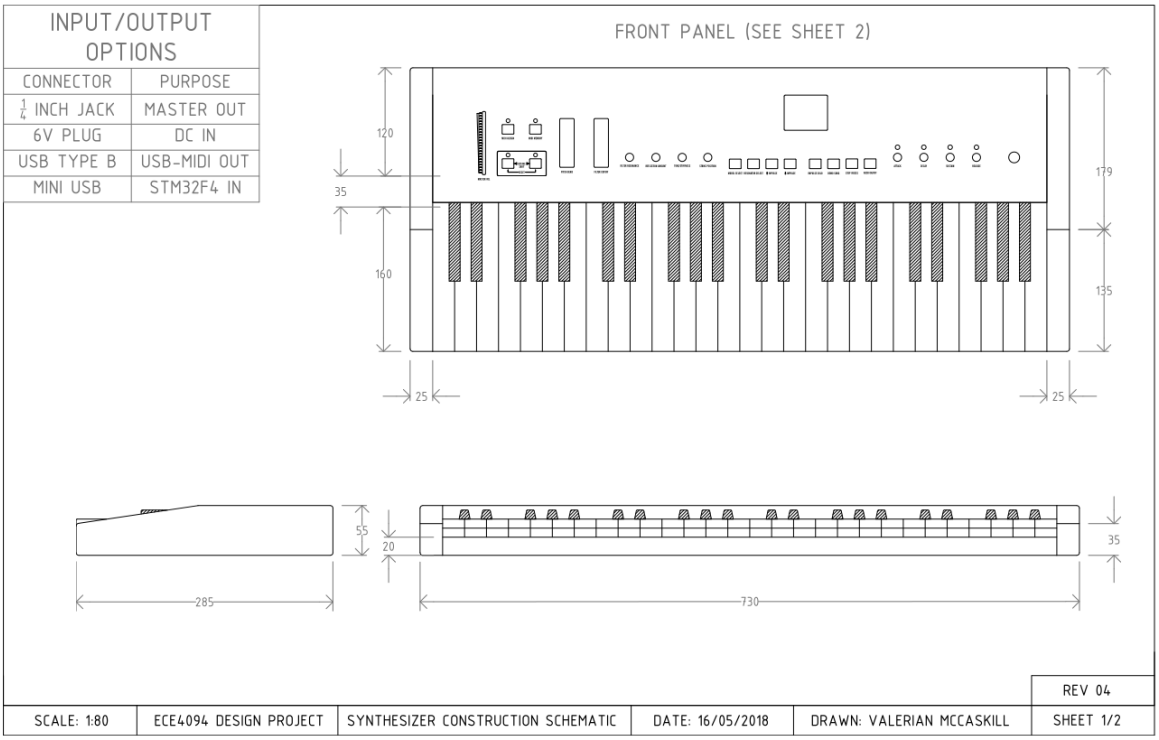


Figure 9.8 AutoCAD drawing of synth design

Initial prototypes were cut out of wood to test hole sizes and understand the size required to fit components.

Figure 9.9 Early prototype of front panel

After a layout was completed, the front and rear panel were laser cut using the Monash laser cutter. One revision had to be made to add labels to the front panel. Full sized schematics can be found in the appendices.

10. Further work

The models developed over the course of this project were primarily limited by the processing power of the hardware and were therefore not given as many modulation options as I would have liked. Benefits of a faster processor would involve higher levels of polyphony and more complicated models to be implemented.

I also ran out of time to properly 'finish' the models other than the struck string/tube model, and in their current state they still have minor tuning problems and less modulation options than I would have liked. With more time I would be able to get these models to a better state, however that would involve creating a large amount of lookup tables and optimization, both of which are quite time intensive.

Based on the Yamaha patents, there appear to be more efficient methods of implementing the string model that only involve one delay line. Due to me finding the patents quite late into the project, I didn't implement these.

Furthermore, the 2D mesh implementation was still not viable on the device in question. Using a more powerful device may lead to a tuneable 2D mesh for real-time use.

On the device construction side, work should be done to implement polyphony with a 'last note' priority, so that voice stealing doesn't occur as often. Furthermore, the screen driver should be rewritten as to not be affected by external interrupts. Adding it as an event for the event processor is advised.

11. Conclusion

Over the course of this year, I have developed a standalone, physical modelling synthesizer, capable of playing multiple models that simulate the properties of real-world instruments. The theory behind this synthesizer was based on work done by physicists in the 1700s, that was discretised into delay lines in the late 1900s. By creating a real-time implementation of these models, not only do I now have a working synthesizer, but I have shown that the barriers of entry into physical modelling have decreased over the years. More complex models can be developed based on prior work, and more interesting implementations and instruments can be developed on faster hardware. This is very exciting news, and I personally can not wait to see what amazing new instruments are invented in the future.

12. References

[1] N. Hind. (2005, 3/10). *Common Lisp Music (CLM) Tutorials*. Available: https://ccrma.stanford.edu/software/clm/compmus/clm-tutorials/toc.html

[2] J. Kojs, S. Serafin, and C. Chafe, "Cyberinstruments via Physical Modeling Synthesis: Compositional Applications," *Leonardo Music Journal,* vol. 17, pp. 61-66, 2007.

[3] P. R. Cook, *Real sound synthesis for interactive applications*. Natick, Mass.: A K Peters, 2002.

[4] S. Bilbao, *Numerical sound synthesis: finite difference schemes and simulation in musical acoustics*: John Wiley & Sons, 2009.

[5] M. Mersenne, *Harmonie universelle: contenant la théorie et la pratique de la musique (Paris, 1636)* vol. 2: Editions du centre national de la recherche scientifique, 1975.

[6] J. Sauveur, *Syste'me general des intervalles des sons, & son application à tous les systèmes & à tous les instruments de musique*: Chez J. Boudot, 1701.

[7] O. Darrigol, "The acoustic origins of harmonic analysis," *Archive for history of exact sciences,* vol. 61, pp. 343-424, 2007.

[8] J.-P. Rameau, *The complete theoretical writings of Jean-Philippe Rameau* vol. 6: American Institute of musicology, 1972.

[9] J. Bernoulli, "1742a. Propositiones variae mechanico–dynamicae," *Bernoulli,* vol. 4, pp. 253-386, 1742.

[10] J. Smith, "History of Virtual Musical Instruments and Effects

Based on Physical Modeling Principles," ed, 2017.

[11] K. Karplus and A. Strong, "Digital Synthesis of Plucked-String and Drum Timbres," *Computer Music Journal,* vol. 7, pp. 43-55, 1983.

[12] D. A. Jaffe and J. O. Smith, "Extensions of the Karplus-Strong plucked-string algorithm," *Computer Music Journal,* vol. 7, pp. 56-69, 1983.

[13] P. M. Morse, *Vibration and sound* vol. 3: McGraw-Hill New York, 1936.

[14] M. Karjalainen and U. K. Laine, "A model for real-time sound synthesis of guitar on a floating-point signal processor," in *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, 1991, pp. 3653-3656.

[15] P. R. Cook, "A meta-wind-instrument physical model, and a meta-controller for real-time performance control," 1992.

[16] M. Karjalainen, J. Backman, and J. Polkki, "Analysis, modeling, and real-time sound synthesis of the kantele, a traditional Finnish string instrument," in *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, 1993, pp. 229-232.

[17] J. O. Smith III, "Efficient synthesis of stringed musical instruments," 1993.

[18] V. Välimäki, J. Huopaniemi, M. Karjalainen, and Z. Jánosy, "Physical modeling of plucked string instruments with application to real-time sound synthesis," in *Audio Engineering Society Convention 98*, 1995.

[19] P. R. Cook, "Identification of control parameters in an articulatory vocal tract model, with applications to the synthesis of singing," 1991.

[20] D. A. Jaffe and J. O. Smith III, "Performance Expression in Commuted Waveguide Synthesis of Bowed Strings," in *ICMC*, 1995.

[21] J. O. Smith, "Physical modeling synthesis update," *Computer Music Journal,* vol. 20, pp. 44-56, 1996.

[22] S. Van Duyne and J. O. Smith, "The 2-D digital waveguide mesh," in *Applications of Signal Processing to Audio and Acoustics, 1993. Final Program and Paper Summaries., 1993 IEEE Workshop on*, 1993, pp. 177-180.

[23] J. Smith and D. Rocchesso, "Aspects of digital waveguide networks for acoustic modeling applications," *Web published at http://ccrma. stanford. edu/jos/wgj,* 1997.

[24] D. T. Murphy, C. J. Newton, and D. M. Howard, "Digital waveguide mesh modelling of room acoustics: Surround-sound, boundaries and plugin implementation," in *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFx)*, 2001.

[25] B. Eaton. (2000, 15/09). *Drum Project Overview*. Available: https://ccrma.stanford.edu/~be/drum/drum.htm

[26] J. A. Laird, "The physical modelling of drums using digital waveguides," University of Bristol, 2001.

[27] M.-L. Aird, "Musical instrument modelling using digital waveguides," Ph. D. thesis, University of Bath, 2002.

[28] S. A. Van Duyne, D. A. Jaffe, G. P. Scandalis, and T. Stilson, "A Lossless, Click-free, Pitchbend-able Delay Line Loop Interpolation Scheme," in *ICMC*, 1997.

[29] L. Manzara, "The tube resonance model speech synthesizer," 2009.

[30] R. V. Migneco, "Analysis and synthesis of expressive guitar performance," 2012.

[31] B. Bank, F. Avanzini, G. Borin, G. De Poli, F. Fontana, and D. Rocchesso, "Physically informed signal processing methods for piano sound synthesis: a research overview," *EURASIP Journal on Advances in Signal Processing,* vol. 2003, p. 464536, 2003.

[32] P.-A. Grumiaux, R. Michon, E. G. Arias, and P. Jouvelot, "Impulse-Response and CAD-Model-Based Physical Modeling in Faust," in *Proceedings of the Linux Audio Conference (LAC-17), Saint-Etienne, France*, 2017, pp. 18-21.

[33] T. Smyth, J. Hsu, and R. Done, "Toward a real-time waveguide mesh implementation," in *Proceedings of the International Symposium on Musical Acoustics. International Symposium on Musical Acoustics, forthcoming*, 2017.

[34] R. Boulanger and V. Lazzarini, *The audio programming book*: the MIT Press, 2010.

[35] C. Roads, *Musical signal processing*. Lisse Netherlands ; Exton, PA: Swets & Zeitlinger, 1997.

[36] ST. UM1472 User manual [Online]. Available: https://www.st.com/resource/en/user_manual/dm00039084.pdf

[37] A. Lowery, "Model for multimode picosecond dynamic laser chirp based on transmission line laser model," *IEE Proceedings J (Optoelectronics),* vol. 135, pp. 126-132, 1988.

[38] M. Karjalainen, U. K. Laine, T. I. Laakso, and V. Välimäki, "Transmission-line modeling and real-time synthesis of string and wind instruments," in *Proceedings of the International Computer Music Conference*, 1991, pp. 293-293.

[39] A. Simper, "Solving the continuous SVF equations using trapezoidal integration and equivalent circuits," 2013.

[40] J. O. Smith, "Physical audio signal processing," *Linear Predictive,* 2010.

[41] T. Klose. (2018). *The MIDIbox Operating System*. Available: http://www.ucapps.de/mios.html

[42] A. Limited. (2018). *Using floating point calculations in an Interrupt Service Routine*. Available: https://community.arm.com/processors/b/blog/posts/10-useful-tips-to-using-the-floating-point-unit-on-the-arm-cortex--m4-processor

[43] S. Birch. (2009). *The MIDI Physical Layer*. Available: http://www.personal.kent.edu/~sbirch/Music_Production/MP-II/MIDI/midi_physical_layer.htm

13. Table of Figures

14. Appendicies

All C source code and audio demos are available at: https://github.com/valmcc/pmsynth

14.1 MATLAB Code (Audio)

14.1.1 Metal plate (2D Mesh)

```
close all; clear all;

Fs = 44100;
freq = 440;

tmax = 1;

wavespeed = 1;

grid_size = round(Fs/freq/2);

d= 0.5;

% boundary conditions

refln_EW = -.99;
refln_NS = -.99;

% loading in excitation sample

filename = 'mallet_2.wav';
exciter = audioread(filename);
%exciter = rand(1,30000);

%%% Resonator %%%

initial_strike_position_x = 0.2;
initial_strike_position_y = 0.4;


% preallocate mesh

vJ = zeros(grid_size-1,grid_size-1); % junction "velocity"

vN = zeros(grid_size,grid_size);
vS = zeros(grid_size,grid_size);
vE = zeros(grid_size,grid_size);
vW = zeros(grid_size,grid_size);

vN1 = zeros(grid_size,grid_size);
vS1 = zeros(grid_size,grid_size);
vE1 = zeros(grid_size,grid_size);
vW1 = zeros(grid_size,grid_size);

% where is the strike position?

exciter_x = round((grid_size-1)*initial_strike_position_x);
exciter_y = round((grid_size-1)*initial_strike_position_y);

% initial

vJ = d * (vE1(1:grid_size-1,1:grid_size-1) + vW(1:grid_size-1,2:grid_size) + ...
            vN(1:grid_size-1,1:grid_size-1) + vS(2:grid_size,1:grid_size-1));
%surf(vJ);

N = tmax * Fs;

out = zeros(1,N);

oldfilt = vJ(grid_size-10,grid_size-10);

% audio generating loop
```

```
63  for i = 1:N
64
65
66    if ( mod(i,2) == 0 ) % tick
67
68      for x=1:grid_size-1
69          for y=1:grid_size-1
70              vJ(x,y) = d * (vE(x,y) + vW(x,y+1) + vN(x,y) + vS(x+1,y));
71          end
72      end
73          % adding the exciter sample
74      if (i<length(exciter))
75          vJ(exciter_x,exciter_y) = exciter(i) + vJ(exciter_x,exciter_y);
76      end
77
78
79      for x=2:grid_size
80          for y=2:grid_size
81
82              % junction wave calcs
83              vE1(x-1,y) = vJ(x-1,y-1) - vW(x-1,y);
84              vN1(x,y-1) = vJ(x-1,y-1) - vS(x,y-1);
85              vW1(x-1,y-1) = vJ(x-1,y-1) - vE(x-1,y-1);
86              vS1(x-1,y-1) = vJ(x-1,y-1) - vN(x-1,y-1);
87
88          end
89      end
90      % boundary calculations
91      for x=2:grid_size
92          vE1(x-1,1) = refln_EW*vW(x-1,1);
93          vW1(x-1,grid_size) = refln_EW*vE(x-1,grid_size);
94      end
95      for y=2:grid_size
96          vN1(1,y-1) = refln_NS*vS(1,y-1);
97          vS1(grid_size,y-1) = refln_NS*vN(grid_size,y-1);
98      end
99
100   else % tock
101
102     for x=1:grid_size-1
103         for y=1:grid_size-1
104             vJ(x,y) = d * (vE1(x,y) + vW1(x,y+1) + vN1(x,y) + vS1(x+1,y));
105         end
106     end
107
108     % adding the exciter sample
109     if (i<length(exciter))
110         vJ(exciter_x,exciter_y) = exciter(i) + vJ(exciter_x,exciter_y);
111     end
112
113
114     for x=2:grid_size
115         for y=2:grid_size
116             % junction wave calcs
117             vE(x-1,y) = vJ(x-1,y-1) - vW1(x-1,y);
118             vN(x,y-1) = vJ(x-1,y-1) - vS1(x,y-1);
119             vW(x-1,y-1) = vJ(x-1,y-1) - vE1(x-1,y-1);
120             vS(x-1,y-1) = vJ(x-1,y-1) - vN1(x-1,y-1);
121
122         end
123     end
124
125     % boundary calculations
126     for x=2:grid_size
127         vE(x-1,1) = refln_EW*vW1(x-1,1);
128         vW(x-1,grid_size) = refln_EW*vE1(x-1,grid_size);
129     end
130     for y=2:grid_size
131         vN(1,y-1) = refln_NS*vS1(1,y-1);
132         vS(grid_size,y-1) = refln_NS*vN1(grid_size,y-1);
133     end
134   end
```

```
135
136
137     %surf(vJ);
138     %colormap winter
139     %axis([0, grid_size, 0, grid_size, -0.15 0.15])
140     %pause(0.01);
141
142     % pickup location
143
144     out(i) = 0.5 * (vJ(grid_size-10,grid_size-10) + vJ(grid_size-9,grid_size-10) ...
145     + vJ(grid_size-10,grid_size-9) + vJ(grid_size-9,grid_size-9));
146       out2(i) = mean(mean(vJ));
147
148   end
149   %play audio
150   sound(0.99*out/max(abs(out)), Fs);
151   plot(out);
152
153   plot(linspace(0,N/Fs,length(out)),out)
154   title('Ouput waveform')
155   xlabel('Time (s)')
156   ylabel('Amplitude')
157   figure(2)
158
159   y = fft(out);
160   P2 = abs(y/N);
161   P1 = P2(1:N/2+1);
162   P1(2:end-1) = 2*P1(2:end-1);
163   P1 = 10*log10(P1);
164   f = Fs*(0:(N/2))/N;
165   plot(f(1:4400),P1(1:4400))
166   title('Single-Sided Amplitude Spectrum of model')
167   xlabel('f (Hz)')
168   ylabel('|dB(f)|')
169   hold on
170   % plot line of intended frequency
171   plot([freq freq],[min(P1) max(P1)])
172   hold off
```

14.1.2 Stiff String

```
1   clear; close all
2
3   Fs = 44100;
4   freq = 440; %freq of note
5
6   p = round(Fs/freq/2)+1; %delay line length
7
8   tmax = 3;
9
10  N = 50000; % length of sound
11  plot_output = 0;
12
13  % loading in excitation sample
14
15  filename = 'mallet_2.wav';
16  exciter = audioread(filename);
17  %exciter = rand(1,30000);
18
19
20
21  %%% Resonator %%%
22
23
24  initial_strike_position = 0.1; % between 0 and 1
25
26  is_p = round(initial_strike_position*p);
27
28  dl_1 = zeros(1,p); %initial contents of shift register
29  dl_2 = zeros(1,p);
30
```

```
31  out = zeros(1,N); %pre-allocating output matrix
32
33  dl_1_ptr_1 = is_p; %pointer to shift register (and initial strike)
34  dl_1_ptr_2 = dl_1_ptr_1+1; %pointer to shift register
35
36
37  dl_2_ptr_1 = is_p; %pointer to shift register (and initial strike)
38  dl_2_ptr_2 = dl_2_ptr_1+1; %pointer to shift register
39
40  dl_1_bridge_ptr = p;
41  dl_2_bridge_ptr = p;
42
43  dl_1_nut_ptr = 1;
44  dl_2_nut_ptr = 1;
45
46  %%% fn = n * f0 (1+B*n^2)^(1/2)
47  % modify p by inverse of (1+B*n^2)^(1/2)
48  % B = 0.004 - piano strings
49  % B = 0.00001 - guitar strings
50  % can be implemented with an allpass filter
51  %        ----------->*-a-----
52  %       |                  \/
53  % x(n) -->(+)-->| Z^-1 |-->(+)-->y(n)
54  %          /\               |
55  %           --------<*a------
56  %
57  B = 0.004;
58  p_dec = p;
59  state_1 = 0;
60  state_2 = 0;
61  c = .997;
62
63  for i=1:N
64      if p == 0
65          break
66      end
67
68      if plot_output == 1
69          %need to rerrange due to the pointer implementation
70
71         plot(dl_1);
72         pause
73      end
74
75      if (i<length(exciter))
76          dl_1(dl_1_ptr_1) = exciter(i) + dl_1(dl_1_ptr_1);
77          dl_2(dl_2_ptr_1) =  exciter(i) + dl_2(dl_2_ptr_1);
78      end
79
80      % implementing the all pass filter
81
82      if dl_1_bridge_ptr > dl_1_nut_ptr
83          for j=dl_1_nut_ptr:dl_1_bridge_ptr
84              old_val_1 = dl_1(j);
85              dl_1(j) = state_1 + c * dl_1(j);
86              state_1 = dl_1(j) - c * old_val_1;
87
88              old_val_2 = dl_2(j);
89              dl_2(j) = state_2 + c * dl_2(j);
90              state_2 = dl_2(j) - c * old_val_2;
91
92          end
93      else
94          for j=dl_1_bridge_ptr:dl_1_nut_ptr
95              old_val_1 = dl_1(j);
96              dl_1(j) = state_1 + c * dl_1(j);
97              state_1 = dl_1(j) - c * old_val_1;
98
99              old_val_2 = dl_2(j);
100             dl_2(j) = state_2 + c * dl_2(j);
101             state_2 = dl_2(j) - c * old_val_2;
102
```

```
        end
    end

    %%%%%%%%%%%%%%%%%%%%%
    %string boundary conditions
    %%%%%%%%%%%%%%%%%%%%%

    % perfect reflection

    nut = - 1 * dl_1(dl_1_nut_ptr);
    %dl_1(dl_1_nut_ptr) = 0;

    dl_2(dl_2_nut_ptr) = nut;

    %%%

    bridge = - .99 * dl_2(dl_2_bridge_ptr);
    %dl_2(dl_2_bridge_ptr) = 0;

    dl_1(dl_1_bridge_ptr) = bridge;

    % Output

    out(i) = 0.5 * (0.5 * dl_1(dl_1_ptr_1) + 0.5 * dl_2(dl_2_ptr_1)); %linear interpolation for output
    out(i) = out(i) + 0.5 * (0.5 * dl_1(dl_1_ptr_2) + 0.5 * dl_2(dl_2_ptr_2)); %linear interpolation for output


    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    %update and wrap pointers
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
    %for delay line 1
    dl_1_ptr_1 = 1 + dl_1_ptr_1; if dl_1_ptr_1 > p, dl_1_ptr_1 = 1; end
    dl_1_ptr_2 = 1 + dl_1_ptr_2; if dl_1_ptr_2 > p, dl_1_ptr_2 = 1; end
    %for delay line 2
    dl_2_ptr_1 = dl_2_ptr_1 - 1; if dl_2_ptr_1 < 1, dl_2_ptr_1 = p; end
    dl_2_ptr_2 = dl_2_ptr_2 - 1; if dl_2_ptr_2 < 1, dl_2_ptr_2 = p; end
    %for bridge_ptr
    dl_1_bridge_ptr = 1 + dl_1_bridge_ptr; if dl_1_bridge_ptr > p, dl_1_bridge_ptr = 1; end
    dl_2_bridge_ptr = dl_2_bridge_ptr - 1; if dl_2_bridge_ptr < 1, dl_2_bridge_ptr = p; end
    %for nut ptr
    dl_1_nut_ptr = 1 + dl_1_nut_ptr; if dl_1_nut_ptr > p, dl_1_nut_ptr = 1; end
    dl_2_nut_ptr = dl_2_nut_ptr - 1; if dl_2_nut_ptr < 1, dl_2_nut_ptr = p; end




end

sound(out,Fs)


% plotting output

plot(linspace(0,N/Fs,length(out)),out)
title('Ouput waveform')
xlabel('Time (s)')
ylabel('Amplitude')
figure(2)

y = fft(out);
P2 = abs(y/N);
P1 = P2(1:N/2+1);
P1(2:end-1) = 2*P1(2:end-1);
P1 = 10*log10(P1);
f = Fs*(0:(N/2))/N;
plot(f(1:4400),P1(1:4400))
title('Single-Sided Amplitude Spectrum of model')
```

```
171  xlabel('f (Hz)')
172  ylabel('|dB(f)|')
173  hold on
174  % plot line of intended frequency
175  plot([freq freq],[min(P1) max(P1)])
176  hold off
177
```

14.1.3 Flute

```
1   clear; close all
2
3   %                                          based                                          off
    https://quod.lib.umich.edu/cache//b/b/p/bbp2372.1992.072/bbp2372.1992.072.pdf#page=1;zoom
    =75
4
5   Fs = 44100;
6   freq = 440; %freq of note
7
8   p = round(Fs/freq); %delay line length
9
10  tmax = 5;
11
12  N = tmax*Fs; % length of sound
13  plot_output = 0;
14
15  %%% breath input %%%
16
17  breath_len = 1; % in seconds
18
19  % envelope
20
21  %-------------------------------------------------
22  % set envelope for pressure vector
23
24  attack = 40; % attack time, 1/attack seconds
25  sustain = 2; % sustain, seconds
26  release = 5; % release time, 1/release seconds
27  sus_level = 0.7; % sustain level, >0.37 < 1
28  % calculated variables
29  dex= ((Fs/freq)-0.7)/4;
30  delay = floor(dex); % delay time (samples)
31  frac = dex - delay;
32  sa = round(Fs/attack); % attack end sample
33  ss = round(Fs*sustain); % sustain end sample
34  rs = round(Fs/release); % release end sample
35
36
37  % attack
38  pu(1:sa) = linspace(0,sus_level,sa);
39  % sustain
40  pu(sa+1:sa+1+ss) = sus_level;
41  % release
42  pu(sa+2+ss:sa+1+ss+rs) = linspace(sus_level,0,rs);
43  % set iterations as length of envelope
44  N_p = length(pu);
45  % initialise output vector
46  N_p= length(pu);
47  n= 0:N_p-1;
48  u = 2*rand(1,N_p)-1; % definition of u, the white noise input
49  vib= sin(2*pi*4.5*n/Fs); % definition of sine wave for vibrato
50  mult = zeros(1,N_p);
51  mult2= zeros(1, N_p);
52  % loop to setup the mouth pressure.
53  for i = 1:N_p
54  mult(i)= u(i)*pu(i);
55  mult2(i) = vib(i)*pu(i);
56  real_in(i)= pu(i)+(0.0085*mult(i))+0.008*vib(i); % create the actual input
57  end
```

```
58
59  exciter = real_in;
60  %exciter = ones(1,N_p);
61  %%% resonator %%%
62
63  % dl2
64  % short delay line (for jet reed)
65  dl_2_len = round(p/2);
66  dl_2 = zeros(1,dl_2_len);
67
68  dl_2_ptr_in = dl_2_len;
69  dl_2_ptr_out = 1;
70
71  flute_out_old = 0;
72  reed_out_old = 0;
73  filter_coef = 0.6;
74
75  % dl1
76  dl_1_len = p;
77  dl_1 = zeros(1,dl_1_len);
78
79  dl_1_ptr_in = dl_1_len;
80  dl_1_ptr_out = 1;
81
82  dl_1_out = 0;
83
84  % out
85
86  out = zeros(1,N);
87
88  % reflection coefficients
89
90  r = 0.42;
91  r2 = 0.53;
92
93  for i=1:N
94      if i<(N_p)
95          reed_in = exciter(i);
96      end
97      % dl2 (reed delay)
98      dl_2(dl_2_ptr_in) = reed_in + r * dl_1_out;
99      dl_2_ptr_in = 1 + dl_2_ptr_in; if dl_2_ptr_in > dl_2_len, dl_2_ptr_in = 1; end
100     dl_2_ptr_out = 1 + dl_2_ptr_out; if dl_2_ptr_out > dl_2_len, dl_2_ptr_out = 1; end
101     reed_out = dl_2(dl_2_ptr_out);
102     dl_2(dl_2_ptr_out) = 0;
103     % summing and adding
104     reed_out = reed_out - reed_out^3; % simulate non linearity of reed
105     reed_out = reed_out + r2 * dl_1_out;
106     % low pass filter
107     % ym = (xm - xm-1) + (0.995 * ym-1)
108     %flute_out = reed_out - reed_out_old + (filter_coef * flute_out_old);
109     flute_out = flute_out_old + filter_coef * (reed_out - flute_out_old);
110     flute_out_old = flute_out;
111     reed_out_old = reed_out;
112
113     % delay line 1
114     dl_1(dl_1_ptr_in) = flute_out;
115     dl_1_ptr_in = 1 + dl_1_ptr_in; if dl_1_ptr_in > dl_1_len, dl_1_ptr_in = 1; end
116     dl_1_ptr_out = 1 + dl_1_ptr_out; if dl_1_ptr_out > dl_1_len, dl_1_ptr_out = 1; end
117     dl_1_out = dl_1(dl_1_ptr_out);
118     out(i) = flute_out;
119 end
120
121
122 % loading in excitation sample
123
124 %filename = 'mallet_2.wav';
125
126 %exciter = audioread(filename);
127 %exciter = rand(1,30000);
128
129 %out = exciter;
```

```
130   out = out(4000:end);
131
132   sound(0.99*out/max(abs(out)), Fs);
133
134   % plotting output
135
136   plot(linspace(0,N/Fs,length(out)),out)
137   title('Ouput waveform')
138   xlabel('Time (s)')
139   ylabel('Amplitude')
140   figure(2)
141
142   y = fft(out);
143   P2 = abs(y/N);
144   P1 = P2(1:N/2+1);
145   P1(2:end-1) = 2*P1(2:end-1);
146   P1 = 10*log10(P1);
147   f = Fs*(0:(N/2))/N;
148   plot(f(1:4400),P1(1:4400))
149   title('Single-Sided Amplitude Spectrum of model')
150   xlabel('f (Hz)')
151   ylabel('|dB(f)|')
152   hold on
153   % plot line of intended frequency
154   plot([freq freq],[min(P1) max(P1)])
155   hold off
```

14.2 MATLAB Code (Utility)

14.2.1 Frequency Plots

```
1  % analyser
2  clear all; clc; close all
3  set(gcf, 'Units', 'Inches', 'Position', [0, 0, 15, 5])
4  filename = 'C:\Users\Valerian\Dropbox\University\2018\fyp\audio clips\off
   synth\A4\string_minor_stiffnesss.wav';
5  [out,Fs] = audioread(filename);
6  freq = 440; % intended frequency
7  subplot(1,2,1)
8  N = length(out);
9  c = [0 0.4470 0.7410];
10 plot(linspace(0,N/Fs,length(out)),out,'Color',c)
11 title('Ouput waveform')
12 xlabel('Time (s)')
13 ylabel('Amplitude')
14 subplot(1,2,2)
15 y = fft(out);
16 P2 = abs(y/N);
17 P1 = P2(1:N/2+1);
18 P1(2:end-1) = 2*P1(2:end-1);
19 P1 = 10*log10(P1);
20 f = Fs*(0:(N/2))/N;
21 semilogx(f(1:length(f)),P1(1:length(f)))
22 title('Single-Sided Amplitude Spectrum of model')
23 xlabel('Frequency (Hz)')
24 ylabel('Amplitude |dB(f)|')
25 xlim([40 10000])
26 ylim([-60 0])
27 hold on
28 %plot line of intended frequency
29 plot([freq freq],[min(P1) max(P1)])
30 hold off
```

14.2.2 Convert image to screen format

```
1  % input a monochrome gif image
2  [input, cmap] = imread('blow.gif', 'Frames', 'all');
```

```
% show image
imshow(input,[0,1]);
%output file
file = fopen('out.txt','w');
input = rot90(input);
input = flip(input);
flat = reshape(input,[],1);
z = floor(length(flat)/32);
i = 1;
j = 1;
out = zeros(z,32);
for a = 1:z
    out(i,:) = flat(j:j+31)';
    fprintf(file,'0x%s, ',bin2hex(sprintf('%d%d', out(a,:))));
    if mod(i,4) == 0
        fprintf(file,'\n');
    end
    j = j+32;
    i = 1 + i;
end

```

14.3 C code (synth)

All source code is available at: https://github.com/valmcc/pmsynth

Below are some of the implemented models:

14.3.1 1D Waveguide

```
//--------------------------------------------------------------------------
-----
/*

Waveguide synth

*/
//--------------------------------------------------------------------------
-----

#include "pmsynth.h"
#include "utils.h"

#define DEBUG
#include "logging.h"

//--------------------------------------------------------------------------
-----

// location of pickup (where the output is extracted from the delay line)
#define WG_PICKUP_POS 4

//--------------------------------------------------------------------------
-----


void wg_gen(struct wg *osc, float *out, size_t n) {
    float am[n];
    adsr_gen(&osc->adsr, am, n);
    for (size_t i = 0; i < n; i++) {
        // pointer edition


```

```
31          if (i % osc->downsample_amt == 0){
32              float mallet_out = 0;
33              if (osc->estate == 1){
34                  mallet_out = impulse_gen(osc);
35                  osc->delay_l[osc->x_pos_l] += mallet_out;
36                  osc->delay_r[osc->x_pos_r] += mallet_out;
37                  //if (osc->impulse_solo){
38                  //  out[i] = mallet_out;
39                  //}
40              }
41
42              //DBG("epos=%d, etate=%d, mallet_out=%d\r\n",osc->epos,osc-
    >estate, (int) (mallet_out*1000));
43              // nut reflection
44              osc->delay_r[osc->bridge_pos] = osc->tube * osc->delay_l[osc-
    >nut_pos];
45              // bridge reflection
46              osc->delay_l[osc->bridge_pos] =  osc->r * osc->delay_r[osc-
    >nut_pos];
47
48              // all pass filter for stiffness (when used for pitch correc-
    tion it changes the "stiffness")
49              osc->delay_l[osc->x_pos_l_2] = osc->a * osc->delay_l[osc-
    >x_pos_l_2] +
50                                             osc->delay_l[osc->x_pos_l] -
51                                             osc->a * osc->delay_l[osc->x_pos_l];
52
53
54              // with linear interp
55              float frac = (float) 1.0f - osc->delay_len_frac;
56              osc->delay_l[osc->x_pos_l] = (1.0f - frac) * osc->delay_l[osc-
    >x_pos_l]+
57                                           (frac) * osc->delay_l[osc->x_pos_l_2];
58              osc->delay_r[osc->x_pos_r] = (1.0f - frac) * osc->delay_r[osc-
    >x_pos_r]+
59                                           (frac) * osc->delay_r[osc->x_pos_r_2];
60
61
62              // added scaling factor for frac due to linear interp varying
    amplitudes due to low pass effect
63
64              out[i] = mallet_out * osc->impulse_solo + (
65                  ((osc->velocity) / 0.8f + 0.2f) * 0.75f * (frac) * (osc-
    >delay_l[osc->x_pos_l] +
66                  osc->delay_r[osc->x_pos_r]))*(1.0f - osc->impulse_solo);
67
68
69
70              //stepping and wrapping pointers
71              osc->x_pos_l += 1;
72              if (osc->x_pos_l > osc->delay_len){
73                  osc->x_pos_l = 0;
74              }
75
76              osc->x_pos_l_2 += 1;
77              if (osc->x_pos_l_2 > osc->delay_len){
78                  osc->x_pos_l_2 = 0;
79              }
80
81              osc->x_pos_r += 1;
82              if (osc->x_pos_r > osc->delay_len){
83                  osc->x_pos_r = 0;
84              }
85
```

```
86              osc->x_pos_r_2 += 1;
87              if (osc->x_pos_r_2 > osc->delay_len){
88                  osc->x_pos_r_2 = 0;
89              }
90
91              osc->bridge_pos += 1;
92              if (osc->bridge_pos > osc->delay_len){
93                  osc->bridge_pos = 0;
94              }
95
96              osc->nut_pos += 1;
97              if (osc->nut_pos > osc->delay_len){
98                  osc->nut_pos = 0;
99              }
100
101             osc->pickup_pos += 1;
102             if (osc->pickup_pos > osc->delay_len){
103                 osc->pickup_pos = 0;
104             }
105         } else
106         out[i] = out[i] * 0.5f + out[i-1] *0.5f; // linear interp
107         osc->epos += 1; // incrementing impulse sample
108     }
109     block_mul(out, am, n);
110     //svf2_gen_lpf(&osc->opf, out, out, n, FILT_LOW_PASS); //TODO remove
111 }
112
113 //-------------------------------------------------------------------------
    -----
114
115 void wg_excite(struct wg *osc) {
116     // On each time interval, insert the next sample point of the exciter
117     // sample into a point of the delay line.
118
119     // for now it sets a flag for wg_gen to begin adding this to the delay
120     // line
121     osc->estate = 1;
122     osc->epos = 0;
123
124     // setting up the positions of the various parts of the delay line
125     //
126     // bridge                  pickup         exciter            nut
127     // |                         x              x                |
128     // ------------------------------------------------------------
129     //
130     //
131     //osc->excite_pos = 3;
132     osc->x_pos_l = osc->excite_pos;
133     osc->x_pos_r = (osc->delay_len) - osc->excite_pos;
134     osc->x_pos_l_2 = osc->x_pos_l + 1;
135     osc->x_pos_r_2 = osc->x_pos_r + 1;
136     osc->bridge_pos = osc->delay_len;
137     osc->nut_pos = 0;
138     osc->pickup_pos = WG_PICKUP_POS;
139
140
141
142 }
143
144
145 //-------------------------------------------------------------------------
    -----
146
147 void wg_ctrl_impulse_type(struct wg *osc, int impulse) {
```

```
148     osc->epos = 0;
149     osc->estate = 0;
150     osc->impulse = impulse;
151 }
152 
153 void wg_ctrl_reflection(struct wg *osc, float reflection) {
154     osc->r = reflection;
155 }
156 
157 void wg_ctrl_frequency(struct wg *osc, float freq) {
158     osc->freq = freq;
159     osc->delay_len_total = (AUDIO_FS/freq/2.0f/osc->downsample_amt)+1;
160     osc->delay_len = (uint32_t) osc->delay_len_total; // delay line length
161     osc->delay_len_frac = osc->delay_len_total - (float) osc->delay_len;
162     //DBG("delay length: %d\r\n", osc->delay_len);
163 }
164 
165 void wg_ctrl_stiffness(struct wg *osc, float stiffness) {
166     osc->a = stiffness;
167 }
168 
169 void wg_ctrl_pos(struct wg *osc, float excite_loc) {
170     osc->excite_loc = excite_loc;
171     osc->excite_pos = excite_loc * osc->delay_len_total;
172     // needs to update position for current voices
173 }
174 
175 void wg_ctrl_impulse_solo(struct wg *osc, int impulse_solo) {
176     osc->impulse_solo = impulse_solo;
177 }
178 
179 void wg_set_velocity(struct wg *osc, float velocity) {
180     osc->velocity = velocity;
181     // velocity adjusts volume and brightness
182 }
183 
184 void wg_set_samplerate(struct wg *osc, float downsample_amt) {
185     osc->downsample_amt = downsample_amt;
186 }
187 
188 void wg_exciter_type(struct wg *osc, int exciter_type) {
189     switch(exciter_type){
190         case 0: //struck string
191             osc->tube = -1;
192             break;
193         case 1: // struck tube
194             osc->tube = 1;
195             break;
196         default:
197             break;
198     }
199 
200 }
201 
202 void wg_init(struct wg *osc) {
203     // setting all pass values
204     osc->ap_state_1 = 0.0f;
205     osc->ap_state_2 = 0.0f;
206     osc->downsample_amt = 1;
207 }
208 
209 //-------------------------------------------------------------------------
-----
```

14.3.2 Flute Model

```
//--------------------------------------------------------------------------
-----
/*

Woodwind modelling (with the use of waveguides)

*/
//--------------------------------------------------------------------------
-----

#include "pmsynth.h"
#include "utils.h"

#define DEBUG
#include "logging.h"
#include "display.h"
#include "lcd.h"


//--------------------------------------------------------------------------
-----

#define R_1 0.42f
#define R_2 0.53f
#define FILTER_COEF 0.6f
#define DC_FILT_GAIN 0.99f

//--------------------------------------------------------------------------
-----

void ww_gen(struct ww *osc, float *out, size_t n) {

    // input generation
    //-----------------------------------------------------------------------
---------

    float am[n];
    float breath[n];
    float vibrato[n];

    sin_gen(&osc->vibrato, vibrato, NULL, n);
    adsr_gen(&osc->adsr, am, n);
    noise_gen_white(&osc->ns, breath, n);

    block_mul(breath, am, n); // white noise following adsr
    //block_mul_k(breath, 0.0085f, n); // scaling white noise
    block_mul_k(breath, osc->noise_amt, n); // scaling white noise

    //block_mul_k(vibrato, 0.008f, n); // scaling vibrato
    block_mul_k(vibrato, osc->vibrato_amt, n); // scaling vibrato
    block_add(breath, vibrato, n); // adding vibrato to white noise

    block_add(breath, am, n); // adding to adsr to make pressure input

/*   pressure input looks a bit like:

         _-_-_-_-_-_-_
       /              \
      /                \
     /                  \ (with dc offset)*/

```

```
57
58      // delay line calcs
59      for (size_t i = 0; i < n; i++) {
60
61              if (i % osc->downsample_amt == 0){
62              // delay line 2 (for jet reed)
63              osc->dl_2[osc->dl_2_ptr_in] = breath[i] + osc->r_1 * osc-
    >dl_1_out;
64
65              // stepping and wrapping pointers for delay line 2
66              osc->dl_2_ptr_in += 1;
67              if (osc->dl_2_ptr_in > osc->dl_2_len){
68                  osc->dl_2_ptr_in = 0;
69              }
70
71              osc->dl_2_ptr_out += 1;
72              if (osc->dl_2_ptr_out > osc->dl_2_len){
73                  osc->dl_2_ptr_out = 0;
74              }
75
76              float reed_out = osc->dl_2[osc->dl_2_ptr_out];
77
78              // summing and adding
79              reed_out = reed_out - (pow2(reed_out) * reed_out);
80              reed_out = reed_out + osc->r_2 * osc->dl_1_out;
81
82              // low pass filter
83              float flute_out = osc->flute_out_old + osc->lp_filter_coef *
    (reed_out - osc->flute_out_old);
84              osc->flute_out_old = flute_out;
85
86              // for delay line 1
87              osc->dl_1[osc->dl_1_ptr_in] = flute_out;
88
89              // stepping and wrapping pointers for delay line 1
90              osc->dl_1_ptr_in += 1;
91              if (osc->dl_1_ptr_in > osc->dl_1_len){
92                  osc->dl_1_ptr_in = 0;
93              }
94
95              osc->dl_1_ptr_out += 1;
96              if (osc->dl_1_ptr_out > osc->dl_1_len){
97                  osc->dl_1_ptr_out = 0;
98              }
99
100             // setting output
101             osc->dl_1_out = osc->dl_1[osc->dl_1_ptr_out];
102
103             // outputting audio
104             out[i] = flute_out;
105
106             // dc block
107
108             osc->dc_filt_out = flute_out - osc->dc_filt_in + (DC_FILT_GAIN
    * osc->dc_filt_out);
109             osc->dc_filt_in = flute_out;
110             out[i] = osc->dc_filt_out;
111             //out[i] = breath[i];
112         } else
113         out[i] = out [i-1]; // sample and hold (least cpu)
114     }
115     block_mul_k(out, (osc->velocity / 0.8f + 0.2f), n);
116 }
117
```

```
118 //------------------------------------------------------------------------
    -----
119
120 void ww_blow(struct ww *osc) {
121     osc->estate = 1;
122
123     // delay line 2 (for jet reed)
124     osc->dl_2_ptr_in = osc->dl_2_len;
125     osc->dl_2_ptr_out = 0;
126
127     // delay line 1 (reflection from the output hole)
128
129     osc->dl_1_ptr_in = osc->dl_1_len;
130     osc->dl_1_ptr_out = 0;
131
132
133 }
134
135 //------------------------------------------------------------------------
    -----
136
137 void ww_set_velocity(struct ww *osc, float velocity) {
138     osc->velocity = velocity;
139     // velocity adjusts volume
140 }
141
142 void ww_update_vib_noise(struct ww *osc, float vibrato_amt, float
    noise_amt) {
143     osc->vibrato_amt = vibrato_amt;
144     osc->noise_amt = noise_amt;
145 }
146
147 void ww_set_samplerate(struct ww *osc, float downsample_amt) {
148     osc->downsample_amt = downsample_amt;
149 }
150
151 void ww_update_coefficients(struct ww *osc, float lp_filter_coef, float
    r_1, float r_2) {
152     osc->lp_filter_coef = lp_filter_coef;
153     osc->r_1 = r_1;
154     osc->r_2 = r_2;
155 }
156
157 void ww_ctrl_frequency(struct ww *osc, float freq) {
158     osc->freq = freq;
159
160     osc->dl_1_len_total = (AUDIO_FS/freq)/osc->downsample_amt;
161     osc->dl_1_len = (uint32_t) osc->dl_1_len_total/2.0f; // delay line 1
    length
162     osc->dl_1_len_frac = osc->dl_1_len_total - (float) osc->dl_1_len;
163
164     osc->dl_2_len_total = osc->dl_1_len_total/4.0f;
165     osc->dl_2_len = (uint32_t) osc->dl_2_len_total; // delay line 2 length
166     osc->dl_2_len_frac = osc->dl_2_len_total - (float) osc->dl_2_len;
167
168     DBG("freq: %d, delay 1 length: %d, delay 2 length: %d\r\n", (int)freq,
    osc->dl_1_len, osc->dl_2_len);
169
170 }
171
172 void ww_init(struct ww *osc) {
173     osc->downsample_amt = 1;
174 }
175
```

```
176 //------------------------------------------------------------------------
    -----
```

14.3.3 Banded Waveguide

```
  1 //------------------------------------------------------------------------
    -----
  2 /*
  3
  4 Banded waveguide model
  5
  6 */
  7 //------------------------------------------------------------------------
    -----
  8
  9 #include "pmsynth.h"
 10 #include "utils.h"
 11
 12 #define DEBUG
 13 #include "logging.h"
 14 #include <math.h>
 15 //------------------------------------------------------------------------
    -----
 16
 17 void wgb_gen(struct wgb *osc, float *out, size_t n) {
 18     float am[n];
 19     adsr_gen(&osc->adsr, am, n);
 20     for (size_t i = 0; i < n; i++) {
 21         out[i] = 0.0f;
 22
 23         for (size_t j = 0; j < NUM_MODES; j++) {
 24             if (i % osc->mode[j].downsample_amt == 0){
 25                 // mallet hit
 26                 float mallet_out = 0;
 27                 if (osc->estate == 1){
 28                     mallet_out = impulse_gen_wgb(osc);
 29                     osc->mode[j].delay[osc->mode[j].dl_ptr_in] += mal-
    let_out;
 30                     // uncomment for direct mallet output
 31                     //out[i] = mallet_out;
 32                 }
 33
 34                 out[i] += osc->mode[j].delay[osc->mode[j].dl_ptr_out]*osc-
    >mode[j].mix_factor;
 35                 svf2_gen(&osc->mode[j].bpf, &osc->mode[j].delay[osc-
    >mode[j].dl_ptr_out], &osc->mode[j].delay[osc->mode[j].dl_ptr_in], 1,
    FILT_BAND_PASS);
 36
 37                 //-----------------------------------------------------
    -
 38                 // linear interp for tuning, currently only applies to
    lowest harmonic to save cpu
 39                 if (j == 0){
 40                 float frac = (float) osc->mode[j].delay_len_frac;
 41
 42                 osc->mode[j].delay[osc->mode[j].dl_ptr_lin_tuner_2] =
    (frac) * osc->mode[j].delay[osc->mode[j].dl_ptr_lin_tuner_1] + (1.0f -
    frac) * osc->mode[j].delay[osc->mode[j].dl_ptr_lin_tuner_2];
 43
 44                 // wrapping linear interp pointers
 45                 osc->mode[j].dl_ptr_lin_tuner_1 += 1;
 46                     if (osc->mode[j].dl_ptr_lin_tuner_1 > osc->mode[j].de-
    lay_len){
 47                         osc->mode[j].dl_ptr_lin_tuner_1 = 0;
 48                     }
```

```
49                  osc->mode[j].dl_ptr_lin_tuner_2 += 1;
50                      if (osc->mode[j].dl_ptr_lin_tuner_2 > osc->mode[j].de-
    lay_len){
51                          osc->mode[j].dl_ptr_lin_tuner_2 = 0;
52                      }
53                  }
54                  //-------------------------------------------------------
    --
55
56                  osc->mode[j].dl_ptr_in += 1;
57                      if (osc->mode[j].dl_ptr_in > osc->mode[j].delay_len){
58                          osc->mode[j].dl_ptr_in = 0;
59                      }
60                  osc->mode[j].dl_ptr_out += 1;
61                      if (osc->mode[j].dl_ptr_out > osc->mode[j].delay_len){
62                          osc->mode[j].dl_ptr_out = 0;
63                          // mixing between modes
64                          osc->mode[j].delay[osc->mode[j].dl_ptr_out] =
    (osc->mode_mix_amt) * out[i] + (1.0f - osc->mode_mix_amt) * osc-
    >mode[j].delay[osc->mode[j].dl_ptr_out];
65                      }
66                  } else {
67                  out[i] = out[i] * 0.5f + out[i-1] *0.5f; // linear interp
    on output when downsampling
68                  osc->epos += 1; // incrementing impulse sample
69                  }
70              }
71          //all pass filter for tuning!
72          // float temp = out[i];
73          // out[i] = - osc->a * out[i]+ osc->ap_old_in + osc->a * osc-
    >ap_old_out;
74          // osc->ap_old_out = out[i];
75          // osc->ap_old_in = temp;
76      }
77      block_mul(out, am, n);
78
79      // low pass linked to envelope ended up being too cpu intensive so was
    removed
80      //svf2_gen_lpf(&osc->opf, out, out, n, FILT_LOW_PASS);
81
82      block_mul_k(out, (osc->velocity / 0.8f + 0.2f), n);
83  }
84
85
86  //-------------------------------------------------------------------------
    -----
87
88  void wgb_pluck(struct wgb *osc) {
89      osc->estate = 1;
90      osc->epos = 0;
91      for (size_t i = 0; i < NUM_MODES; i++) {
92          osc->mode[i].dl_ptr_out = 1;
93          osc->mode[i].dl_ptr_in = 0;
94
95          osc->mode[i].dl_ptr_lin_tuner_1 = 2;
96          osc->mode[i].dl_ptr_lin_tuner_2 = 3;
97      }
98
99  }
100
101 //-------------------------------------------------------------------------
    -----
102
103 void wgb_ctrl_attenuate(struct wgb *osc, float attenuate) {
```

```
104     // do nothing
105 }
106 
107 void wgb_ctrl_frequency(struct wgb *osc, float freq) {
108 
109     osc->freq = freq;
110     osc->mode[0].freq_coef = 1.0f;
111     float mode_1_freq;
112     float mode_2_freq;
113 
114     switch (osc->resonator_type) {
115         case 3:
116             mode_1_freq = 2.756f;
117             mode_2_freq = 5.404f;
118             break;
119         case 4:
120             mode_1_freq = 4.0198391420f;
121             mode_2_freq = 10.718498659f;
122             break;
123         case 5:
124             mode_1_freq = 3.16f;
125             mode_2_freq = 2.24f;
126             break;
127         case 6:
128             mode_1_freq = 1.58f;
129             mode_2_freq = 2.55f;
130             break;
131         default:
132             mode_1_freq = 2.756f;
133             mode_2_freq = 5.404f;
134             break;
135     }
136 
137     float h_mod = osc->h_coef * 0.5f;
138     osc->mode[1].freq_coef = h_mod * mode_1_freq + mode_1_freq;
139     osc->mode[2].freq_coef = h_mod * mode_2_freq + mode_2_freq;
140     osc->mode[0].mix_factor = 1.0f - 0.2f * clampf(osc->brightness,0.8f,
    1.0f);
141     osc->mode[1].mix_factor = 1.2f * clampf(osc->brightness,0.0f, 0.5f);
142     osc->mode[2].mix_factor = 0.3f * osc->brightness;
143 
144     for (size_t i = 0; i < NUM_MODES; i++) {
145 
146         uint32_t delay_len = AUDIO_FS/(osc->mode[i].freq_coef * freq);
147 
148         if (delay_len > WGB_DELAY_SIZE) {
149             uint32_t rough_ds = (delay_len / WGB_DELAY_SIZE) + 1;
150             osc->mode[i].downsample_amt = rough_ds + rough_ds % 2;
151 
152             osc->mode[i].delay_len_total = delay_len / osc-
    >mode[i].downsample_amt;
153 
154             osc->mode[i].delay_len  = (uint32_t) osc->mode[i].de-
    lay_len_total;
155             osc->mode[i].delay_len_frac = osc->mode[i].delay_len_total -
    (float) osc->mode[i].delay_len;
156 
157         } else {
158             osc->mode[i].downsample_amt = 1;
159             osc->mode[i].delay_len_total = delay_len;
160             osc->mode[i].delay_len  = (uint32_t) osc->mode[i].de-
    lay_len_total;
161             osc->mode[i].delay_len_frac = osc->mode[i].delay_len_total -
    (float) osc->mode[i].delay_len;
```

```
162         }
163         //DBG("delay length for mode %d: %d.%d \r\n", i, osc->mode[i].de-
    lay_len, (uint32_t) osc->mode[i].delay_len_frac * 100.0f);
164         svf2_ctrl_cutoff(&osc->mode[i].bpf, osc->mode[i].freq_coef * freq
    * osc->mode[i].downsample_amt);
165         svf2_ctrl_resonance(&osc->mode[i].bpf, 0.4999999f - osc->reflec-
    tion_adjust);
166 
167         //DBG("downsample amount: %d delay length: %d\r\n", osc-
    >mode[i].downsample_amt, osc->mode[i].delay_len);
168 
169     }
170 
171 }
172 
173 //---------------------------------------------------------------------------
    -----
174 
175 void wgb_ctrl_impulse_type(struct wgb *osc, int impulse) {
176     osc->epos = 0;
177     osc->estate = 0;
178     osc->impulse = impulse;
179 }
180 
181 void wgb_set_velocity(struct wgb *osc, float velocity) {
182     osc->velocity = velocity;
183     // velocity adjusts volume
184 }
185 
186 void wgb_ctrl_reflection_adjust(struct wgb *osc, float reflection_adjust)
    {
187     osc->reflection_adjust = reflection_adjust;
188 }
189 
190 void wgb_ctrl_harmonic_mod(struct wgb *osc, float h_coef) {
191     osc->h_coef = h_coef;
192 }
193 
194 void wgb_ctrl_mode_mix_amt(struct wgb *osc, float mode_mix_amt) {
195     osc->mode_mix_amt = mode_mix_amt;
196 }
197 
198 void wgb_ctrl_impulse_solo(struct wgb *osc, int impulse_solo) {
199     osc->impulse_solo = impulse_solo;
200 }
201 
202 void wgb_ctrl_brightness(struct wgb *osc, float brightness) {
203     osc->brightness = brightness;
204 }
205 
206 void wgb_ctrl_resonator_type(struct wgb *osc, int resonator_type) {
207     osc->resonator_type = resonator_type;
208 }
209 
210 void wgb_init(struct wgb *osc) {
211     // do nothing
212 }
213 
214 //---------------------------------------------------------------------------
    -----
```

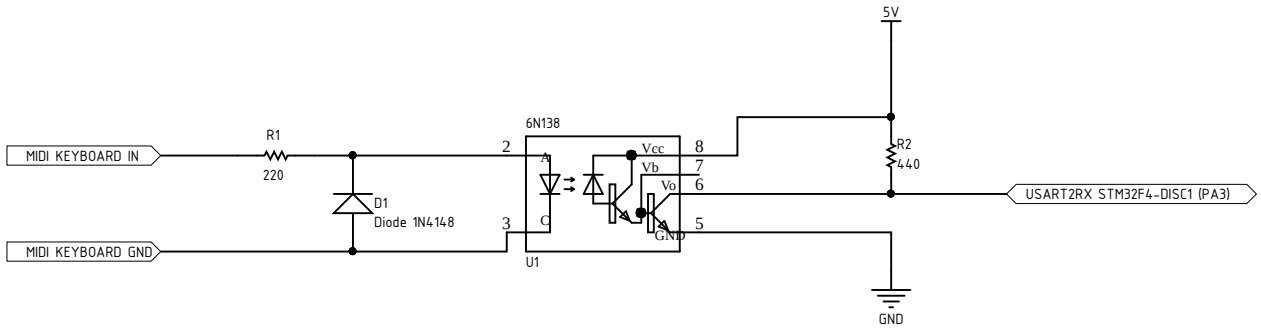
14.3.4 2D Waveguide Mesh

```
//--------------------------------------------------------------------------
-----
/*

2d waveguide grid

*/
//--------------------------------------------------------------------------
-----

#include "pmsynth.h"
#include "utils.h"

#define DEBUG
#include "logging.h"

//--------------------------------------------------------------------------
-----

#define STRIKE_POS_X 2
#define STRIKE_POS_Y 4
#define GRID_SIZE 7
#define DECAY 0.5f // decay between junctions (keep below 0.5)
#define BOUNDARY_REFLN -0.99f

//--------------------------------------------------------------------------
-----

void wg_2d_gen(struct wg_2d *osc, float *out, size_t n) {
    float mallet_out = 0;
    for (size_t i = 0; i < n; i++) {
        if (i % 2 == 0){

            // TICK

            // mallet hit
            if (osc->estate == 1){
                mallet_out = impulse_gen_2d(osc);
            }

            // calculate junction velocity
            for (size_t x = 0; x <= (GRID_SIZE - 1); x++){
                for (size_t y = 0; y <= (GRID_SIZE - 1); y++){
                    osc->mesh[x][y].vJ = DECAY * (osc->mesh[x][y].vE +
                                                  osc->mesh[x][y+1].vW +
                                                  osc->mesh[x][y].vN +
                                                  osc->mesh[x+1][y].vS);
                }
            }
            // add the exciter sample
            osc->mesh[STRIKE_POS_X][STRIKE_POS_Y].vJ += mallet_out;
            // calculate out-travelling waves from junction
            for (size_t x = 1; x <= GRID_SIZE; x++){
                for (size_t y = 1; y <= GRID_SIZE; y++){
                    osc->mesh[x-1][y].vE1 = osc->mesh[x-1][y-1].vJ -
                                            osc->mesh[x-1][y].vW;

                    osc->mesh[x][y-1].vN1 = osc->mesh[x-1][y-1].vJ -
                                            osc->mesh[x][y-1].vS;

```

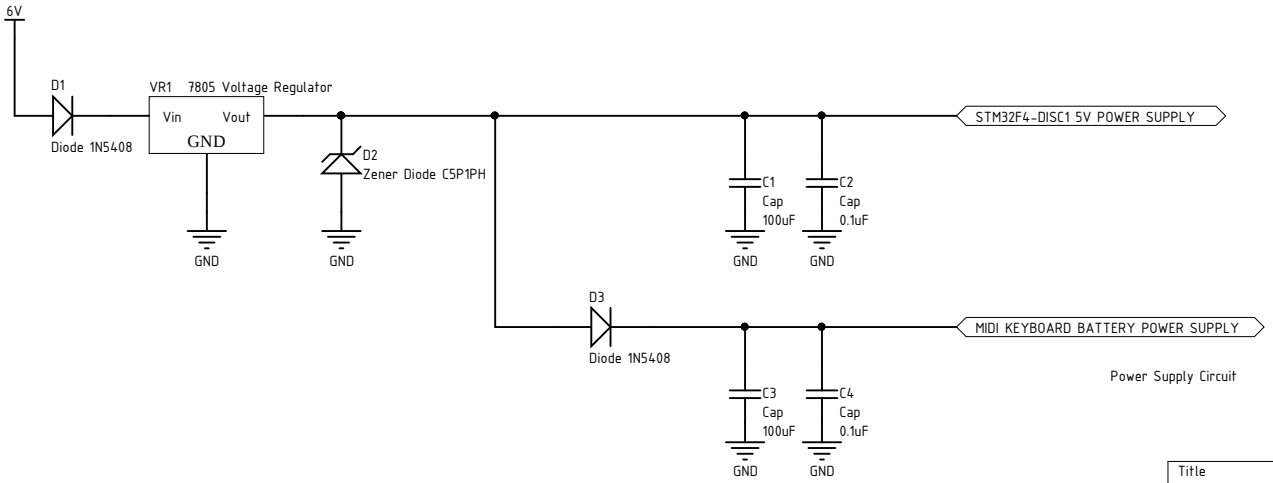
```
                osc->mesh[x-1][y-1].vW1 = osc->mesh[x-1][y-1].vJ -
                                          osc->mesh[x-1][y-1].vE;

                osc->mesh[x-1][y-1].vS1 = osc->mesh[x-1][y-1].vJ -
                                          osc->mesh[x-1][y-1].vN;
            }
        }

    } else{

        // TOCK
        // mallet hit
        if (osc->estate == 1){
            mallet_out = impulse_gen_2d(osc);
        }

        // calculate junction velocity
        for (size_t x = 0; x <= GRID_SIZE - 1; x++){
            for (size_t y = 0; y <= GRID_SIZE - 1; y++){
                osc->mesh[x][y].vJ = DECAY *
                                     (osc->mesh[x][y].vE1 +
                                      osc->mesh[x][y+1].vW1 +
                                      osc->mesh[x][y].vN1 +
                                      osc->mesh[x+1][y].vS1);
            }
        }

        // add the exciter sample
        osc->mesh[STRIKE_POS_X][STRIKE_POS_Y].vJ += mallet_out;

        // calculate out-travelling waves from junction
        for (size_t x = 1; x <= GRID_SIZE; x++){
            for (size_t y = 1; y <= GRID_SIZE; y++){
                osc->mesh[x-1][y].vE = osc->mesh[x-1][y-1].vJ -
                                       osc->mesh[x-1][y].vW1;

                osc->mesh[x][y-1].vN = osc->mesh[x-1][y-1].vJ -
                                       osc->mesh[x][y-1].vS1;

                osc->mesh[x-1][y-1].vW = osc->mesh[x-1][y-1].vJ -
                                         osc->mesh[x-1][y-1].vE1;

                osc->mesh[x-1][y-1].vS = osc->mesh[x-1][y-1].vJ -
                                         osc->mesh[x-1][y-1].vN1;
            }
        }
        // boundary calculations
        for (size_t x = 1; x <= GRID_SIZE; x++){
            osc->mesh[x-1][0].vE = BOUNDARY_REFLN *
                                   osc->mesh[x-1][0].vW1;

            osc->mesh[x-1][GRID_SIZE].vW = BOUNDARY_REFLN *
                                           osc->mesh[x-1][GRID_SIZE].vE1;
        }
        for (size_t y = 1; y <= GRID_SIZE; y++){
            osc->mesh[0][y-1].vN = BOUNDARY_REFLN *
                                   osc->mesh[0][y-1].vS1;

            osc->mesh[GRID_SIZE][y-1].vS = BOUNDARY_REFLN *
                                           osc->mesh[GRID_SIZE][y-1].vN1;
        }

    }
    out[i] = osc->mesh[2][2].vJ;
```

```
121         }
122     }
123 
124     //-------------------------------------------------------------------------
    -----
125 
126     void wg_2d_pluck(struct wg_2d *osc) {
127         // On each time interval, insert the next sample point of the exciter
128         // sample into a point of the delay line.
129 
130         // for now it sets a flag for wg_gen to begin adding this to the delay
131         // line
132         osc->estate = 1;
133         osc->epos = 0;
134     }
135 
136     //-------------------------------------------------------------------------
    -----
137 
138     void wg_2d_ctrl_attenuate(struct wg_2d *osc, float attenuate) {
139         //osc->k = 0.5f * attenuate;
140     }
141 
142     void wg_2d_ctrl_frequency(struct wg_2d *osc, float freq) {
143         //osc->freq = freq;
144         //osc->xstep = (uint32_t) (osc->freq * KS_FSCALE);
145     }
146 
147     void wg_2d_init(struct wg_2d *osc) {
148         // do nothing
149     }
150 
151     //-------------------------------------------------------------------------
    -----
```

14.4 Diagrams and Schematics

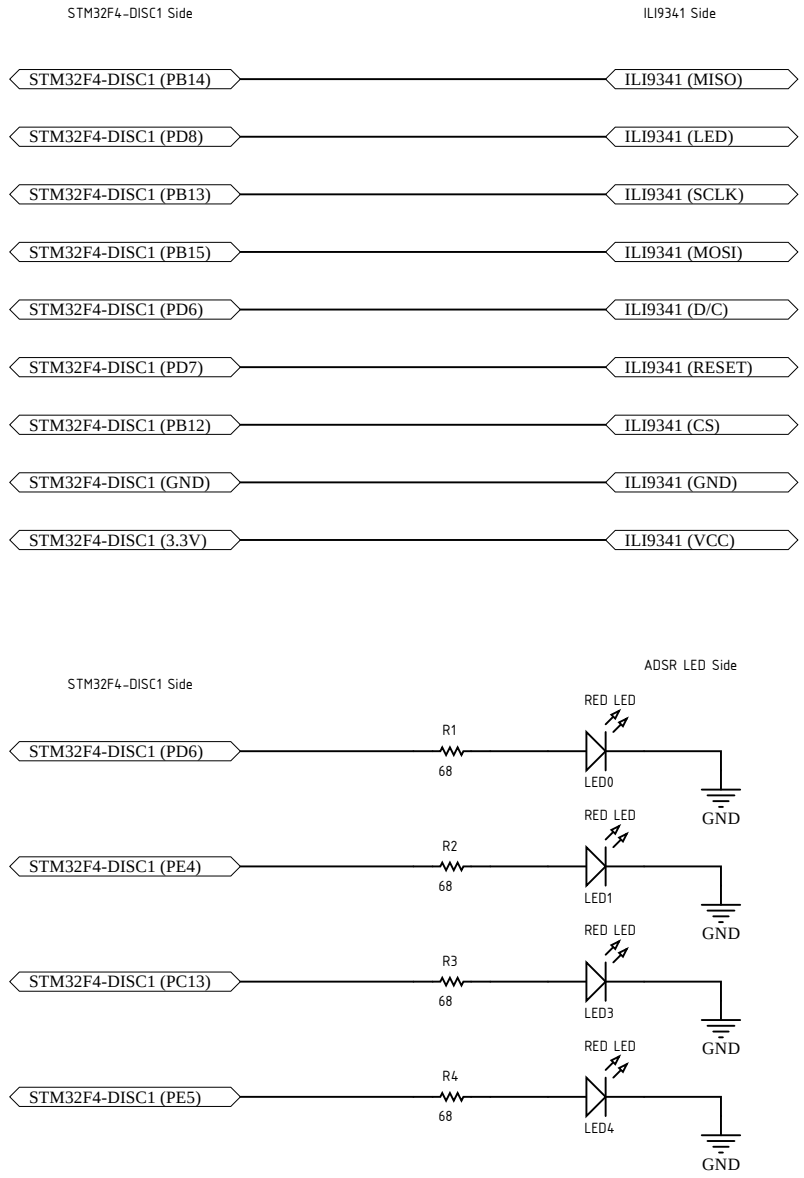


Midi Input Circuit



Power Supply Circuit

Title: Supporting Circuits for Synth Project (ECE4045)		
Size: A4	Number: 01	Revision: P1.1
Date: 20/10/2018		Sheet of 1/2
File: C:\Users\..\midi_circuit.SchDoc		Drawn By: Valerian McCaskill 2602 4608

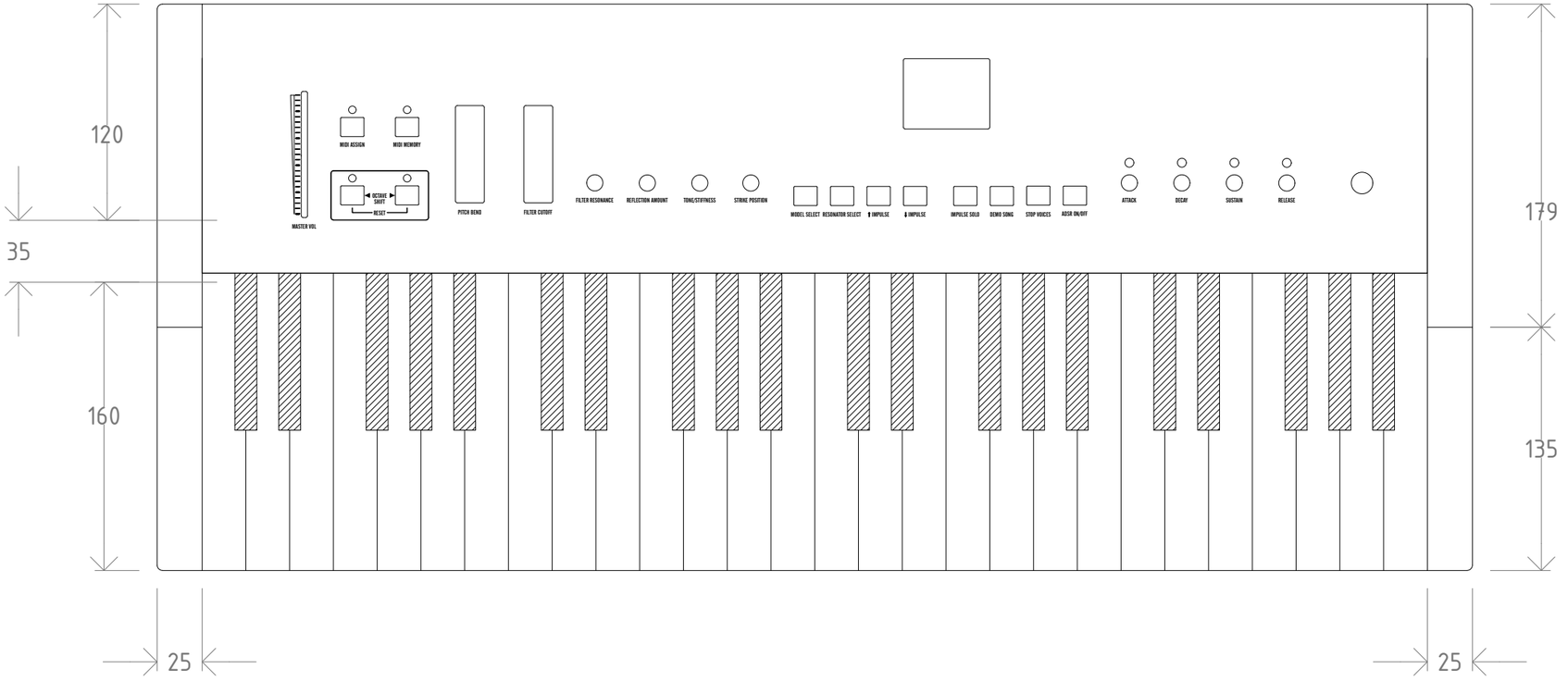


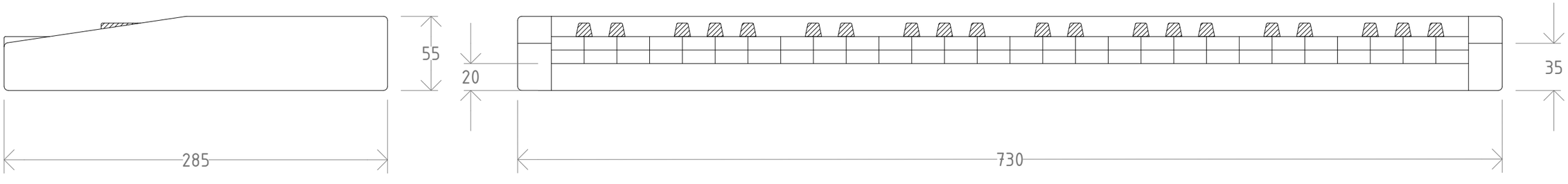
Title: STM32F4 DISCOVERY BOARD CONNECTIONS (ECE4045)		
Size: A4	Number: 01	Revision: P1.0
Date: 20/10/2018		Sheet of 2/2
File: C:\Users\..\stm32f4.SchDoc		Drawn By: Valerian McCaskill 2602 4608

INPUT/OUTPUT OPTIONS

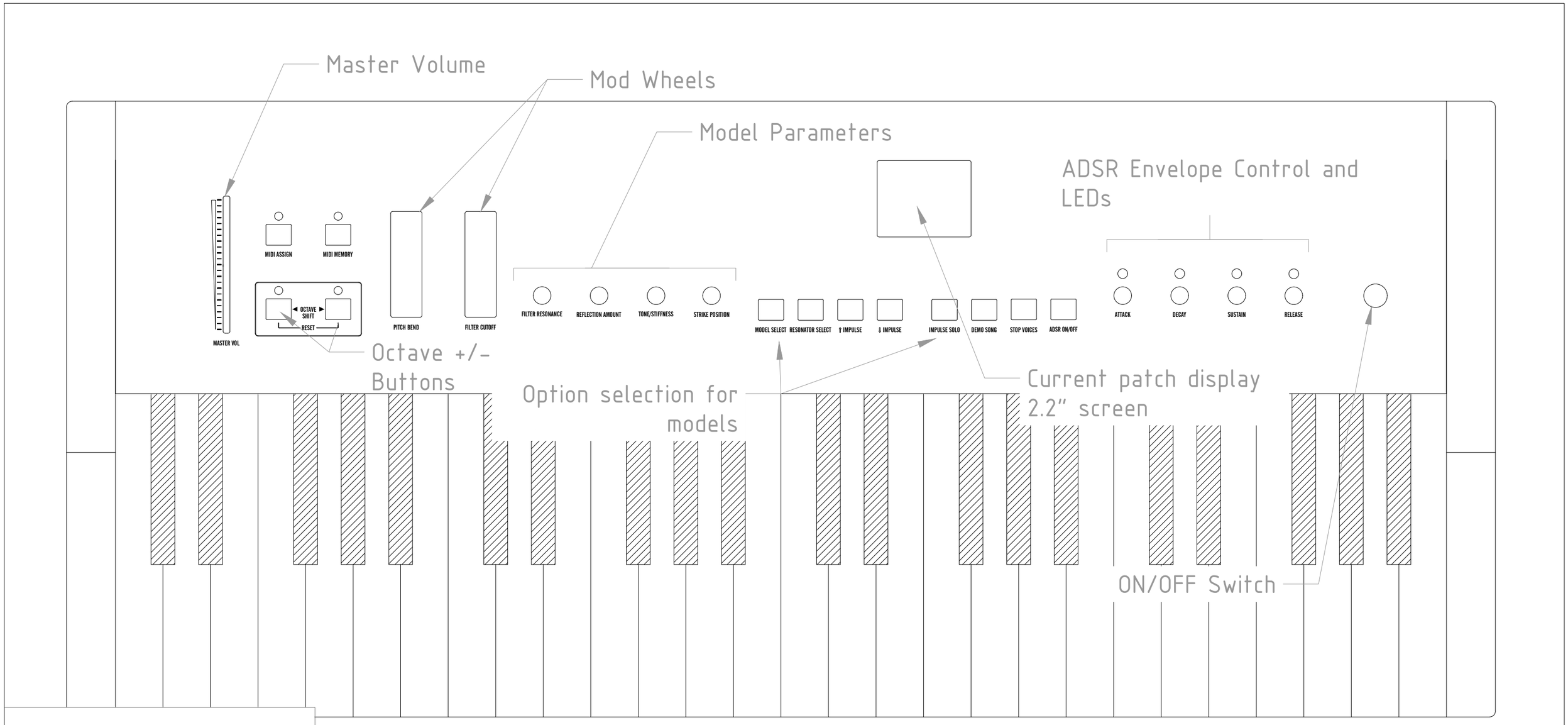
CONNECTOR	PURPOSE
$\frac{1}{4}$ INCH JACK	MASTER OUT
6V PLUG	DC IN
USB TYPE B	USB-MIDI OUT
MINI USB	STM32F4 IN

FRONT PANEL (SEE SHEET 2)





SCALE: 1:80	ECE4094 DESIGN PROJECT	SYNTHESIZER CONSTRUCTION SCHEMATIC	DATE: 16/05/2018	DRAWN: VALERIAN MCCASKILL	REV 04
					SHEET 1/2



PANEL PARTS LIST

ITEM	QTY
ROTARY POTENTIOMETER	8
MOD WHEEL	2
BUTTON	12
SLIDE POTENTIOMETER	1
LEDs	8
2.2" LCD SCREEN	1
ON/OFF SWITCH	1

REV 04

SCALE: 1:50	ECE4094 DESIGN PROJECT	SYNTHESIZER CONSTRUCTION SCHEMATIC	DATE: 16/05/2018	DRAWN: VALERIAN MCCASKILL	SHEET 2/2