Designing Software for a Wavetable Audio Synthesiser

Marco Rademan

21561273

Report submitted in partial fulfilment of the requirements of the module
Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic
Engineering at Stellenbosch University.

Supervisor: Prof J. Versfeld

November 2021

Acknowledgements

Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   I agree that plagiarism is a punishable offence because it constitutes theft.

3. Ek verstaan ook dat direkte vertalings plagiaat is.
   I also understand that direct translations are plagiarism.

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

21561273	
Studentenommer / *Student number*	Handtekening / *Signature*
MW Rademan	November 2021
Voorletters en van / *Initials and surname*	Datum / *Date*

Abstract

English

Hardware and software sound synthesisers have played an important role in music to date, since its inception in 1928. The design and processes behind such products are often proprietary, or not well documented. The aim of this project is to design the wavetable-based software audio generation core for a stereo synthesiser, which is able to produce high quality audio, with the intention of microcontroller implementation.

A system is designed, which includes all the fundamental aspects of modular synthesis: oscillators; filters and cut-off modulation; ADSR envelope control signals; volume modulation; FM synthesis; waveshaping. Look-up tables (LUTs) and wavetable oscillators form the crux of this system. It is able to generate up to a fixed number of notes, using on/off note triggers, ideal for MIDI integration. It can do so efficiently. System and audio tests are performed to verify operation, primarily though the use of spectrograms.

This system can form the backbone for any complex hardware synthesiser system, through the process of well-documented design, and analysis of design choices.

Afrikaans

Vanaf die begin van sintetiseerders in 1928, speel hardeware en sagteware sintetiseerders 'n belangrike rol in musiek. Die onwerp en prosesse agter so projekte is dikwels eie tot die vervaardigers, of nie goed gedokumenteer nie. Die doel van hierdie projek is om 'n golftabel-basseerde sagteware vir 'n stereo sintetiseerder te ontwerp, wat kwaliteit digitale klank kan produseer. Dit is geteiken vir mikroverwerkers.

'n Sisteem wat al die fundamentele aspekte van modulére sintese implementeer, is ontwerp. Die aspekte sluit in: ossillators; filters en afsnypunt modulasie; volume modulasie; FM sintese; golf-vorming. Opsoektabelle (LUTs) en golftabel ossillators vorm die kern van hierdie sisteem. Die sisteem kan 'n vasgestelde aantal note speel deur middel van noot-aan/af kennisgewings, wat ideaal vir MIDI integrasie is. Die sisteem kan sy take doeltreffend uitrig. Verskeie toetse om korekte werking te toets is gedoen, en hoofsaaklik ontleed deur die gebruik van spektrogramme.

Hierdie sisteem beoog om die rugraat van enige komplekse hardeware sintetiseerder sisteem te vorm, deur middel van gedetailleerde dokumentasie en analise van ontwerpskeuses.

Contents

List of Figures

List of Tables

Listings

Nomenclature

Variables and functions

$y[n]$	A discrete-time signal with samples indexed by variable n.
$x_1[n] * x_2[n]$	The convolution of discrete-time functions.
$h[n]$	The impulse response of a discrete-time system.
$X(f)$	The discrete-time Fourier transform (DTFT) of a function.
$\lfloor x \rfloor$	The floor of a variable x, corresponding to the integer component of x.
$\{x\}$	The fractional part of x.
$\{n_1, n_2, ..., n_K\}$	A set of numbers, not to be confused with the fractional function.
η	The frequency scaling factor in wavetable sampling.
$y[n]_{\uparrow L}$	Upsampling a discrete-time signal by a factor L.
$y[n]_{\downarrow M}$	Downsampling a discrete-time signal by a factor M.
$\text{tri}[\frac{n}{L}]$	A discrete triangular pulse beginning at $-L$ and ending at L, with an amplitude of 1.
$\text{rect}[\frac{n}{L}]$	A discrete rectangular pulse beginning at $\frac{-L}{2}$ and ending at $\frac{L}{2}$, with an amplitude of 1.
$\mathbb{N}$	The set of all natural numbers ($\{1, 2, 3, 4, ...\}$).
$\mathbb{N}_0$	The set of all natural numbers including 0.
$\mathbb{R}$	The set of all real numbers.
$\mathbb{Z}$	The set of all integers ($\{..., -2, -1, 0, 1, 2, ...\}$).
$\text{lerp}(x_1, x_2, \delta)$	Linear interpolation between points x_1 and x_2, by a distance factor δ.

Acronyms and abbreviations

VCA	Voltage-controlled amplifier
ADSR	Attack, Decay, Sustain, Release
VCF	Voltage-controller filter
VST	Virtual studio technology
DAW	Digital audio workstation
FM	Frequency modulation
LUT	Lookup table
LFO	Low frequency oscillator
IR	Impulse response
VCO	Voltage-controlled oscillator
CV	Control voltage
LPF	Low-pass filter
HPF	High-pass filter
BPF	Band-pass filter
FIR	Finite impulse response
IIR	Infinite impulse response
MIDI	Musical instrument digital interface
SMF	Standard MIDI file
SQNR	Signal to quantisation noise ratio
PCM	Pulse-code modulation
DPCM	Differential pulse-code modulation
UART	Universal asynchronous receiver/transmitter
SAI	Serial audio interface
DSP	Digital signal processing
LP12	12 dB/octave LPF
HP12	12 dB/octave HPF
BP12	12 dB/octave BPF
LP24	24 dB/octave LPF
HP24	24 dB/octave HPF
MCU	Microcontroller unit
MPU	Microprocessing unit
IC	Integrated circuit
FPU	Floating-point unit
STFT	Short-time Fourier transform
RISC	Reduced instruction set computer
CISC	Complex instruction set computer

Chapter 1

Introduction

1.1. Background and motivation

Hardware synthesisers play a crucial role in all aspects of modern music. It has seen a vast expanse in a variety of product lines across the globe, since the inception of analogue sound synthesisers in 1928. A hardware synthesiser refers to a physical system that generates audio signals, using external control inputs. These inputs can either be pre-programmed, such as a drum-machine, or sent manually by the user, such as a piano keyboard via the MIDI protocol. These audio systems can either be digital, where sound is generated by a MCU/MPU, or analogue, where it is generated by the manipulation of VCOs using a variety of control signals. A few user-specified parameters can often be enough to generate a wide array of interesting and complicated sounds, some of which are iconic and recognisable in old and modern music alike.

Due to the highly competitive nature of the market, as well as the complexity of hardware audio generation systems, much of the design and techniques used in products are often proprietary. Many of the high-end synthesisers also include special features and filters that have been qualitatively designed through an iterative process, as their sound takes priority. Low-level design for MCUs, such as the STM32 Cortex M4 and M7 series, are often poorly documented or analysed. Open-source code [5] for these processors is often available for DIY projects, but is usually minimal and leaves many of the frequency domain effects and MCU optimisations unexplored. This thesis aims to design the software for such a system, in a well-documented and analysed fashion, which can be used as a building block for a wide array of more complicated products.

1.2. Problem statement

The aim of the project is to design wavetable-based audio generation software. The system can play up to a fixed number of notes, each with an arbitrary frequency, by using on/off note triggers. This makes the system compatible with most forms of user note inputs, whether it be a button or MIDI messages. This system is designed conceptually and implemented in C, such that it considers MCU applications. It is especially targeted for ARM-based MCUs, such as the STM32 M4 and M7 series.

High-quality stereo audio is produced, whilst speed and memory consumption is taken into account. The system includes all basic synthesis features: volume modulation; filtering and cut-off modulation; ADSR envelope control signals; FM and waveshaping. This thesis provides a foundation for synthesiser implementation on MCUs, which can also be altered to produce a more complex product. Ergo, the system is designed in such a way as to allow for easy implementation on hardware, while being input-type and hardware agnostic.

1.3. System specification

1.3.1. Scope

A digital hardware synthesiser is a complex system with many interacting parts, which can be broadly split into the following categories:

1. **Synthesis software**: all aspects that encompass the DSP chain of audio.

2. **Driving software**: software drivers for the hardware, which includes MIDI and SAI drivers, user-input conditioning, and inter-processor communication (for multiple cores).

3. **Digital hardware**: the digital components of the PCB, which include the micro-controller, the audio codec and user IO components, such as touch screens, buttons and rotary encoders.

4. **Analogue hardware**: any hardware required to condition the signal into a line-level output.

5. **Communication and data transmission**: hardware concerning data transmission and protocols across the product, such as the screen, MIDI (USB or UART) and any other external device links.

The scope of this thesis is restricted to synthesis software, which excludes the design and/or implementation of any hardware-related code, such as SAI drivers or MIDI decoders. That which cannot be tested and analysed in an entirely software-based workflow, is out of scope. Thus, the audio data that is generated by the system is stored in a file and analysed externally, using MATLAB. The following items are within the scope:

1. The audio-synthesis core software, that is triggered by note-on and note-off information.

2. The emulation of MIDI input by interpreting SMFs for testing.

3. User-parameters that can be changed under test conditions to demonstrate system functionality.

Although none is designed, the hardware is considered in the design of the software. This is to ensure good support for a variety of hardware topologies. Computational efficiency is emphasised in the design, however timing is not tested, since it is hardware dependent (RISC versus CISC, clock frequency, available peripherals, etc.).

1.3.2. Functional specification

The following items specify the audio generation capabilities, which are formulated with reference to figure B.1 in the appendix, along with some of the functionality or features provided by other eurorack modules:

1. Selection of the basic waveforms (sine, triangle, sawtooth and square) and interpolation between them are done, as is common in many wavetable synthesisers (see figure A.2 in the appendix).

2. The stereo width of the audio is specified by detuning 2 additional oscillators per note, and by panning and changing the volume of the additional oscillators.

3. An ADSR envelope, with user-set parameters, controls the volume envelope of a single note over time.

4. A selection of filters (LPF, BPF and HPF) is available. The filters have a user-defined Q, if applicable.

5. The filter cut-off frequency is set relative to the fundamental frequency of a note. This ensures that all notes have the same timbre at different frequencies. An ADSR envelope, with user-set parameters, controls the cut-off frequency of the note over time.

6. Stereo waveshaping is performed on the audio, where a gain value into a waveshaper function is specified by the user. The available waveshaper functions are the hyperbolic tangent and the sinusoid.

7. Sinusoidal vibrato is applied at a specified rate and amount.

8. The instrument is polyphonic, i.e. can play multiple notes at once.

1.3.3. Technical requirements

The functional requirements stipulated in subsection 1.3.2 are translated into technical requirements for the software audio core. Since the hardware aspect is not explicitly considered in design, the system is able to scale

according to hardware requirements and performance, so that a multitude of processors and hardware topologies can benefit from the design. The system is targeted for ARM-based MCUs, especially for STM Cortex M4 and M7, which both have FPUs available. However, the system should also efficiently function on other MCUs with included FPUs. Care is taken when writing mathematically-intensive code, to ensure optimal use of the hardware, such as leveraging FPU instructions, avoiding data duplication, utilising cache memory, etc.

The technical requirements for the system are as follows:

1. An audio system generates buffers of a specified size of stereo audio data efficiently.

2. An arbitrary sampling rate may be specified. Common audio sampling rates [6] such as 44.1, 48 and 88.2 kHz can be used.

3. Care is taken to avoid or minimise aliasing.

4. An arbitrary polyphony number (maximum number of notes) may be specified. A "generator" refers to the object that is generating the audio for a single note.

5. When the number of active notes exceeds the polyphony, the generator containing the oldest playing note is re-triggered.

6. The system managing the generators does so efficiently in $O(N)$ time, where N refers to the polyphony number.

7. FPU/CPU intensive operations, such as the divide or modulus operation, is avoided at all costs, if possible.

8. The DSP chain considers efficiency, and simplifies processing as far as reasonably practical.

9. The core software is written in C, to allow compilation for any MCU, and for full control over memory usage and predictability in the assembly code. Therefore, no object-orientation is used, and functional data manipulation is the prime focus.

10. Processing speed takes priority over memory consumption, as far as reasonably practical.

11. An arbitrary wavetable buffer size may be specified, to provide control over the memory consumption and maximum harmonic content of a waveform.

12. All DSP is performed with single-precision floating-point operations, to allow for full 24-bit audio resolution for DAC conversion, which many SAI codec ICs support [7].

13. Filtering is computationally efficient. This implies that IIR filtering is used, since human hearing is insensitive to phase shifts within the filtered signal.

1.4. Summary of work

The following items summarise the work that is performed in this thesis:

1. A computationally efficient method of doing table-lookups with linear interpolation for periodic signals is created.

2. The effect of frequency scaling and linear interpolation is modelled in the frequency domain, which forms the basis of predicting the effect of higher-order interpolation filters for wavetable frequency scaling.

3. A wavetable harmonic content schema is designed to avoid aliasing for higher frequency notes.

4. FM and inter-wavetable interpolation is explored.

5. The design and efficient coefficient computation for 5 IIR filters is done (LP12, LP24, HP12, HP24, BP12).

6. LUTs for waveshaping are generated, and a technique for anti-aliasing is designed using numerical analysis.

7. An ADSR state-machine that outputs a piecewise-exponential is created.

8. A way to create a note with stereo width is detailed.

9. An efficient way for managing generator objects is devised for note on/off triggers.

1.5. Report overview

Literature overview

The history and operation of sound synthesis is explored in this chapter. An explanation of synthesis concepts and components is detailed by using existing synthesisers and/or eurorack modules as examples. The required musical, audio and mathematical knowledge for this thesis is detailed, which would not be considered general engineering knowledge. This chapter exclusively covers existing techniques and knowledge, with some elaboration when necessary.

Design

This chapter details the design of the audio system. A top-down system specification with a bottom-up component synthesis methodology is applied. All the mathematics and self-developed techniques, as well the application of existing techniques, are documented in this chapter.

System testing

Each component is quantitatively tested, along with a full system test that demonstrates most of its ability. A musical test is also done in a more practical setting, where qualitative comments are provided on audio generated from Beethoven's Moonlight Sonata. Three musicians provide feedback on the generated audio.

Conclusion

The design and its achievements are discussed, with reference to the system specification. Possible improvements and further applications are detailed.

Chapter 2

Literature overview

2.1. A brief history of musical synthesisers

An instrument usually described as a "synthesiser" or "synth" is any electronic device or software that can generate audio signals. These signals are generated through a variety of techniques that have been developed and have evolved throughout the 20th and 21st centuries.

In the 1960s, a non-keyboard style of synthesis - modular synthesis - was popularised by the companies Moog and Buchla [8]. Analogue electronics were used for synthesis, where multiple modules generated control voltages in tandem to modulate parameters. Common modules are oscillators (VCOs); envelope generators (ADSR); filters (VCFs); amplifiers (VCAs); sequencers; waveshapers and noise generators. These remain the fundamental components of most synthesisers to date. Emulation or cloning of original Moog or Buchla hardware, such as the 4-pole ladder VCF, is still sought-after in the current commercial market [9].

In contrast, modern eurorack modules (a standardised modular hardware and electronics specification) offer a wide variety of complex options, which are often analogue or digital-analogue hybrids . An example is the Make Noise MATHS [10] module, which is very popular in the modular synth community [11]. It provides features such as amplification, integration, summation and function generation, and is considered an essential module by many influencers.

Due to the complexity of analogue electronics, most synths were monophonic in the 1960s and 1970s, or possessed very limited polyphony. Each extra note required duplication of electronics, which, in turn, required greater effort in manual tuning, as well as more space. The introduction of digital technology in the 1980s allowed for more flexible polyphony at affordable prices, such as the Yamaha DX7. The DX7 is a very well-known early digital synthesiser released in 1983 [12], which uses FM-synthesis (see section 2.3).

The introduction of more powerful computation led to the development of software synths and VST plugins for DAWs (music production software), which use a variety of synthesis techniques such as FM; additive synthesis; subtractive synthesis; physical modelling and wavetable synthesis. Wavetable synthesis is very popular in all music genres and sound design for film. Some of the most popular instrument VSTs are Serum by Xfer Records [13], Massive by Native Instruments [14], and PIGMENTS by Arturia [15]. The aforementioned VSTs focus on wavetable synthesis with sampling, filtering, parameter modulation and FM capabilities.

2.2. Wavetable synthesis

Wavetable synthesis [16] is a very powerful, efficient and popular technique used in many modern synthesisers, which include VST instruments, keyboards and eurorack modules. A periodic waveform is stored in a table , which is sampled at a specific rate (see sections 3.4.1 and 3.4.2). A variety of these tables can be stored (even created by the user as in Serum [13]) and manipulated by interpolating between wavetables or manipulating the wavetables themselves, such as with folding or adjustable duty cycle. This technique allows for complete freedom in parameter modulation, which makes FM an easy addition to such a system. Figure A.2 in the appendix shows an example of a wavetable VST synthesiser [17].

2.3. An overview of other synthesis techniques

There is a variety of other techniques [16] that are used to generate audio, used by VSTs and hardware synthesisers alike. These techniques are often combined in different ways to create a unique, and musically useful product. The most prominent techniques are:

1. **Additive synthesis**: various sinusoids are added together with different amplitudes and phases to produce a signal. The amplitudes, phases and frequencies may be time-varying as well, which ties into physical modelling techniques. This method can be computationally inefficient, and often requires optimisation, such as look-up tables (LUTs).

2. **Subtractive synthesis**: this technique is very simple and is possible in most synthesisers [14] [13] [17]. It requires an harmonically rich source signal (generated by any means), such as a square wave, which is then filtered to reduce harmonic content.

3. **FM synthesis**: this technique uses the same principles as FM for data communication, except in the audible frequency range. Multiple oscillators modulating each other's frequency, often in a coupled or recursive manner, is common, as is seen with Ableton's Operator [17] plugin; a FM-centric VST. Many non-FM-centric synths also offer a vibrato feature, which requires the use of a dedicated vibrato oscillator (LFO) that slightly modulates the source signal's frequency [18].

4. **Physical modelling**: this method [19] involves simulating the sound source of interest. It is usually separated in continuous models for bowed or blown instruments, or impulsive models, such as struck or picked instruments. A variety of methods can be used, such as IR modelling, analytical simulationm, frequency domain modelling, and waveguide synthesis, for example, the Karplus-Strong plucked string algorithm [20] .

5. **Sampling**: sampling synthesis [19] is the technique of using pre-recorded audio samples to reproduce sounds. An example would be to record every key of a piano at different volumes and assigning a sample to trigger when conditions are met. Sampling often uses large buffers that are not necessarily intended to reproduce a periodic waveform (but sometimes do for continuous sound produced by instruments like flutes). Samples and wavetables can be manipulated and modulated in the same way. This technique is computationally efficient, but may require a large amount of memory to store the samples – often in the order of gigabytes, as for Kontakt libraries [21] [22].

2.4. Basic modular synthesiser building blocks

This overview focuses on modular synthesiser building blocks directly, but is relevant to most forms of synthesis, since most standard modern synthesis products are based on the building blocks popularised by Moog and Buchla [23]. Example modules are shown, discussed, and compared to features present in commercial wavetable synthesisers, and features to be considered for design in this thesis. Refer to appendix B.2 for an explanation of a basic modular synthesiser setup. Figure A.1 in the appendix shows the modules that are used as examples [24] in this section.

2.4.1. The VCO

VCO modules commonly include 1V/octave CV inputs for frequency control and provide the basic waveforms as outputs, either separately via a switch, or simultaneously. They can be analogue or digital in nature and can

use a variety of synthesis techniques to generate their waveforms. They often come with the ability set the offset tuning voltage, and can be used to create FM signals through control voltages. Extra features, such as wave folding, are sometimes also present.

The Doepfer A-111-3 Micro Precision VCO/LFO [25] is an analogue VCO that can also operate in LFO configuration, either with a linear or exponential voltage control. Sync (for phase/frequency syncing) and PWM CV inputs are also available.All the basic waveforms are present, except for the sinusoid which is notoriously difficult to generate with analogue electronics, and is commonly implemented as a high-Q unstable filter [26].

2.4.2. The VCF

The VCF is an incredibly important module that forms the basis of subtractive synthesis techniques. Most synths also offer filtering capabilities, such as the widely used Nord Stage 3 [27].

Filters come in many types, often designed with unique characteristics. This can include special control-voltage behaviour; feedback path saturation to limit resonance, while adding additional harmonics; the ability to achieve exceptionally high Q values that cause purposeful instability, allowing filters to function as a sinusoidal oscillator (which many VCOs do not generate).

Thus, filters for musical applications are usually not designed to be as "clean" and stable as possible; instead, they focus on usability and uniqueness. Filter types can include a switchable LPF, BPF or HPF mode, a ladder filter, 12dB/octave or 24dB/octave varieties and a state variable filter configuration.

The IntelliJel UVCF [26] is a popular state-variable filter that simultaneously outputs a 2-pole low-passed, 2-pole high-passed, and 1-pole band-passed signal, which has a cut-off that can be modulated by 2 separate 1V/octave control voltages. It can also be set to a high Q-value, so that it can act as a sinusoidal VCO, due to filter instability.

2.4.3. The VCA

The VCA has the primary purpose of performing the multiplication of signals for uses in AM and otherwise. It acts as an amplifier with a voltage-controllable gain. It is often used in conjunction with a LFO to create a tremolo effect, or with an ADSR envelope to shape the transient of a signal to emulate bowing or plucking sounds. Many VCOs only output a continuous signal. A VCA is thus required to mute any oscillators that are not triggered. The ring modulation effect can also be achieved by multiplying 2 signals in the audible frequency range with each other.

The MFB VCA [28] is a module that has 3 different inputs and 2 CV inputs that modulate the gain. The interaction between the various inputs is specific to this module. Signal multiplication is trivial in the digital domain.

2.4.4. The ADSR envelope

The ADSR envelope is a critical component in synthesis that is used to achieve realistic sounds. It is often used to modulate filter cut-off to allow for dynamic subtractive synthesis. It is also used to emulate the natural attack and decay characteristics of real instruments by volume modulation. It can be utilised in FM applications to mimic the typical pitch modulation found when striking percussive instruments. Due to the logarithmic nature of human hearing [29], an exponential shape is required of the envelope for a linear change in volume to be perceived.

ADSR envelopes are available in most wavetable synthesisers for parameter modulation, such as Serum, Massive and Ableton's stock Wavetable VST instruments. Many keyboards also include this feature, such as the

Nord Stage 3 [27]. The envelope consists of 4 phases. The curve is initiated with an "on" trigger signal, after which a rising function is observed. Once a threshold is reached, determined by the attack time, the decay state is activated. The decay is specified by a decay time parameter. The function decreases until a sustain level is reached, which is a parameter set by the performer. The sustain level is usually less than the peak achieved by the attack phase. The sustain phase remains constant, until an "off" trigger occurs - initiating the release phase. The release phase is a decreasing function that decays until zero is reached (or close to zero in the case of a RC circuit). This is determined by a release time parameter. The A, D and R phases are usually exponential functions implemented by a RC circuit. This is well suited for AM and FM, since octaves are exponential in nature (doubling in frequency) and human hearing is logarithmic in nature [29].

The Doepfer A-140 ADSR Envelope Generator [1] is a classic envelope generator with a gate CV input, an envelope output, and a negated envelope output. It also has a re-trigger input that allows the "on" trigger to occur again; re-initiating the attack phase, independent of the current phase of the envelope. Typical ADSR envelope behaviour can be seen in figure A.3 in the appendix.

2.5. Prerequisite knowledge

2.5.1. Equal temperament tuning

Various tuning systems have existed since the inception of standardised instruments. A common problem with dividing the octave into 12 notes (used in most Western music), is the inconsistent ratios between notes when a different root key is chosen. This problem is the result of using the natural harmonic series to define the ratios between pitches, i.e, a perfect fifth is the ratio 3:2, which was used to define the Pythagorean tuning [30].

There is no way to tune 12 notes in a scale that will result in equal integer ratios for all intervals across all notes and octaves. Equal temperament tuning was introduced to solve this problem [31].

This tuning system uses $\sqrt[12]{2}$ as the relationship between semitones, resulting in equal ratios for all intervals across all octaves. The standard for tuning is defined by the frequency of concert A (A4) to be 440 Hz. Consequently, the n'th semitone after A4 is $440(\sqrt[12]{2})^n$, and the k'th semitone before A4 is $440(\sqrt[12]{2})^{-k}$. Furthermore, each semitone is divided into 100 equal steps, which is known as cents, with each cent differing from the next by a ratio of $\sqrt[1200]{2}$. This is a measure of the intonation of note. Refer to table C.2 in the appendix for the frequencies of notes in equal temperament tuning.

A frequency ratio of $c \in \mathbb{R}$ cents, can be split into a combination of semitones ($x \in \mathbb{Z}$) and cents ($y \in [0, 100)$) as shown in equation 2.1.

$$2^{\frac{c}{1200}} = 2^{\frac{x}{12} + \frac{y}{1200}} = 2^{\frac{x}{12}} \cdot 2^{\frac{y}{1200}} \qquad (2.1)$$

Where $x = \lfloor \frac{c}{12} \rfloor$ and $y = c - 12x$. Using this type of decomposition for a frequency ratio, allows for easy table look-ups into semitone and cent LUTs to ensure accuracy for frequency scaling on the cent level, i.e. when $y \in \{0, 1, ..., 99\}$. Linear interpolation can then approximate these ratios between integer cent values. See code listing D.1 in the appendix for the C implementation.

2.5.2. Audio

Stereo

Most consumer audio is in a stereo format, i.e., having a left and right audio channel. It can then further be converted to mid (M) and side (S) channels using equations 2.2 and 2.3 [32]. The mid channel can be considered as the "mono'd" version of stereo audio, which is used by devices such as phone speakers that cannot play stereo

audio. The side channel can be considered as the signal containing all the stereo information, i.e., a measure of audio width. Audio processing is sometimes done on the mid and side channels instead of the left and right channels. After processing, the left and right channels are reconstructed.

$$M = \frac{L+R}{2} \tag{2.2}$$

$$S = \frac{L-R}{2} \tag{2.3}$$

Note that if $L = R = y[n]$ (a mono signal played through both stereo channels), then $S = 0$, $M = y[n]$, implying that no stereo information is present. Using equations 2.2 and 2.3, the L and R signals can be reconstructed using:

$$L = M + S \tag{2.4}$$

$$R = M - S \tag{2.5}$$

Quality

A variety of digital properties can determine the quality of the audio that is streamed. Prime considerations are sampling rate and bit depth. Other considerations include dithering and encoding (PCM, DPCM, etc.) [33].

The sampling rate determines the bandwidth of the audio, as per the Nyquist criterion [34]. Since human hearing is restricted to 20 Hz to 20 kHz [29], sampling rates for high-fidelity audio often exceed 40 kHz. Common rates are 44.1, 48 and 88.2 kHz [6]. The sampling rate is often higher than required, to avoid aliasing when processing the signals. The signal is sometimes up-sampled (often through linear interpolation) to double or quadruple the sampling rate before processing in order to combat aliasing.

The bit depth refers to the amount of bits used to store the audio data. The way in which it is stored depends on the encoding, which will yield different quantisation error probability distributions. The bit depth determines the noise-floor of the signal, expressed via SQNR. Typical bit-depths are 16, 24 and 32 bits, with SQNRs of 96.33, 144.49 and 192.66 dB, respectively [35].

2.5.3. Prototype IIR filters

A popular technique for designing an IIR filter is by using a continuous prototype filter $H(s)$ with critical frequencies $\omega_c = 1$ rad/sec, by utilising the bilinear transform and fequency scaling [2].

Prototype filters are converted to the Z-domain to be in the form

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}} = \frac{N_0 + N_1 z^{-1} + N_2 z^{-2}}{1 - D_1 z^{-1} - D_2 z^{-2}} \tag{2.6}$$

where N_0, N_1, N_2, D_1, D_2 are the coefficients normalised by a_0. D_1 and D_2 are negated.

Converting back to the discrete time domain, the output of the filter ($y[n]$) can be calculated as

$$y[n] = N_0 x[n] + N_1 x[n-1] + N_2 x[n-2] + D_1 y[n-1] + D_2 y[n-2] \tag{2.7}$$

The negation of D_1 and D_2 proves useful in this circumstance, since multiply-and-accumulate FPU instructions can be taken advantage of.

A list of substitutions to convert from the S to Z domain, using the bilinear transform, is shown in table C.1 in the appendix. The table also takes frequency mapping into account. The cut-off frequencies are mapped from 1 rad/s in the prototype filter to $\omega_0 = 2\pi \frac{f_0}{f_s}$ rad/sample (normalised digital frequency).

Chapter 3

Design

In this chapter, bottom-up synthesis is applied, by designing, modeling and deriving behaviour of the lowest-level components first. In implementing code for this section, compilation and FPU instructions are considered. This will not always be discussed. See table C.3 in the appendix for a list of ARM FPU assembly instructions.

Only short code implementations are shown in this section. If implementations are trivial or large, the C code is listed in appendix D. Enough information will be provided by the system block diagrams and accompanying equations to make other code implementations possible.

Initially, this system was partially implemented for integer arithmetic using Q-numbers, but is rewritten for floating-point operations after problems were encountered, and extra design and implementation considerations were taken into account. See appendix B.3 for details.

All code is publicly available at https://github.com/marcorad/skripsie.

3.1. DSP pipeline

The system consists of a number of functional components which are detailed in section 1.3.2. Figure 3.1 shows a high-level description of all of the components and the processes that are performed. The components are numbered according to a top-down design approach, where number 1 refers to the highest level of operation between components. Components are only dependent on other lower-level components.

3.1.1. Component definition

Each component can be defined in terms of its lower-level component dependencies, the required functionality, and its inputs/outputs. The following list describes the components, with reference to the numbering on figure3.1:

1. **Generator manager**: manages the generators by efficiently choosing an available generator, or the oldest playing generator if none are available. It receives note on/off requests, associates it with a generator, and does all triggering and configuration (such as frequency) that is required. The active generators are sampled for an audio buffer request. Generator samples are summed, scaled and clamped to a range [-1, 1]. This component uses a MIDI-note hashtable, and a generator queue and stack, to manage note associations and active/inactive generators.

2. **Generator**: represents the collection of data that is required to produce stereo audio samples for a single triggered note. It is responsible for retrieving and blending wavetable samples; applying waveshaping (and the anti-aliasing required); sampling the ADSR envelopes for volume and filter modulation; triggering the recalculation of filter coefficients and applying the filtering.

3. **Wavetable**: is responsible for all LUT lookups and inter-LUT interpolation, while also managing the periodicity (as per subsection 3.3), frequency and harmonic content of a periodic waveform. It can receive FM input for vibrato. It produces mono samples that are to be used in conjunction with other wavetables in stereo blending.

4. **ADSR**: produces mono samples from an ADSR curve, using table lookups into an exponential function LUT. It is a state-machine that manages the piece-wise exponential ADSR curve, given ADSR parameters.
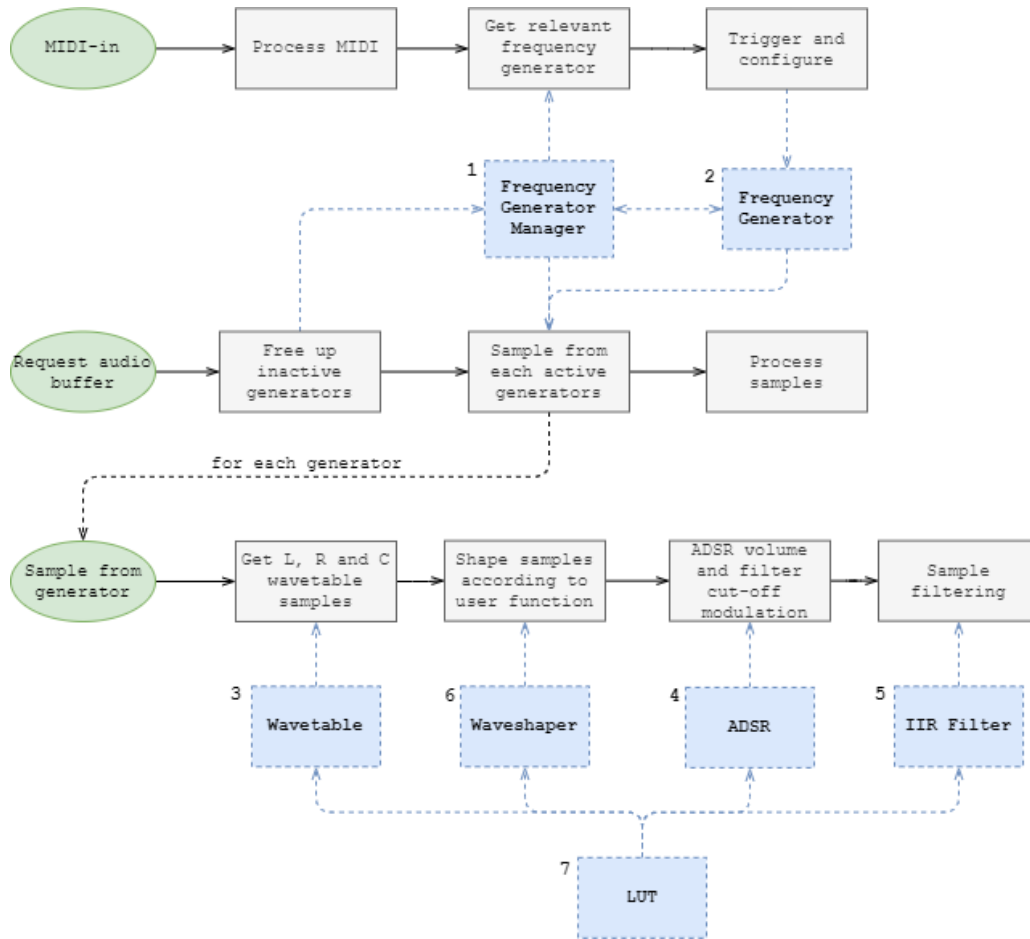
Figure 3.1: High-level system description

It can be queried to determine whether the release phase has been completed. It can receive on and off triggers, and can be re-triggered to the attack phase, at any moment, if necessary.

5. **IIR filter**: contains all the functions from different filter types that can calculate the IIR coefficients. It stores all necessary sample delays for each filter instance, that are required to perform filtering. It applies filtering to an input signal, and has the ability to be reset (sample delays set to 0).

6. **Waveshaper**: applies a waveshaping function (either sin, tanh, or none) to an input. It does not handle anti-aliasing (the generator does). It is responsible for doing lookups into the function LUTs and managing periodicity with the sin lookup, and out-of-bounds tanh lookups.

7. **LUT**: preforms a singular table lookup, with linear interpolation, into any array, without checking for an out-of-bounds index - ensuring a within-bounds lookup index is the responsibility of parent components. It is the core of the audio engine.

3.1.2. Global user parameters

From the functional specification in section 1.3.2, the global parameters that are applicable to every playing note are determined. These parameters are separated from the data contained within the components, so that it can easily be changed during an audio stream, to provide the performer with continuous feedback, and avoid data duplication.

Table 3.1 summarises the global parameters; indicating to which category of processing they are relevant, and the units they are stored in. Note that the attack, decay and release parameters are stored in terms of digital

frequency, and not seconds. This is so that LUT phases can easily be updated (see section 3.3. All frequencies in this system are stored as digital frequency, so as to remain sampling frequency agnostic.

Table 3.1: All global parameters

Category	Parameter	Unit(s)
Volume	Envelope: attack, decay, release	cycles/sample
	Envelope: sustain	-
Filtering	Envelope: attack, decay, release	cycles/sample
	Envelope: sustain	-
	Relative frequency start	-
	Relative frequency end	-
	Q	-
	Type	-
Detune	Amount	cents
	Stereo width	-
	Volume	-
Vibrato	Frequency	cycles/sample
	Intensity	cents
Waveshaping	Input gain	-
	Type	-

3.2. Top-level system in MCU implementation

The top-level system is the entry-point of MIDI messages and is illustrated in figure 3.3. In this part of the system, MIDI commands are parsed and added to a queue. An audio buffer request is made, where a fixed amount of stereo samples are retrieved from the generators and stored in a buffer. This is the system that is to be implemented on a MCU, and is thus **not within the scope** . Instead, a SMF provides note input (see appendix B.1) to emulate MCU functionality. The MIDI data is converted to audio and stored in a wav file. Refer to appendix B.1 for more information.

Figure 3.2 shows the legend that should be used to interpret the block diagrams shown in this chapter.

The note queue is only processed at the beginning of a buffer request, thus adding latency into the system. The latency (l) is a function of the size of the audio buffer (N), specified in number of stereo samples and the sampling rate (f_s).

$$l = \frac{N}{f_s} \tag{3.1}$$

The added latency must be minimal to avoid affecting live performance. Typically, latencies of 10 ms to 15 ms are acceptable and unnoticeable to performers [36]. Thus, at a sampling rate of 44.1 kHz, a maximum size of 661 stereo samples are acceptable for the buffer.

A note on/off-trigger is restricted to a minimum time equal to the latency, otherwise a note on/off trigger pair will cancel out, since they are both processed at the same time. Furthermore, extra equipment in the signal processing chain, such as external effect and amplifiers, may add more latency. It is better choose an audio buffer size of around 2 ms, so that the instrument can still be real-time-usable in conjunction with an external signal chain.

The choice of buffer size is not relevant in this thesis, with the exception of note triggering time, since system testing is not done for a real-time application. The system is designed to be compatible for any choice of buffer size and sampling rate. Selecting these must be done with the hardware platform in mind. Generally, a higher
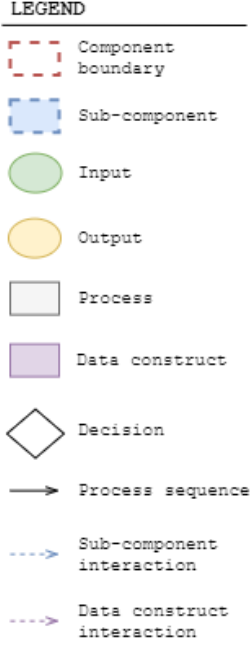
Figure 3.2: System block diagram legend.
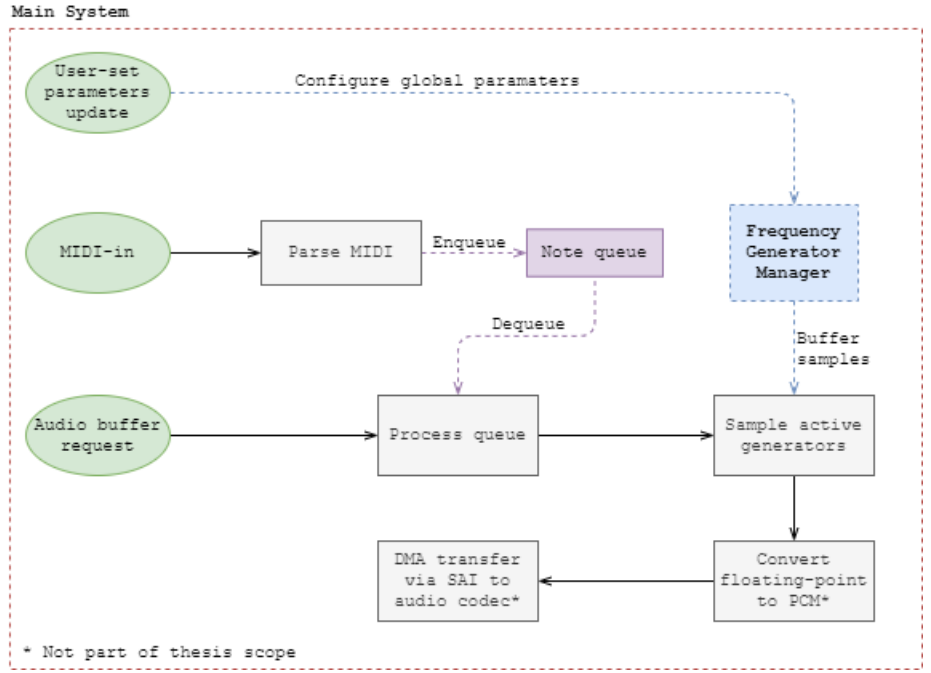
Figure 3.3: The top-level system block diagram

sampling rate will reduce aliasing and latency, which should be leveraged if the hardware has enough processing power.

3.3. LUT

3.3.1. Linear interpolation

Linear interpolation is the process of defining a function in terms of points, and then connecting the points in a piecewise-linear fashion [37]. This is especially relevant for LUT lookups, when a value between points is required for better accuracy. This is the core of the system. Figure 3.4 shows the block diagram for this system component.
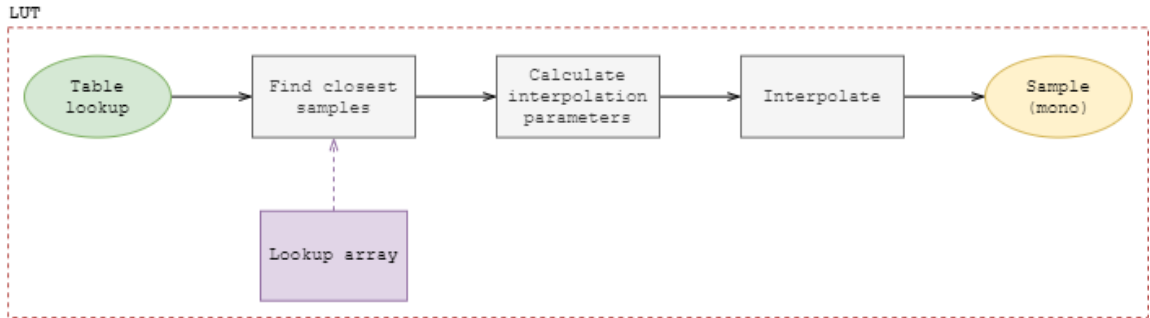
Figure 3.4: LUT system block diagram

The linear interpolation (lerp) function is defined as follows [37]:

$$\mathrm{lerp}(x_1, x_2, \delta) = x_1 + \delta(x_2 - x_1) \tag{3.2}$$

where x_1 is the starting point, x_2 is the endpoint, and $\delta \in [0, 1]$ is the interpolation distance.

Assuming a LUT ($L[n]$) is storing N samples indexed with $n < N$, $n \in \mathbb{N}_0$, we can find a linearly interpolated

point $p(i)$ at position $0 \le i < N - 1$, $i \in \mathbb{R}$ using equation 3.2.

$$p(i) = \text{lerp}(L[\lfloor i \rfloor], L[\lfloor i \rfloor + 1], \{i\}) \tag{3.3}$$

If periodic behaviour is required from the LUT, i is wrapped around when it exceeds the limits, either through using the fractional function or the modulo function for real numbers.

The modulo operation can be defined through floored division as shown in equation 3.4, which can also provide an extension of the function into the real numbers for $a, r, n \in \mathbb{R}$.

$$a \equiv r \pmod{n} \Leftrightarrow r = a - n\lfloor \frac{a}{n} \rfloor \tag{3.4}$$

Equation 3.4 is implemented efficiently in code if n is a power of 2, and $a \in \mathbb{N}_0$. Listing 3.1 shows the C implementation for this function [38]. Note that the use of the "inline" keyword is a C++ compiler directive, but is only used to increase the speed of the program by avoiding branch penalties when calling the function. All periodic LUTs are thus restricted to have size that is a power of 2.

```
inline uint16_t fast_mod(uint16_t x, uint16_t mod) {
    return x & (mod - 1);
}
```

Listing 3.1: Fast modulo for a power of 2

Listing 3.2 shows the implementation of equation 3.3 and figure 3.4 in C, where special care is taken to ensure that if $i \in [N - 1, N)$, the interpolation is performed between the samples $L[N - 1]$ and $L[0]$. This is required for interpolating periodic LUTs.

```
inline float lut_lookup(float lut[], uint32_t lut_size, float i) {
    uint32_t floor_i = (uint32_t) i;
    float delta = i - (float)floor_i;
    float x1 = lut[floor_i];
    float x2 = lut[fast_mod(floor_i + 1, lut_size)]; //wraps i to 0 if i = lut_size-1
    return lerp(x1, x2, delta);
}
```

Listing 3.2: LUT lookup with linear interpolation

The fractional function can also be used to ensure periodicity. Equation 3.5 shows the definition of the fractional function and its range.

$$\{x\} = x - \lfloor x \rfloor \ , \ \{x\} \in [0, 1) \ \forall \, x \in \mathbb{R} \tag{3.5}$$

The fractional function is periodic with a frequency of 1 Hz, and contains a straight line starting at (0,0) and approaching (1,1) over $x \in [0, 1)$. Thus, a function $(W(t))$ is created that wraps the index so it repeats with a period of p, with a shifting factor (k), and an amplitude (A), shown by equation 3.6. This is used to easily wrap the index into a LUT of size A, with p defining the input range of t, and k setting the starting lookup index $(W(t = 0))$.

$$W(t) = A\{\frac{1}{p}(t - k)\} \tag{3.6}$$

3.3.2. Constructing the basic waveform LUTs

In this section, the method of generating a Fourier series of the basic waveforms (square, triangle and sawtooth) for LUT look-ups and explore memory optimisation techniques is investigated. The waveform of interest is

represented as the following series:

$$y[n] = \sum_{k=1}^{M} a_k \sin\left(2\pi k \frac{n}{N}\right) \tag{3.7}$$

where k is the wavenumber, a_k is the k'th amplitude coefficient, M is the number of harmonics, and $N = 2^b$ is the size of the LUT that will store a complete period of the waveform, i.e. $0 \le n < N$, $n \in \mathbb{N}_0$.

With reference to equation 3.7, the coefficients a_k are derived as follows [34]:

1. Square:

$$a_k = \begin{cases} \frac{1}{k}, & k \equiv 1 \pmod 2 \\ 0, & k \equiv 0 \pmod 2 \end{cases} \tag{3.8}$$

2. Triangle:

$$a_k = \begin{cases} \frac{(-1)^{\frac{k-1}{2}}}{k^2}, & k \equiv 1 \pmod 2 \\ 0, & k \equiv 0 \pmod 2 \end{cases} \tag{3.9}$$

3. Sawtooth:

$$a_k = \frac{(-1)^{k+1}}{k} \tag{3.10}$$

Equations 3.8 and 3.9 imply only odd harmonics, hence for harmonic index (see equation 3.14) $i = 0$, it will represent a simple sinusoid.

Furthermore, since equation 3.7 is a linear combination of harmonic sinusoids with no phase-shift, and $a_k > 0$ for all odd k, $a_k \le 0$ for all even k in equations 3.8 to 3.10, it is concluded that all basic waveforms and their harmonics are in phase. This makes it possible to interpolate between wavetables (as seen in Ableton's Wavetable VST in figure A.2) without phase cancellation. Furthermore, to allow for range predictability, the samples are normalised so that the range falls within $[-1, 1]$.

The sinusoid is trivial to construct, since it does not contain any other harmonics. Thus, the 4 basic waveforms can be stored in a $4 \times K \times N$ array. "4" corresponds to the basic waveforms, ordered by increasing harmonic content (sine, triangle, sawtooth, square); K refers to the amount of copies with different harmonics we want to store of each waveform and $N = 2^b$, $b \in \mathbb{N}$ is the amount of samples per waveform period. The 3D array allows for code simplicity when each note requires an harmonic index and inter-wavetable interpolation (see section 3.4).

In this case, the sinusoid LUT is duplicated K times. However, speed and code simplicity are prioritised over memory consumption . Adding code to deal with the sinusoid specifically could incur branch penalties, and requires a special array for the sinusoid, which increases code complexity. Refer to code listing D.2 in the appendix for the C implementation.

3.4. Wavetable

This component is responsible for generating a specific frequency of a waveform using the basic waveform LUTs. Figure 3.5 shows the block diagram.

Each wavetable requires its own data structure to store relevant values, which include stride and base stride (stride $\propto \eta$, see equation 3.13); phase (ϕ) and harmonic index (i). Base stride refers to the center frequency of the wavetable, whereas stride includes FM.

On a sample request, the appropriate wavetable lookup and inter-wavetable interpolation is performed. The
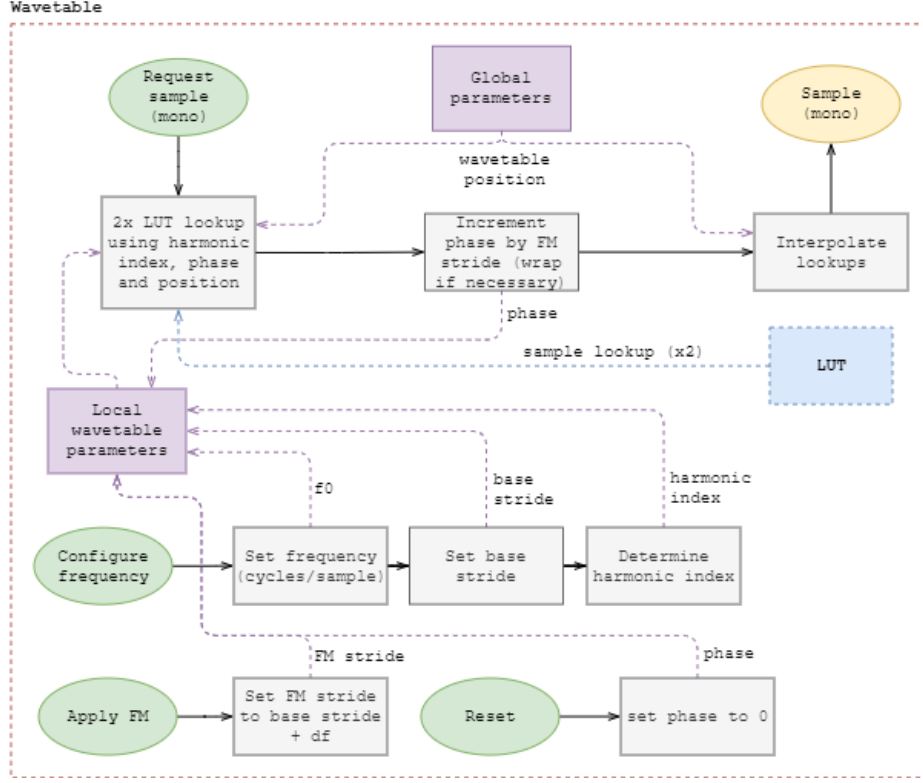
Figure 3.5: Wavetable system block diagram

phase is incremented after each request. The frequency must also be configured before use, as it is required to calculate the harmonic index. A reset-trigger resets the phase. The rest of this section provides further explanation, derivation and insight into these parameters.

3.4.1. Modelling wavetable frequency conversion

This section focuses on modelling wavetable sampling and determining the effects of linear interpolation and frequency scaling in the frequency domain.

Suppose a periodic signal $y[n]$ is stored in a buffer consisting of 2^b samples, where $b \in \mathbb{N}$. The fundamental frequency of the buffer as normalised digital frequency (cycles per sample) is thus 2^{-b}.

If the buffer is required to store H harmonics, including the fundamental, the Nyquist frequency is used to determine the maximum number of harmonics the buffer can store. This is specified by the user of the code, according to the memory restrictions of the hardware platform.

$$H_{max}2^{-b} = 0.5 \Rightarrow H_{max} = 2^{b-1} \tag{3.11}$$

Therefore, a bigger buffer size results in the ability to store more harmonics. The process shown in figure 3.6 is considered when scaling a sampled signal's frequency ($y[n]$) by a factor of $\eta = \frac{M}{L}$. The signals in the frequency conversion process is modelled as $\hat{y}[n] = (y[n]_{\uparrow L} * h[n])_{\downarrow M}$.
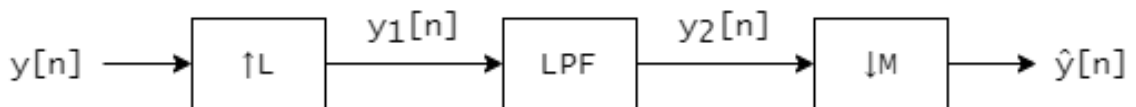
Figure 3.6: Frequency scaling using upsampling and downsampling

As per the scaling theorem [39], the frequency axis of DTFT$\{y_1[n]\}$ contracts by a factor L. A LPF is used

to remove unwanted copies of the spectrum, which would result in aliasing if the frequency axis is expanded by a factor M through downsampling.

There are many choices for the LPF, but for arbitrary frequency scaling, M and L can become large to achieve close approximation for any real number. The simplest filter choice is the linear interpolator, which is a FIR filter characterised by its impulse response [39]:

$$h[n] = \text{tri}[\frac{n}{L}] = \frac{1}{L}\text{rect}[\frac{n}{L}] * \text{rect}[\frac{n}{L}]$$

Taking the DTFT,

$$H(f) = \frac{1}{L}\text{DTFT}\{\text{rect}[\frac{n}{L}]\}^2 = \frac{\sin^2(L\pi f)}{L\sin^2(\pi f)}$$

Note that $H(f) = 1$ for $L = 1$. For $L \neq 1$, it has zeroes at $f = \frac{p}{L}, p \in \mathbb{N} \setminus \{0\}$. Figure 3.7 shows the effect of the linear interpolation process in the frequency domain. The bandwidth of $y[n]$ is represented as a triangular pulse.
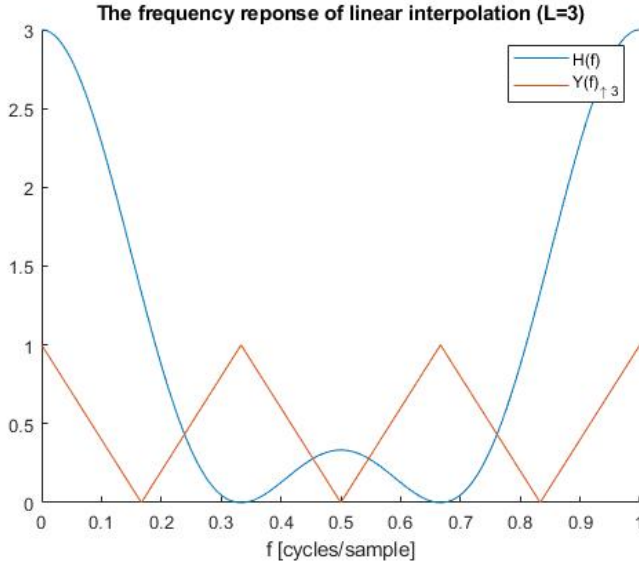
Figure 3.7: The effects of linear interpolation in the frequency domain

With reference to figure 3.7, $H(f)$ has nulls at the DC component of the upsampled signal spectrum. All the frequencies that are close to DC, i.e., lower frequencies, have high attenuation. Thus, when the sampling rate is increased, the spectral distortion resulting from downsampling, is unaffected, since the wavetable fundamental is fixed at 2^{-b} cycles/sample. Increasing b, however, will lower the fundamental frequency and result in less spectral distortion when interpolating and downsampling lower frequencies.

The linear interpolation filter has a fixed cutoff, which is a function of L. When the frequency is scaled up, i.e. $M > L$, the original spectrum with a full bandwidth will alias, since the linear interpolation filter does not account for this. To combat this issue, a variety of buffers for a single waveform is used, each containing a different number of harmonics, thus band-limiting the signal to combat aliasing for when $M > L$. This problem is not present for $M < L$, i.e., playing frequencies lower than 2^{-b} cycles/sample. Below this threshold, the maximum harmonics played back is limited by b.

Assuming that the waveform is bandlimited to B cycles/sample, it is to be ensured that

$$\frac{M}{L}B \leq 0.5$$

as per the Nyquist criterion. Furthermore, B is determined by the number of harmonics h, i.e. $B = h2^{-b}$.

$$h2^{-b} \leq \frac{L}{M}0.5 \Leftrightarrow h \leq 2^{b-1}\frac{L}{M} = 2^{b-1}\frac{1}{\eta} \tag{3.12}$$

The maximum number of harmonics (including the fundamental) h_{max} can be found for a given digital

frequency f_0. First, the required frequency scaling factor is calculated:

$$2^{-b}\eta = f_0 \Rightarrow \eta = f_0 2^b \Rightarrow \eta = N f_0 \tag{3.13}$$

Substituting equation 3.13 into equation 3.12, we find

$$h \le \frac{0.5}{f} \Rightarrow h_{max} = \frac{0.5}{f_0}$$

If K buffers are used to store a varying number of harmonics, with the buffers indexed by $i \in \{0, 1, ..., K\}$, storing 2^{i+1} harmonics in each buffer, it restricts $K_{max} = b - 2$, as per equation 3.11. Note that this is arbitrary, and only to avoid large memory consumption for storing a linear increase in harmonics. If the MCU has external memory available, it should be considered to store as many waveforms with differing harmonics as possible, using a linear increase instead.

The closest index of a given digital frequency f_0 is found, which will contain the maximum number of harmonics (h_{max}), without aliasing, stored in the buffer i as follows:

$$i = \min(\lfloor log_2(\lfloor h_{max} \rfloor) \rfloor, K) - 1 \tag{3.14}$$

Equation 3.14 is optimised by using a LUT and further memory considerations, to avoid an expensive call to a logarithmic function. See appendix B.4 for details. The C implementation for configuring a wavetable is shown in listing D.4 in the appendix.

Note that the minimum number of stored harmonics is 2 ($i = 0$). Thus, only frequencies up to a max of 0.25 cycles/sample can be played. The highest note on an 88-key piano (C8) is 4186.01 Hz in equal temperament tuning. At a standard audio sampling rate of 44.1 kHz, this corresponds to $f_0 = 0.095 < 0.25$ cycles/sample. Almost all of the MIDI notes (see table C.2 in the appendix) are covered in this range, with the highest non-aliased note being E9 (10.54808 KHz), which corresponds to $f_0 = 0.239 < 0.25$ cycles/sample. Arguably, this is well outside the commonly played range. At 44.1 KHz, the only aliased MIDI notes are F9, F#9 and G9. This problem is completely avoided at a sampling rate of 48 kHz.

3.4.2. Implementing frequency scaling

Frequency scaling is trivially implemented without any upsampling, downsampling or explicit filtering. The frequency scaling factor η (from equation 3.13) is used to update a pointer into a LUT ($W[i]$) storing a periodic waveform.

For a table consisting of 2^b samples, the pointer is considered as the phase $\phi[n]$ of the waveform. The phase is updated using:

$$\phi[n] = \mod(\phi[n-1] + \eta, 2^b) \tag{3.15}$$

Here, the frequency scaling factor η represents the amount that the table pointer must increase by, which is known as **stride**.

The modulo operation ensures that the phase pointer never falls outside of the range of indices of the table, effectively creating the periodicity required. However, $f_0 > 0.5$ is not expected, thus equation 3.15 simplifies to:

$$\phi[n] = \begin{cases} \phi[n-1] + \eta, & \phi[n-1] + \eta < 2^b \\ \phi[n-1] + \eta - 2^b, & \phi[n-1] + \eta > 2^b \end{cases} \tag{3.16}$$

which eliminates using an expensive floating-point modulo operation.

With $\phi[n] \in [0, 2^b)$ due to the modulo operation, the samples of $W[i]$ are linearly interpolated to $L[n]$ using equation 3.3:

$$L[n] = \text{lerp}(W[\lfloor \phi[n] \rfloor], W[\text{mod}(\lfloor \phi[n] \rfloor + 1, 2^b)], \{\phi[n]\}) \tag{3.17}$$

Equation 3.3 is easy to implement in code, since the floor of a positive floating-point number can be determined through integer conversion. The fractional part of a floating-point is determined using 3.5.

Using an if-statement to detect wrapping to the beginning of the table is also an alternative, but not necessarily as efficient, depending on branch penalties in the processor.

3.4.3. Implementing inter-wavetable interpolation

From subsection 3.3.2, the wavetables are stored in a 3D array, notated as $W[t, k, n]$, where $t \in \{0, 1, 2, 3\}$ is the wave type, k is the harmonic index and n the sample index.

Interpolation between 2 wavetables is specified by $\hat{t} \in [0, 4)$, which is the continous extension of t. Interpolation is done similar to equation 3.17. Equation 3.18 shows this, and allows wrapping to occur from the square to the sinusoid LUTs. Listing 3.3 shows the C implementation for wavetable sampling.

$$W(\hat{t}, k, n) = \text{lerp}(W[\lfloor \hat{t} \rfloor, k, n], W[\text{mod}(\lfloor \hat{t} \rfloor + 1, 4), k, n], \{\hat{t}\}) \tag{3.18}$$

```
inline float wt_sample(wavetable* wt, gen_config* gc) {
  uint8_t t = (uint8_t)gc->wt_pos;
  uint8_t tp1 = t + 1;
  np1 = fast_mod(np1, 4); //wrap LUT (square to sine)
  float x1 = lut_lookup(basic_luts[t][wt->harmonic_index], LUT_SIZE, wt->phase);
  float x2 = lut_lookup(basic_luts[tp1][wt->harmonic_index], LUT_SIZE, wt->phase);
  wt->phase += wt->stride;
  if (wt->phase > (float)LUT_SIZE) wt->phase -= (float)LUT_SIZE; //wrap phase
  return lerp(x1, x2, gc->wt_pos - (float)t);
}
```

Listing 3.3: Sampling from a wavetable

3.4.4. Applying FM

FM is applied by modifying the base stride for each sample. Similar to conventional FM techniques, there is a center/base frequency f_0, which is modified by a modulation frequency $f_{FM}[n], as per equation 3.13.

Thus, the base stride (η_0) is stored as part of the internal wavetable state, which is set during a note-on trigger. The final stride is expressed as $\eta_f[n] = \eta_0 + \eta_{FM}[n]$.

3.5. IIR filtering

As detailed in section 2.3, filters are an essential component in many synthesisers. Filter cut-off can be controlled by external modulation sources. Thus, fast filter coefficient calculation is paramount, since it is calculated on a per-sample basis.

To achieve maximal speed, division operations is avoided as far as possible, since most processors do not have single-cycle division operations. Furthermore, a biquad IIR filter is used for speed. This also reflects the typical 2-pole filters often used in analogue synthesisers. Single-pole filters are also analysed, since they are often used for a "softer" roll-off and are required for anti-aliasing techniques in waveshaping, explored in section 3.6.

The 3 most common 2-pole filter types (HP24, BP12, LP24) along with 1-pole filters (HP12, LP12) for synthesis, is detailed.

The techniques mentioned in section 2.5.3 is used. Equations 3.19 to 3.23 are the prototype HP24, LP24, BP12, HP12 and LP12 filters respectively, obtained from [40].

$$H_{hp24}(s) = \frac{s^2}{s^2 + \frac{1}{Q}s + 1} \tag{3.19}$$

$$H_{lp24}(s) = \frac{1}{s^2 + \frac{1}{Q}s + 1} \tag{3.20}$$

$$H_{bp12}(s) = \frac{s}{s^2 + \frac{1}{Q}s + 1} \tag{3.21}$$

$$H_{hp12}(s) = \frac{s}{s + 1} \tag{3.22}$$

$$H_{lp12}(s) = \frac{1}{s + 1} \tag{3.23}$$

The coefficients of the digital biquad filter are calculated for the prototype filters. The denominator coefficients are the same for the HP24, LP24 and BP12 filters, whereas the denominator coefficients of the HP12 and LP12 are the same. It is summarised in table 3.2 below.

Table 3.2: Digital biquad filter denominator coefficients

Filter	a_0	a_1	a_2
HP24, LP24, BP12	$2Q + \sin(\omega_0)$	$-2 \cdot 2Q\cos(\omega_0)$	$2Q - \sin(\omega_0)$
HP12, LP12	$(1 - \cos(\omega_0)) + \sin(\omega_0)$	$2(1 - \cos(\omega_0))$	$(1 - \cos(\omega_0)) - \sin(\omega_0)$

The numerator coefficients are shown in table 3.3 below.

Table 3.3: Digital biquad filter numerator coefficients

Filter	b_0	b_1	b_2
HP24	$-\frac{1}{2}b_1$	$-2Q(1 + \cos(\omega_0))$	$-\frac{1}{2}b_1$
LP24	$\frac{1}{2}b_1$	$2Q(1 - \cos(\omega_0))$	$\frac{1}{2}b_1$
BP12	$Q\sin(\omega_0)$	0	$-b_0$
HP12	$-\sin(\omega_0)$	0	$-b_0$
LP12	$\frac{1}{2}b_1$	$2(1 - \cos(\omega_0))$	$\frac{1}{2}b_1$

Calculating the coefficients is optimised by using a LUT for cosine and sine approximation, and stores $2Q$, $\cos(\omega_0)$, $\sin(\omega_0)$, and $1 - \cos(\omega_0)$ as intermediate values. Unfortunately, a single division operation is unavoidable. All coefficients are normalised by a_0 to obtain the filter form of equation 2.6. This scaling factor is also stored as an intermediate value and then further used via multiplication.

Listing D.5 in the appendix shows the C implementation for calculating the filter coefficients using tables 3.2 and 3.3. Implementation for all filters are similar, utilising pre-negation of a_1 and a_2 to save on MCU clock cycles, storing intermediate values and utilising trigonometric LUTs, which are gained without extra memory consumption. A sinusoid is already stored as a basic waveform. The cosine LUT is also gained without extra memory consumption, by using the identity $\cos(x) = \sin(x + \frac{\pi}{2})$. The trigonometric lookup functions' C implementations are shown in appendix listing D.7.

Filtering is done using equation 2.7. The C implementation is shown in code listing D.6 in the appendix.

3.6. Waveshaper

As per the functional requirements, waveshaping is done by one of 2 user-chosen functions: the hyperbolic tangent or the sinusoid. LUTs are used to perform the waveshaping. Refer to code listing D.3 in the appendix for the waveshaper LUT functions.

The sinusoidal LUT is already present in the basic waveforms, so no extra memory consumption is required. The hyperbolic tangent function will require its own LUT.

Shaping a discrete input function $x[n]$ to $y[n]$ with a continous waveshaping function $w(x)$ is described with $y[n] = w(g \cdot x[n])$, where g is the gain into the waveshaping function. If w is an odd function, it will only add odd harmonics, whereas an even function only adds even harmonics [41]. Both of the considered waveshaping functions are odd, so only odd harmonics are added. The waveshaping function is considered as a means to add signal distortion to the system.

Adding extra harmonics introduces aliasing into the system, since the waveshaping functions add harmonics with amplitudes proportional to the input gain g. Aliasing can be reduced in a variety of ways, such as bandlimiting the signal into the waveshaping function, or by oversampling the input signal, or both. Even though oversampling is viable, it will require more processing. Ergo, for simplicity and speed, only the bandlimiting option is considered. The system block diagram for this component is shown in figure 3.5.
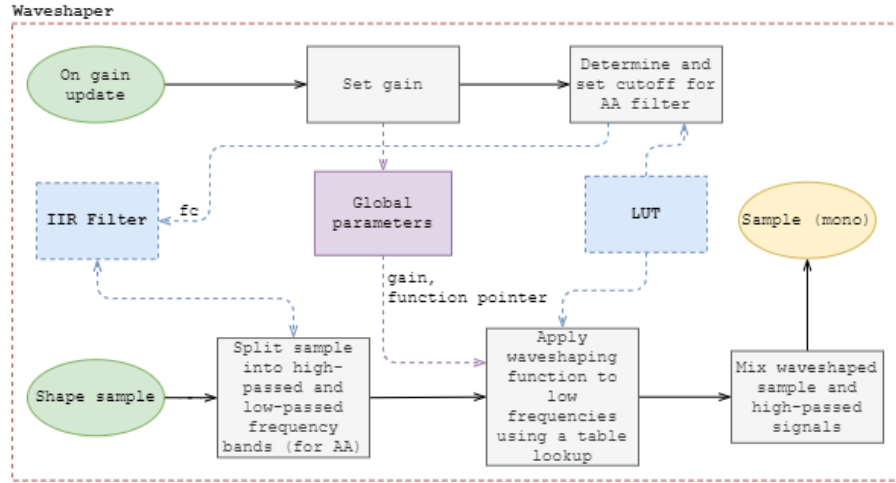
Figure 3.8: Waveshaper system block diagram

3.6.1. Constructing the hyperbolic tangent LUT

To construct a buffer with $N+1$ samples indexed by $n \in \{0, 1, .., N\}$ storing the hyperbolic tangent, some of the properties of $\tanh(x)$ are considered:

$$\lim_{x \to +\infty} \tanh(x) = 1 \quad \text{and} \quad \lim_{x \to -\infty} \tanh(x) = -1 \tag{3.24}$$

$$\frac{d}{dx} \tanh(x) = \text{sech}^2(x) \coth(x) \tag{3.25}$$

$$\lim_{x \to \pm\infty} \frac{d}{dx} \tanh(x) = 0 \tag{3.26}$$

The behaviour of this function is approximated with $y[n]$, by storing a section of $\tanh(x)$ mapped to $x = \frac{n}{N} \in [0, 1)$, The asymptotes from equation 3.24 is approximated by straight lines for $x > 1$ and $x < 0$. Shifting

and scaling the function is required, achieved by A (amplitude scaling) and k (input axis scaling). The function is shifted right by $\frac{1}{2}$, to fit most of the function behaviour in [0,1).

$$y[n] = A \tanh(k(\frac{n}{N} - \frac{1}{2}))\tag{3.27}$$

Note that the gradient $y'[\frac{N}{2}] = k$. Thus, k is also a measure of the gradient steepness at $y[\frac{N}{2}] = 0. y[N] = 1$ and $y'[N] = \varepsilon$ is desired, where ε is the acceptable gradient error close to 0, from equations 3.25 and 3.26. Solving these boundary conditions yield $A = \coth(\frac{1}{2}k)$, with values for k (as a function of ε) shown in table 3.4.

Table 3.4: Axis scaling values and gradient errors for constructing the hyperbolic tangent LUT

ε	**k**
0.1	5.3697
0.01	8.0810
0.001	10.6606
0.0001	13.1750

A higher k corresponds to better accuracy at the edges, but less accuracy within the curve around the x-intercept, since fewer sample points are available there for linear interpolation, due to axis scaling. To balance these factors, a value of $k = 9$ and $N = 256$ is chosen. Figure A.4 in the appendix shows the LUT resulting from these design choices.

3.6.2. LUT indexing

Hyperbolic tangent

To index into the hyperbolic tangent LUT, the transformations applied in equation 3.27 are reversed. Thus, our lookup index n for an input value t is shown by equation 3.28. The final output is not scaled by $\frac{1}{A}$, since $A \approx 1$ for small ε. For $x < 0$, -1 is outputted and for $x > 1$, +1 is outputted.

$$n = \frac{N}{k}(x + \frac{1}{2}) \, , \, x \in (0,1)\tag{3.28}$$

Sinusoid

A LUT storing $\sin(2\pi \frac{n}{N})$ is already present in the basic waveforms. Since finding $\sin(x)$ is desired, n is found utilising equation 3.6, with $p = 2\pi$ and $A = N$ to ensure in-bounds indices and periodic behaviour.

$$n = N\{\frac{t}{2\pi}\}\tag{3.29}$$

3.6.3. Analysing waveshaping frequency content

Analytical analysis of the Fourier series of $\tanh(g \cdot sin(x))$ is possible by using the Taylor series expansion of $\tanh(x)$. However, the Taylor series only converges for $x \in (-\frac{\pi}{2}, +\frac{\pi}{2})$, which proves to be problematic for any input $|g \cdot x[n]| \geq \frac{\pi}{2}$. Furthermore, the Fourier series of $\sin(g \cdot sin(x))$ can be evaluated in terms of the Bessel functions [42], which would require a LUT anyways, since these functions have an infinite series representation.

For any unknown input $g \cdot f(x)$ into the waveshaping function, finding an analytical solution of the Fourier series is incredibly cumbersome, or impossible. Instead, a numerical approach is taken to determine the effects of waveshaping in the frequency domain, where the effect of g is investigated on a sinusoid. A unique technique is devised.

For any waveshaping function $w(x)$ and a set of M increasing gain values $\{g_1, g_2, ..., g_M\}$, the effect of g in the discrete time domain is investigated, by sampling a 1 Hz sinusoid with N points within a period, while recording P periods.

$$Y[k, g] = \text{DTFT}\{w(g \cdot \sin[2\pi\frac{n}{N}])\} \tag{3.30}$$

For each $g \in \{g_1, g_2, ..., g_M\}$, the number of the harmonic that is last in exceeding a predefined amplitude ratio with the fundamental is recorded. In this case, the ratio is chosen as $\frac{1}{100}$ (-40 dB) to the fundamental.

N should be large enough, so that significant aliasing in analysis, which could contaminate the results, is avoided. P should be large enough for good frequency domain resolution. Choosing $N = 1000$ (500 times more than the Nyquist limit) and $P = 100$ proves to be adequate for the set of gains that are analysed. From this choice, the fundamental occurs at $k = 100$, with harmonics occurring at $k = 100 \cdot h$, $h \in \mathbb{N}/\{1\}$. Up to 500 harmonics may be analysed. The hyperbolic tangent has $g \in \{1, 2, ..., 64\}$ analysed. The sinusoid has gains analysed for $g \in \{\frac{\pi}{16}, \frac{2\pi}{16}, \frac{3\pi}{16}, ..., 4\pi\}$. This limits the gains that the user can apply to 64 and 4π, respectively.

Knowing the last harmonic h_f above the -40 dB to the fundamental threshold, the bandwidth B_w can be determined in cycles per sample, using the Nyquist criterion, shown in equation 3.31.

$$B_w = \frac{0.5}{h_f} \tag{3.31}$$

Figure 3.9 plots h_f^{-1} as a function of g for both waveshaping functions. These values are used in a LUT to determine the cutoff frequency for a bandwidth-limiting LPF. For $g = 0$ and $h_f^{-1} = \infty$ (seen in figure 3.9a for $g \in \{\frac{\pi}{16}, \frac{2\pi}{16}\}$), $h_f^{-1} = 2\frac{20000}{f_s}$ is set, which limits the signal to the audible range, as per equation 3.31.

(a) Sinusoidal waveshaping harmonic analysis

(b) Hyperbolic tangent waveshaping harmonic analysis
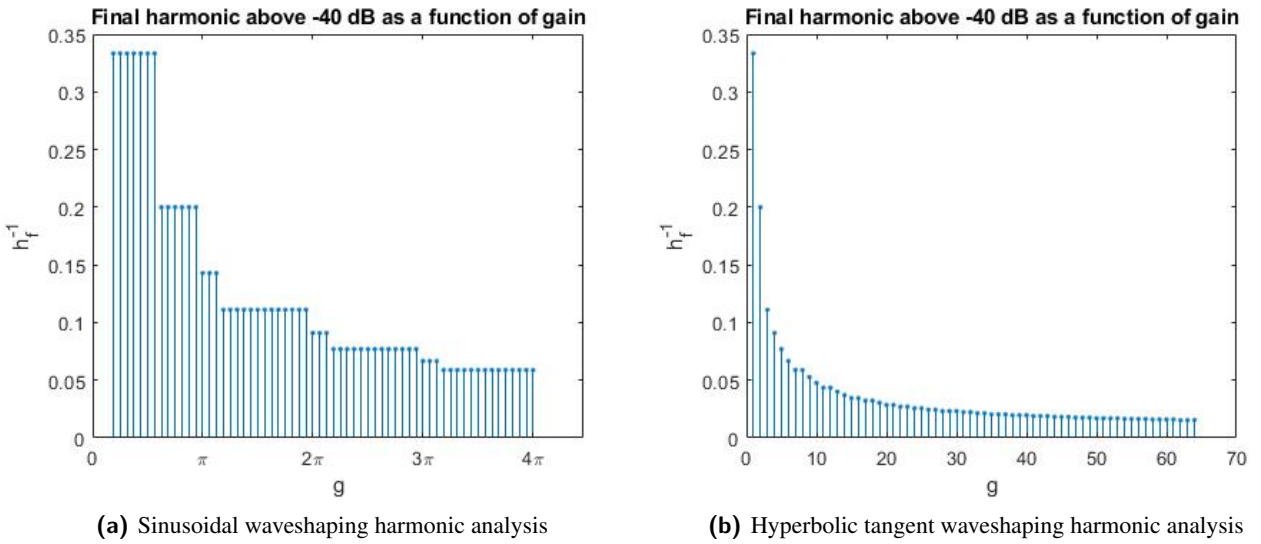
Figure 3.9: Waveshaping harmonic analyses

3.6.4. Anti-aliasing filters

From the previous section, the bandwidth into the waveshaping function is estimated. However, the frequency content above this bandwidth should not be discarded. Thus, the signal is also high-pass filtered and mixed with the band-limited waveshaped signal.

IIR filters in section 3.5 have been designed. A choice between LP12 and LP24 filters is available. Although a LP24 filter would suppress aliasing better, the LP12 filter is chosen. This is so that more harmonic content is

gained, with aliasing as a trade-off, which is often preferable in musical circumstances. This choice is made with the end-product audio in mind.

The LP12 filter also has a convenient property, that could remove the need for a corresponding HP12 filter entirely. Using equation 3.23 and 3.22, the filters are run in parallel.

$$H_{\|}(s) = H_{lp12}(s) + H_{hp12}(s) = \frac{1}{s+1} + \frac{s}{s+1} = 1 \tag{3.32}$$

From , the HP12 filter is constructed from the LP12-filtered signal, as shown in equation 3.33. This eliminates the need for a another filter, and replaces it with a subtraction operation.

$$H_{hp12}(s) = 1 - H_{lp12}(s) \tag{3.33}$$

This is expressed in the discrete-time domain as follows:

$$y_{hp12}[n] = x[n] - h_{lp12}[n] * x[n] \tag{3.34}$$

Given a waveshaping-function $w(x)$, the entire waveshaping system, with input $x[n]$ and output $y_w[n]$, is written, as shown by equation 3.35, using equations 3.33 and 3.34. The low-passed signal $x_{lp12}[n] = h_{lp12}[n] * x[n]$ is stored as an intermediary value to save clock cycles. Therefore only require a single IIR filter is required for this component.

$$y_w[n] = x[n] - h_{lp12}[n] * x[n] + w(h_{lp12}[n] * x[n]) \tag{3.35}$$

Equation 3.35 is the concrete form of what is depicted in the system block diagram from figure 3.8. Convolution is applied by the discrete-time difference filtering equation (2.7) from section 2.5.3.

3.7. ADSR envelope generator

The ADSR envelope generator can be considered a state machine that produces a piecewise-defined function of 3 exponentials and a constant, based on a trigger signal, as shown in figure A.3 in the appendix. Figure 3.10 shows the block diagram of this component. This is a state-machine with 5 states: attack; decay; sustain; release, and not playing. The attack and release states can only be externally triggered, therefore, the sampling mechanism only needs to be concerned with transitions to the decay, release and not playing states. A "playing" state is any state except for the "not playing" state.

For code simplicity, states are encoded using integers: $\{0, 1, 2, 3, 4\}$, referring to attack, decay, sustain, release and not playing states, respectively. Furthermore, the attack, decay and release times must have a minimum value to avoid clicks and pops in the audio. This value is chosen as 1 ms.

3.7.1. Creating the exponential LUT

To construct the LUT, an upwards-decaying exponential $E[n]$ is considered, with $E[0] = 0$ and $E[N-1] = 1$, where N is the required number of samples. Equation 3.36 shows the form of interest.

$$E[n] = K(1 - R^n), \ R \in [0, 1) \tag{3.36}$$

Since $\lim_{n \to \infty} E[n] = K$, a threshold value $p \in (0, 1)$ is chosen for the final sample, such that $E[N-1] = pK$. An analogue circuit would usually use a comparator with such a threshold voltage to charge and discharge a
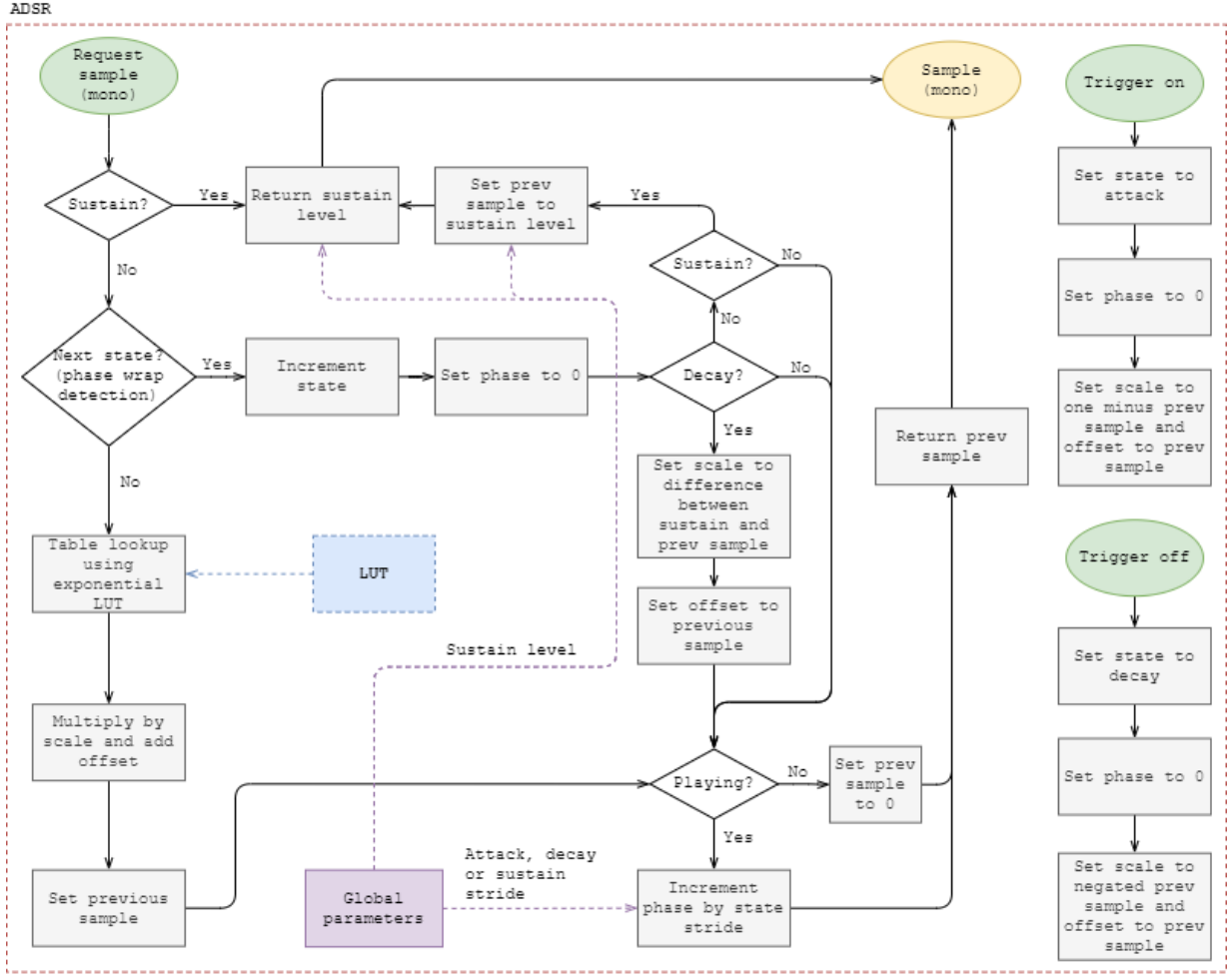
Figure 3.10: ADSR envelope system block diagram

capacitor, which also yields an exponential function. Such a threshold is arbitrary, and usually at the designer's discretion. It will determine the shape of the function stored in the LUT. A higher p will result in a flatter slope as $n \to N - 1$. A typical value to choose is $p = \frac{2}{3}$. Substituting the boundary conditions into 3.36:

$$K = \frac{1}{p}$$

$$R = \sqrt[N-1]{1-p}$$

3.7.2. Implementing the state-machine

The nature of this component requires many state checks to function properly. Since this can cause branch penalties, care is taken when considering the amount of branches and their locations, by minimising branching if possible.

Proper re-trigger behaviour is required, as per the specifications. Thus, the ADSR requires storage of its most recent sample, to allow for scaling and offset calculation, based on the previous output. It cannot be known in advance when triggering will occur, which implies that a release state trigger can occur during any state. This is similar for the attack state. A data structure is required to store all the internal state of a single ADSR envelope. The internal state includes LUT phase, exponential scaling and offset values, the previous sample, and the current state.

As per subsection 3.7.1, the last value in the LUT is 1, and the entire LUT has a range of $[0, 1]$. The previous

value ($y[n-1]$) is recorded as part of internal state. The offset (b) and the scaling factor (a) is calculated, such that the required behaviour from $E[n]$ is obtained. The following is deduced:

1. A trigger-on event must initiate the attack phase, which must rise to 1. Thus, $a = 1 - y[n-1]$ and $c = y[n-1]$.

2. A trigger-off event must initiate the release phase, which must decay to 0. Therefore, $a = -y[n-1]$ and $c = y[n-1]$.

3. The decay phase must decay to the sustain level (s). It does not necessarily transition from a sample that is exactly 1 (due to linear interpolation and phase wrapping), so thus $a = s - y[n-1]$ and $c = y[n-1]$.

To achieve the correct attack, decay and release timings, the exponential LUT is treated as a wavetable, and stride values that correspond with the required timing are used. However, periodic behaviour is not desired, thus, it is ensured that the phase is always less than $N - 1$. A transition and phase reset occurs after the phase exceeds $N - 1$. To calculate the required stride η to sweep over a single exponential ADSR state, given a time T_0 in seconds and a fixed LUT size N, an expression is derived from equation 3.13:

$$\eta = \frac{N}{T_0 f_s} \tag{3.37}$$

The C implementation of the state machine is shown in listing D.8 in the appendix.

3.8. Generator

The generator is a top-level component responsible for managing all sub-components and creating the stereo audio samples for a single note. It receives note on/off-triggers that correctly configure all sub-components with the required parameters, and subsequently trigger the ADSR envelopes. Figure 3.11 shows the system block diagram of this component. This component also requires internal state, which, to prevent clutter, is not shown in the block diagram .

The internal state of the generator, which is a function of the configured frequency, is configured with a note-on trigger. The details of this configuration is discussed further in this section.

3.8.1. Note-on/off triggers

With reference to the system block diagram (figure 3.11), the triggering process is discussed. It can never be known when a note trigger will take place, or at which frequency it takes place. Thus, wavetable frequencies for the center, left and right channels are configured using the MIDI note ID (see table C.2), which is used as a lookup index for the required digital frequency. The note frequency, referred to as **base frequency**, is stored as part of the internal state. Although not required, the phase of the wavetables are reset. A note on/off trigger also triggers the volume and filter ADSR envelope components.

3.8.2. Vibrato

Vibrato is applied using the FM capabilities of the wavetable component from section 3.4.4. From the global parameters, the vibrato intensity v is specified as a value in cents, which can be used to calculate the required frequency deviation amplitude A as a function of the base frequency f_0.

$$A = 2^{\frac{v}{1200}} f_0 - f_0 = (2^{\frac{v}{1200}} - 1) f_0 \tag{3.38}$$
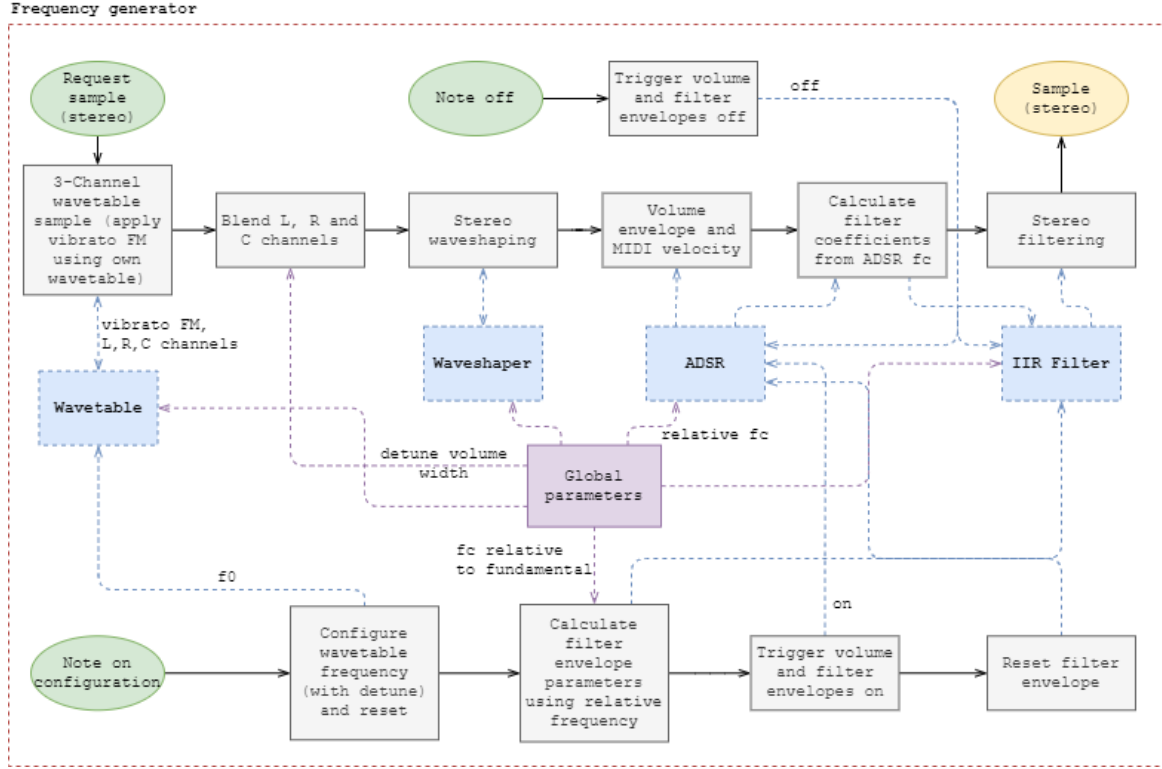
Figure 3.11: System block diagram of the generator

A sinusoidal wavetable with the required vibrato frequency f_v is used to create the frequency deviation samples $df[n]$. The FM is applied to the center, left and right channels.

From equation 3.38, the frequency scaling vibrato factor $2^{\frac{v}{1200}}$ is stored as part of the internal state of the generator, which can be efficiently calculated using the techniques from subsection 2.5.1. Equation 3.39 shows the frequency deviation as a result of vibrato. Note that the frequency deviation is expected to be small (< 50 cents) [43], so no recalculation of the harmonic index of the wavetables is done. On edge cases, this could cause aliasing, but would be unlikely.

$$df[n] = A\sin(2\pi f_v n) \tag{3.39}$$

The C implementation is shown in code listing D.9 in the appendix.

3.8.3. Filter cutoff modulation

The filter cutoff frequency $f_c[n]$ in samples/cycle, is relative to the digital frequency of the note that is playing (f_0), as per the functional requirements. A relative starting and ending cutoff frequency, specified by scaling factors r_s and r_e, are used to determine the amount of cutoff modulation ($m_{filter}[n]$) applied by the filter ADSR envelope. Care is taken to ensure that the filter cut-off does not exceed the Nyquist limit. To ensure that the cut-off always remains within reasonable limits, it is restricted to the audible range (20 Hz to 20 kHz), which limits f_s to a minimum of 40 kHz.

$$f_c[n] = \max(\min(r_s f_0 + f_0(r_e - r_s)m_{filter}[n], \frac{20000}{f_s}), \frac{20}{f_s}) \tag{3.40}$$

From equation 3.40, $r_s f_0$ and $f_0(r_e - r_s)$ are stored in the internal generator state as filter modulation offset and amplitude, respectively. This saves some clock cycles when sampling.

3.8.4. Volume modulation

The note volume $v[n] \in [0,1]$ is determined by the ADSR envelope modulation $m_{vol}[n]$ and the MIDI note velocity $k_{vel} \in \{0,1,...,127\}$, which is a 7-bit unsigned integer that is specified with a note-on MIDI message [44]. Since $m_{vol}[n] \in [0,1]$ by design, k_{vel} is normalised.

$$v[n] = \frac{k_{vel}m_{vol}[n]}{127} \Rightarrow V[n] \in [0,1] \tag{3.41}$$

From equation 3.41, some clock cycles are saved by storing $\frac{k_{vel}}{127}$ in the internal generator state with a note-on configuration.

3.8.5. Stereo width

There is a variety of functions that can be used to introduce stereo width, given center ($y_C[n]$), left ($y_L[n]$) and right ($y_R[n]$) wavetable oscillators that are mixed into stereo audio.

From subsection 2.5.2, the mid-side form of audio is used to specify the mono and stereo content of the audio. As per the requirements, the detune width ($\delta_w \in [0,1]$) and detune volume ($\delta_v \in [0,1]$) parameters are used to blend the detuned oscillators. Volume is a scaling factor, where a width of $\delta_w = 0$ corresponds to no stereo width, and $\delta_w = 1$ to full stereo width. Using these definitions, the mid-side content is defined:

$$M = y_C[n] + \frac{1}{2}\delta_v(2 - \delta_w)(y_R[n] + y_L[n]) \tag{3.42}$$

$$S = \frac{1}{2}\delta_v\delta_w(y_L[n] - y_R[n]) \tag{3.43}$$

Note that when $\delta_w = 1$, the mid channel contains an absolute minimum of $0.5\delta_v$ of the detuned content. This is necessary to ensure that the "mono'd" signal will not result in a complete loss of detune content. $y_C[n]$ is only contained in the mid information. The difference between left and right oscillators (see equation 2.3) is scaled by the detune width and the volume.

Substituting equations 3.42 and 3.43 into equations 2.4 and 2.5, he content of the stereo audio is determined:

$$L = y_C[n] + \delta_v(y_L[n] + (1 - \delta_w)y_R[n])$$

$$R = y_C[n] + \delta_v(y_R[n] + (1 - \delta_w)y_L[n])$$

The C implementation for sampling from a generator is shown in code listing D.10 in the appendix.

3.9. Generator manager

The generator manager is responsible for triggering inactive generator components. This system stores an array of a finite number of generators, which are assigned to notes as appropriate. As per the functional specification, if all generators are active, then the oldest active generator is re-triggered and configured to play any new incoming notes. The expected amount that re-triggering will occur depends on global parameters. A generator is only considered as "inactive" or "available" if it is in the "not playing' state. Otherwise it is "active" .

The amount of samples generated is far greater than the amount of note-on/off requests. However, it is still necessary to make generators available when they transition to their volume envelope "not-playing" phase, which is done once for every buffer request. The active generators are therefore polled at the start of every buffer request to check whether it can be made available. The smaller the buffer, the less the latency (see equation

3.1), but the higher the generator polling rate. Thus, speed is of importance here. It cannot be known when a generator will be available until the release phase is over. It is possible for the release-time parameter to change at any time.

The system block diagram for this component in shown in figure 3.12.
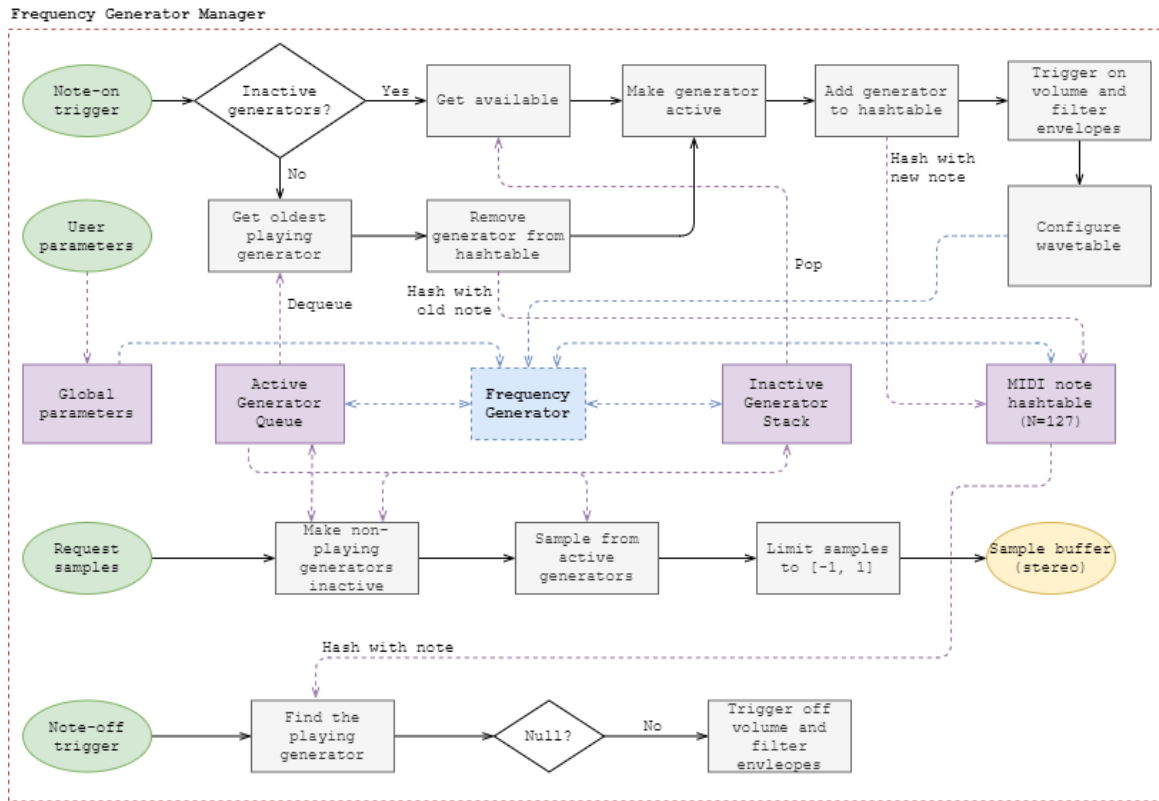
Figure 3.12: Generator manager system block diagram

MCUs often only have a finite amount of memory available to them. Certainty pertaining to the memory consumption of the generators is required. For speed, storing them in an array (and not any other structures such as a C++ vector or queue) is best.

Creating and destroying generator data for every note is an unnecessary process that requires memory allocation, whereas allocating a fixed amount of memory for generators, that are stored at successive memory locations, is preferable.

Pointers to the generator data is used to add generators to other data constructs such as queues and stacks for management. Without calling any memory allocation operations (like "malloc"), the generator data is stored effectively at fixed locations within the heap. It may also help with caching.

See the appendix for the full C implementation in code listing D.11.

3.9.1. Implementing basic data structures with arrays

Efficient management of the generators requires the use of 3 data structures:

1. **Queue**: the FIFO nature of this structure is required to keep track of the order in which generators are triggered. The first item in this queue is the oldest. It also keeps track of all the active generators, so that inactive generators are not sampled as well.

2. **Stack**: the stack has an efficient array implementation which allows for quick access to inactive/available generators.

3. **Hash table**: note-on and note-off triggers come in pairs. It is necessary to keep track of which generator is playing which note, so that it can easily be triggered off. Since the MIDI protocol only supports 128 notes, a hash table provides a quick way to access that generator in $O(1)$ time complexity, with little memory consumption (512 bytes, for a 32-bit system).

With this management scheme, a generator can either be in the active queue, or the available stack, but not in both.

Suppose that the system contains N generators. The queue and the stack are easily implemented with an array of size N, storing generator pointers. Another variable is required that points to the location of the first available slot in the array - the **head pointer**. Inserting into the array, for both the queue and the stack, is done by inserting the item into the head pointer location, and then incrementing the pointer by 1.

Removing an item from the stack, requires decrementing the head pointer by 1, and then retrieving the data at that location. This operation has $O(1)$ time complexity. Removing an item from the queue is more expensive, since the data at location 0 is retrieved. The head pointer is decremented, and all items are shifted one to the left. This operation has $O(N)$ time complexity. In both cases, looping past the item before the head pointer is never done, thus, deleting items is not required.

Note that a more efficient queue implementation is possible ($O(1)$ time complexity), by having a starting and ending index, and treating the array as a circular buffer. However, since N is never realistically very large (≤ 32), and dequeuing items happens less often for large N, it adds unnecessary complexity by requiring extra indexing operations.

The hash table is simple to implement, using an array of size 128, which stores generator pointers. The MIDI note value is used to index into the array, acting as the hashing function. Adding and retrieving items have $O(1)$ time complexity with this implementation, and allows one generator per MIDI note. This system can also be easily modified to use multiple generators per MIDI note, for additional detune effects.

3.9.2. Note-on/off triggers

With reference to figure 3.12, there are two cases to consider for a note-on trigger: triggering if inactive generators are present; re-triggering the oldest generator if all generators are active. To check whether any active generators are available is simple: if the head pointer of the inactive generator stack is at index 0, all generators are inactive.

If there are available generators, a generator is popped off the stack. It is associated with the note in the hash table, and appropriately configured with the correct frequency. This requires two $O(1)$ operations. If no generators are available, then the oldest generator is dequeued, and removed from its associated note in the hash table which is stored in its internal state. This is required so that the note-off trigger for the previous playing note will be ignored. Triggering, configuring and hashing proceeds as per the previous case. This process requires an $O(1)$ and an $O(N)$ operation. If a note-on trigger occurs for the same note before a note-off (which should not happen unless it is artificially forced or through faulty MIDI), then retrieving the generator from the hash table is done. If nothing is present, only then may a new generator be assigned to a note, otherwise the old generator is re-triggered. This is required, since it could lock a generator out from ever being triggered off. This is shown in the C implementation, although not explicitly shown in the block diagram.

For note-off triggers, the MIDI note is used for a hash table lookup. If there is a generator present, it is still associated with that note and is triggered off ($O(1)$). Otherwise, the generator has been re-triggered, and nothing is done.

3.9.3. Sample buffer requests

Once a sample buffer (of a fixed size M) is requested, all the active generators are sampled. To maximise the use of caching, a single generator generates the required M samples before moving on to sampling from the next generator.

This is added to the stereo buffer, and scaled according to the number of voices. Assuming that N voices are managed by this component, the output of each voice is approximately between -1 and +1 (high Q filtering and the detuned wavetables might make this range larger). The output of each voice is therefore scaled by N^{-1}, to ensure that the output is roughly within the -1 and +1 range. Once all the generators are sampled, the buffer is hard-clipped to [-1, +1], so that no bit overflow will occur when converting the buffer data into an appropriate format for the codec IC. This conversion is hardware and application dependent (such as 24-bit PCM audio samples), and is not within scope.

After a buffer request, all inactive generators are freed within the active queue, so that they may be used again. To achieve this in $O(N)$ time, a simple algorithm is used on the active generator queue, demonstrated in figure 3.13. In the figure, 2 buffer requests are shown, with the algorithm being execute twice.
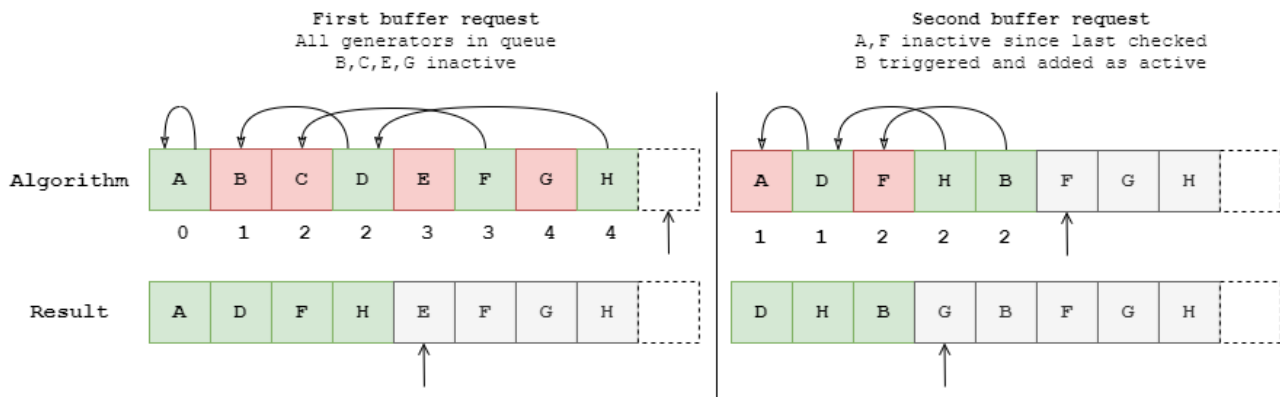
Figure 3.13: Generator queue freeing algorithm example

The generators are labeled from A-H, with available generators shown in red and active generators shown in green. The example manages 8 generators, with the dotted box at the end indicating a non-element location, which the head pointer can be pointed at if the queue is full. The grey elements are items that will never be accessed, since they are past, or at, the head pointer. The algorithm keeps a running count of the number of available generators. If the generator is available, it is pushed onto the available stack and removed from the hash table. Otherwise, if the generator is still active, it is shifted left by the number of counted available generators. The running count is shown below the generators in the figure.

After the algorithm is executed, the head pointer is decremented by the running count. The C implementation of this algorithm ("gm_make_not_playing_available") is seen in listing D.11 in the appendix.

Chapter 4

System testing

Each component of the system is individually tested, along with a final complete system test, and a musical test with some qualitative comments.

Unless otherwise mentioned, all spectrograms are generated with a window size of 512, and a sample overlap of 510 for good time-axis resolution, using a Blackman windowing function for the greatest side-lobe attenuation [45], but large main-lobe width. This windowing function is chosen so that side-lobes do not contaminate the results, which are deemed more important than frequency accuracy. All tests are done at a sampling rate of 44.1 kHz, with 1 second's worth of samples (44100 samples), wavetable LUT sizes of 512 samples and a maximum of 8 generators. All frequency sweeps (chirps) are done from 20 Hz to 10 kHz, since notes above 10 kHz are rarely played. All notes are set to maximum volume (MIDI velocity of 127), where applicable. The data is generated using C code and the implemented components (see appendix E for test code), and analysed using MATLAB. All C and MATLAB code is also pubicly available at https://github.com/marcorad/skripsie.

4.1. Wavetable

A single wavetable set at a frequency of 220 Hz (A3) is sampled with different wavetable positions. This test is done to verify the inter-wavetable interpolation function. Figure 4.1a shows the results. Additional frequencies (55 Hz, 880 Hz) are also tested to show the effect of harmonic indexing, and are shown in figure A.5 in the appendix. To test the harmonic indexing detailed in section 3.4, a linear square-wave chirp is sampled from a single wavetable component, with frequency being updated once per sample. A spectrogram is generated. This is shown in figure 4.1b.

(a) Wavetable position sweep for A3 (220 Hz)   (b) Linear 20 Hz - 10 KHz square-wave chirp (position = 3)
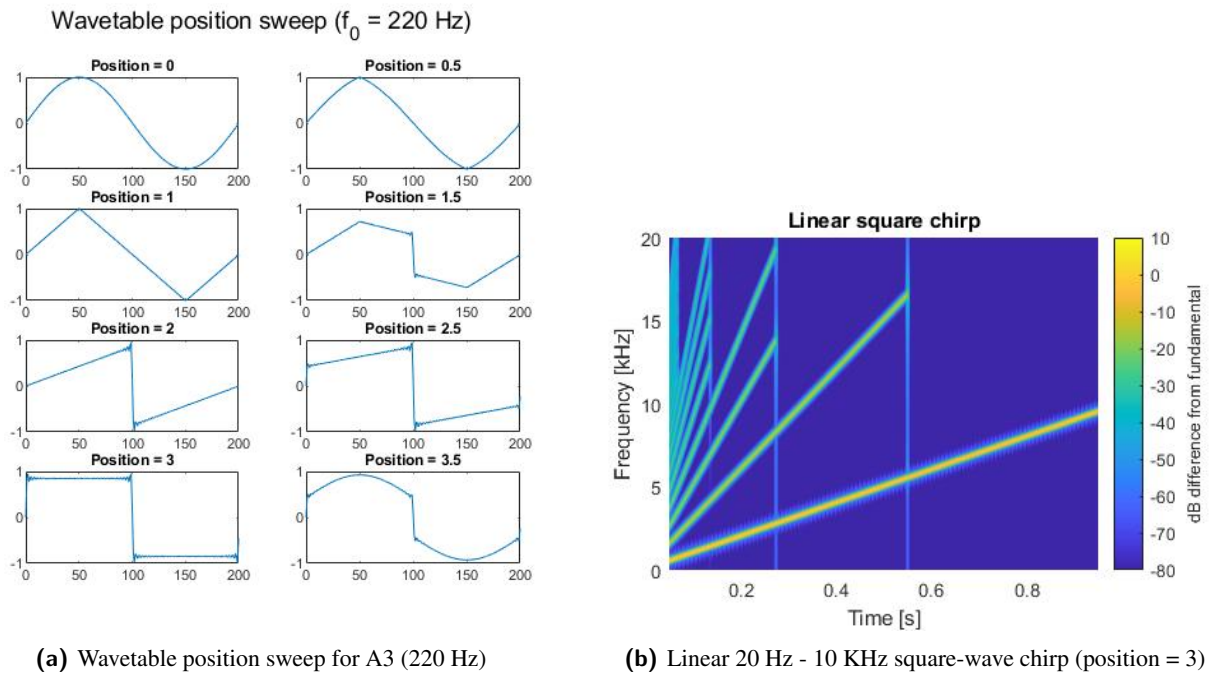
Figure 4.1: Testing inter-wavetable interpolation and harmonic indexing

The inter-wavetable interpolation works as expected. Harmonic indexing also proves effective against

aliasing, with no frequency producing aliased harmonics. The vertical lines in the spectrogram, that is observed when harmonic loss occurs, is result of the previous harmonics instantaneously disappearing, and can be neglected. Also note that the square wave only has uneven harmonics, but only experiences a reduction in total harmonics if the even harmonics will alias, which is why the harmonics do not sweep all the way up to the Nyquist frequency (22.05 kHz).

4.2. ADSR

An ADSR object is sampled and triggered at different points in time. The envelope has time-parameters set for $t_a = 0.125$s, $t_d = 0.125$s, $t_r = 0.125$s with a sustain of 0.5. The exponential LUT is constructed with $p = \frac{2}{3}$. Since 44100 samples were obtained, the triggering times are defined in terms of $k = \lfloor \frac{44100}{8} \rfloor$. A trigger-on is done at sample index $n_0 = \lfloor \frac{k}{3} \rfloor$ ($t \approx 0.042$s). A re-trigger is done at $n_1 = \lfloor n_0 + k + \frac{k}{2} \rfloor$ ($t \approx 0.229$s). A trigger off is done at $n_2 = \lfloor n_1 + 3k \rfloor$ ($t \approx 0.604$s). Figure 4.2 shows the results.

The envelope operates at expected, with the correct timing, sustain level and range. The envelope is successfully re-triggered.
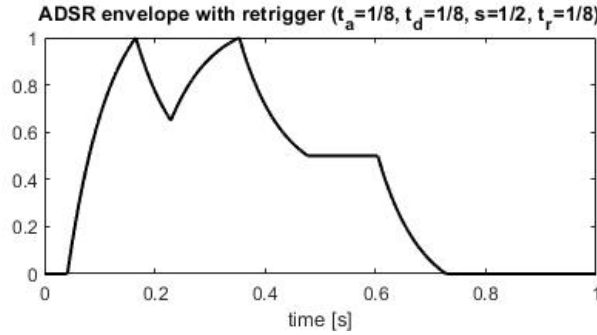
Figure 4.2: Sampled ADSR envelope with retrigger

4.3. IIR filters

The time-dependent cutoff frequency of these filters are tested, which would simultaneously show correct filter operation. It should be noted that pin-point accurate cutoff frequencies are not tested, since deviation of a few Hertz, due to LUT interpolation accuracy, is inconsequential in a musical context. It is only the correct variation over time that is critical.

Guassian white noise ($\mu = 0$, $\sigma^2 = 1$) is generated to test filter operation. The white noise is filtered and a spectrogram is produced, shown in figure 4.3 for the HP24 and BP12 filters. The spectrograms of the other filters (LP24, LP12, HP12) are shown in figure A.6 in the appendix. The white noise contains frequency content at all frequencies, but unequal amounts at any instantaneous point within a STFT. An ADSR envelope is used to modulate the signal, with $t_a = 0.25$s, $t_d = 0.1$s, $t_r = 0.2$s and a sustain of 0.5. The signal is modulated to have a minimum frequency of 1 kHz and a maximum of 20 kHz. The BP12, HP24 and LP24 filters have a Q set to 3, to make the cutoff frequency more apparent. To make the results more interpretable, a 2D Guassion blur is applied to the time-frequency STFT data, using a Guassian smoothing kernel with $\sigma = 15$. This eliminates the jagged edges of the white noise's STFT, but does not change the filter's shape significantly.

The envelope's output is also separately recorded, and plotted as a dotted line for reference. The spectrogram has a window size of 1024, with a sample overlap of 1020, allowing for high time and frequency resolution.
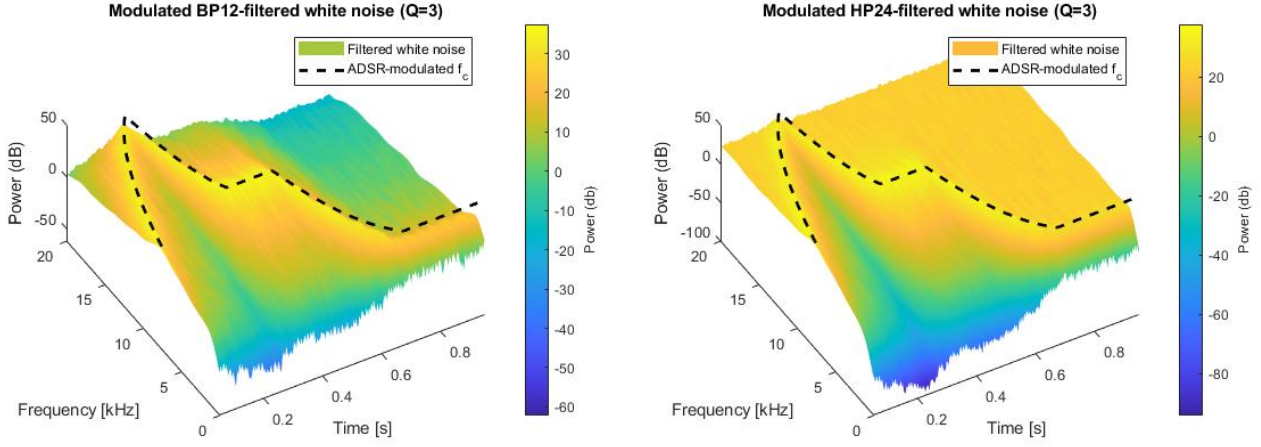
(a) BP12-filtered Guassian noise   **(b)** HP24-filtered Guassian noise

Figure 4.3: Filtered white noise indicating the effect of time-varying filters

The figures clearly show a correct cutoff frequency modulation, with the required range between 1 kHz and 20 kHz. It should be noted that it does not explicitly start at 1 kHz, since the 1024 samples required to perform the first FFT is consumed the start of the envelope.

4.4. Waveshaping

Linear sinuoidal chirps (20 Hz - 10 kHz) are generated by a generator component, with detune volume set to 0; position set to 0; no filtering applied other than for anti-aliasing, and ADSR envelope times set to the minimum of 1 ms with a sustain of 1. The envelope only affects the signal in the attack phase, which occurs at the start of the chirp. This effect is insignificant over the time-span of the recorded data.
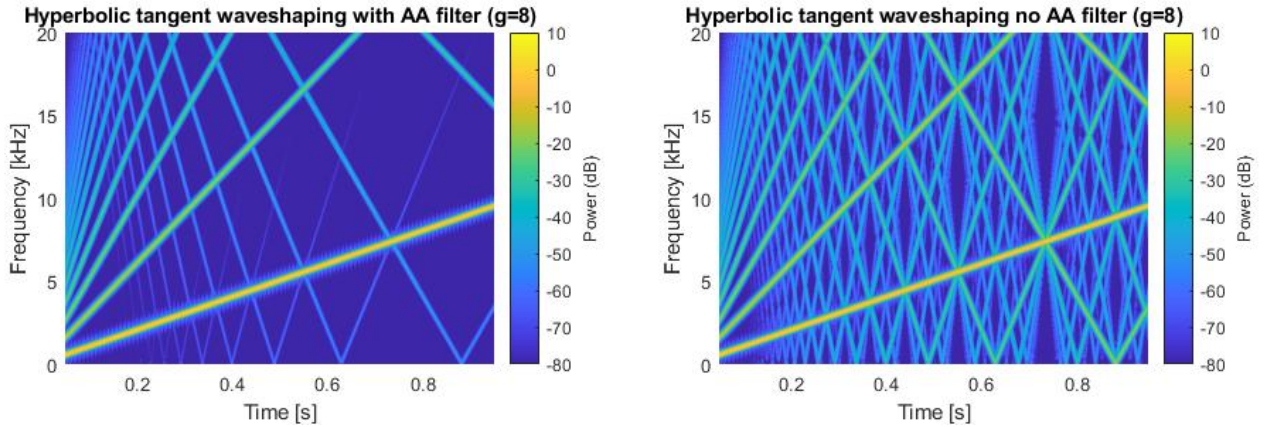
(a) Waveshaped chirp for $g = 8$, with the anti-aliasing filter   **(b)** Waveshaped chirp for $g = 8$, without the anti-aliasing filter

Figure 4.4: Comparing the effect of the anti-aliasing LP12 filter for hyperbolic tangent waveshaping

The gain into both the sinusoidal and the hyperbolic tangent waveshaping functions are swept, with the anti-aliasing LP12 filter's cut-off set to the appropriate value from the look-up table as described in section 3.6. A spectrogram is generated, and compared to the results from having the anti-aliasing filter set to a cutoff of 20 kHz. The comparison is shown in figures 4.4 and 4.6. The output of the STFT is normalised by the maximum value in the time-frequency data, so that the effect of the harmonics relative to the fundamental can more easily be seen.

The reduction in aliasing for the hyperbolic tangent is clear. It is also apparent that the magnitude of the

(a) Waveshaped chirp for $g = \frac{3}{2}\pi$, with the anti-aliasing filter

(b) Waveshaped chirp for $g = \frac{3}{2}\pi$, without the anti-aliasing filter
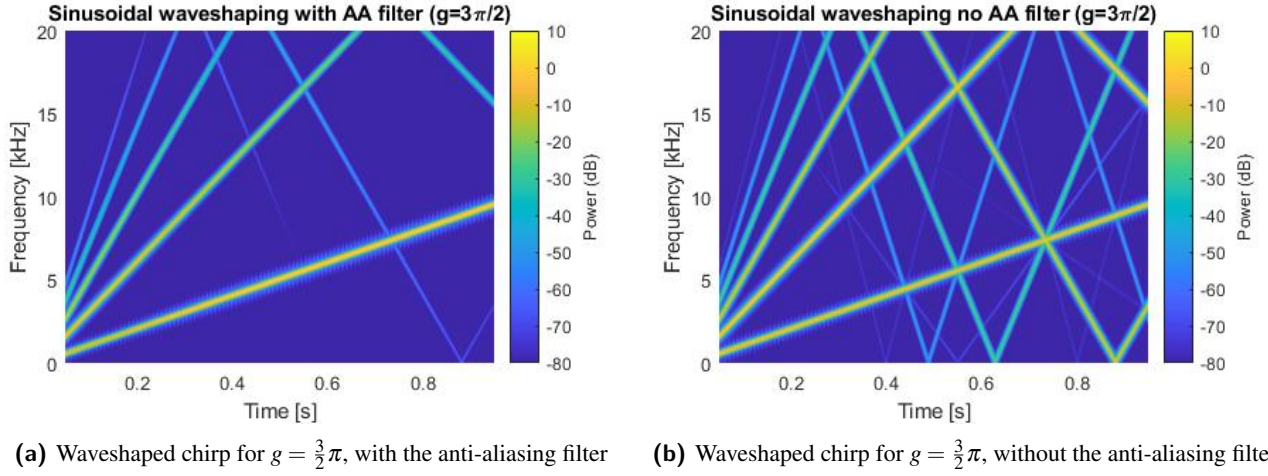
Figure 4.5: Comparing the effect of the anti-aliasing LP12 filter for sinusoidal waveshaping

aliased frequencies in figure 4.4a is smaller than that of 4.4b. Similar observations are made for figure 4.5a and 4.5b, although the extra harmonic content added by sinusoidal waveshaping is less for these examples. Additional gains are also tested, and are shown in figures A.7 to A.10 in the appendix.

4.5. Generator

The vibrato and stereo capabilities of the generator component are tested in this section. All relevant generator parameters are set so that their effect will be non-existing or minimal (detune volume set to 0; ADSR envelope times set to 1 ms, with a 1.0 sustain value; no filtering or waveshaping). A sinusoid with a base frequency of 10 kHz, frequency deviation (vibrato intensity) of 500 cents ($10 \cdot 2^{500/1200} - 10) = 3.35$ kHz deviation), and FM frequency of 2 Hz is sampled. A spectrogram is created with the sampled data normalised by the maximum value in the time-frequency data, along with a straight line that shows the expected center frequency of 10 kHz. This is shown in figure 4.6a.

To test the stereo width, the detune volume is changed to 1, and various stereo widths sampled. The additional left and right oscillators are detuned up and down by 1200 cents (1 octave), respectively. A sinusoid is used, with no filtering applied. The base frequency is arbitrarily set to 100 Hz, which does not matter for the following test.

(a) Sinusoidal vibrato

(b) Lissajous plot of the mid and side stereo content
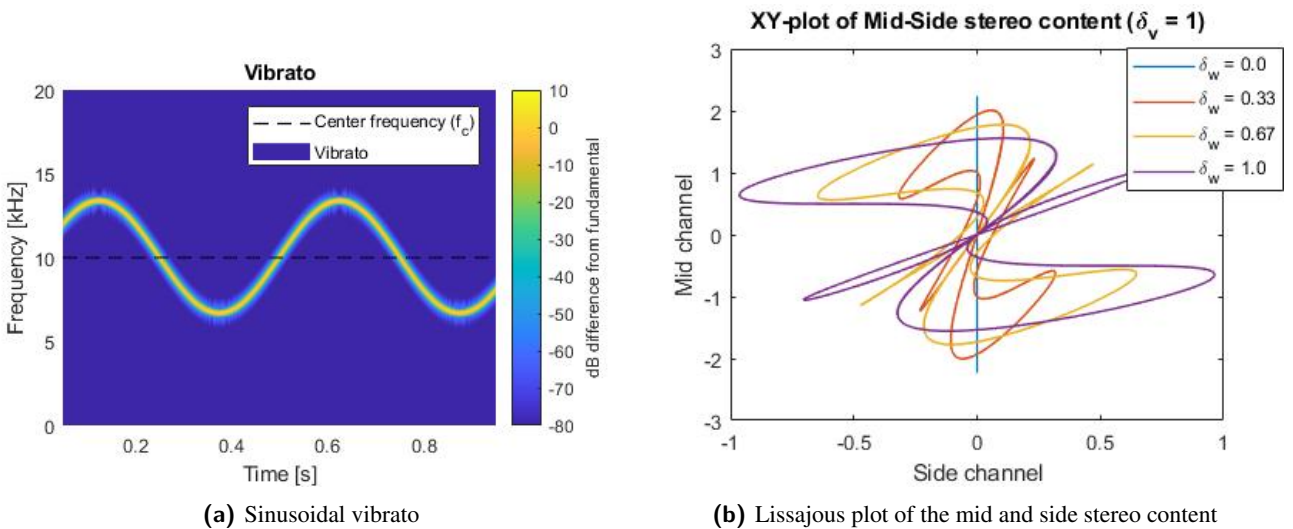
Figure 4.6: Testing vibrato and stereo width

The left and right audio channels are converted to mid and side format, as per equations 2.2 and 2.3. An

XY-plot (commonly known as a Lissajous plot in the music industry) is created from these channels, where the Y-component represents the mid channel and the X-component represents the side channel. For a mono signal ($L = R$), a straight vertical line must be observed. For a phase inverted signal ($L = -R$), a straight horisontal line must be observed. This type of plot is often used in music production to visualise stereo width [46]. The results are shown in figure 4.6b. Figure 4.6a shows that the sinusoidal vibrato acts as expected. Figure 4.6b clearly indicates an increase in side-channel content as the stereo width increases, which can be seen by the axis of symmetry tending towards the side-channel axis.

4.6. Generator manager

Normal operation of the generator manager is tested. This includes note on/off triggers, below capacity. A set of 8 predefined frequencies $\{0.5, 3, ..., 8\}$ kHz is associated with the MIDI notes $\{0, 1, ..., 7\}$. Each sinusoidal tone is triggered on at $t = \{0, 1/16, ..., 7/16\}$ and triggered off in reverse order at $t = \{8/16, 9/16, ..., 15/16\}$. Detune volume is set to 0, ADSR time parameters to 1 ms with a sustain of 1, and no filtering or waveshaping is applied. The results are shown in figure 4.7a.

Generator overload is tested, with frequencies $\{1, 3, ..., 15, 2, 4, ..., 16\}$ kHz associated with MIDI notes $\{0, 1, ..., 15\}$. Each sinusoidal tone is sequentially triggered at $t = \{0, 1/16, ..., 15/16\}$, with no off-triggers occurring. This is shown in figure 4.7b.

(a) Testing normal manager operation

(b) Testing the manager when overloaded
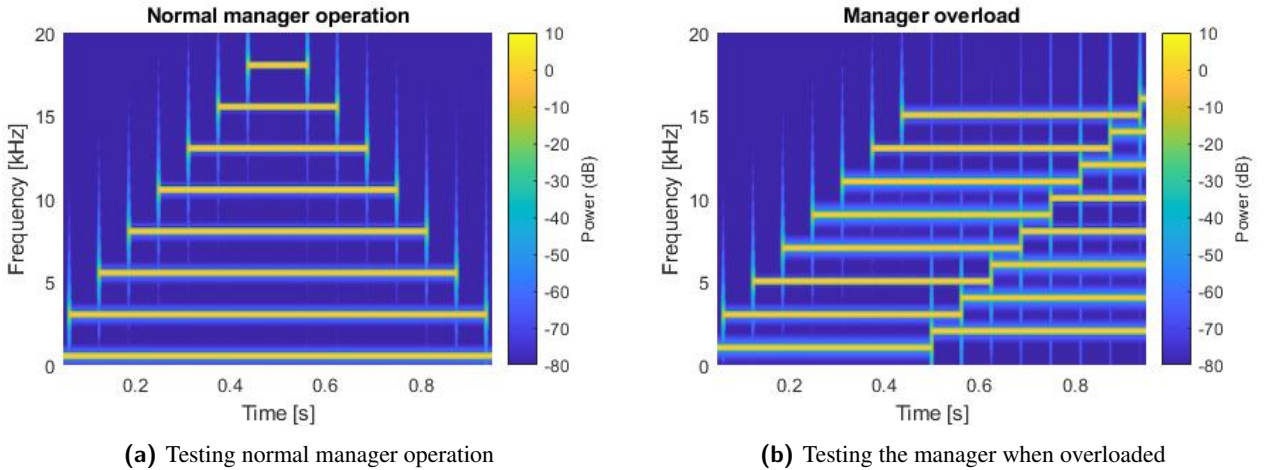
Figure 4.7: Testing the generator manager

For both results, the STFT is normalised by the maximum within the time-frequency data. Also note that the spectral lines at the edges of note on and off triggers are a result of the sudden addition of the tones, and are not of concern. The generator manager operates as required by the specifications, under both normal and overload conditions. The oldest generator is re-triggered.

4.7. Full system test

The entire system is tested by triggering the generator manager with 2 notes, configured with frequencies of 500 Hz and 10 kHz, and associated with MIDI notes 0 and 1, respectively. MIDI note 0 is triggered on at $n = 0$ and off at $n = 22048$ ($t \approx 0.5$ s). MIDI note 1 is triggered on at $n = 11024$ ($t \approx 0.25$ s) and off at $n = 44096$ ($t \approx 1.0$ s). Due to the complexity of this system, the testing conditions are carefully chosen such that the measurements do not appear too chaotic, which is often the case for harmonically rich audio signals.

The parameters are chosen so that their effect can be clearly seen in the time domain and spectrogram data. The wavetables are configured with a position of 2.5, so that both odd and even harmonics are present, a detune of 100 cents (approximately 600 Hz deviation at 10 kHz), detune volume of 0.25 and a detune width of 1, so that the audio channels show a clear frequency and volume distinction, with the detuned waveforms separated in the left and right channels. A LP24 filter with a Q of 7, relative cut-off frequency start of 1 and end of 50 are chosen, such that the filter envelope can be observed due to high Q and sweeps an observable range for MIDI note 0. The filter envelope has an attack of 300 ms, decay of 100 ms, sustain of 0.5 and release of 200 ms. A hyperbolic tangent waveshaper with a gain of 2 is used, along with vibrato, with an intensity of 50 cents and frequency of 5 Hz. The volume envelope has an attack of 10 ms, decay of 200 ms, sustain of 0.5 and release of 200 ms.

Spectrograms are generated for the left and right audio channels, using a window size of 1024 with a 1020 sample overlap. This is shown in figure 4.8a and 4.8b. The audio is also plotted to demonstrate the effect of the volume envelope, shown in figure 4.8c.

(a) Left channel spectrogram

(b) Right channel spectrogram

(c) Left and right channel audio
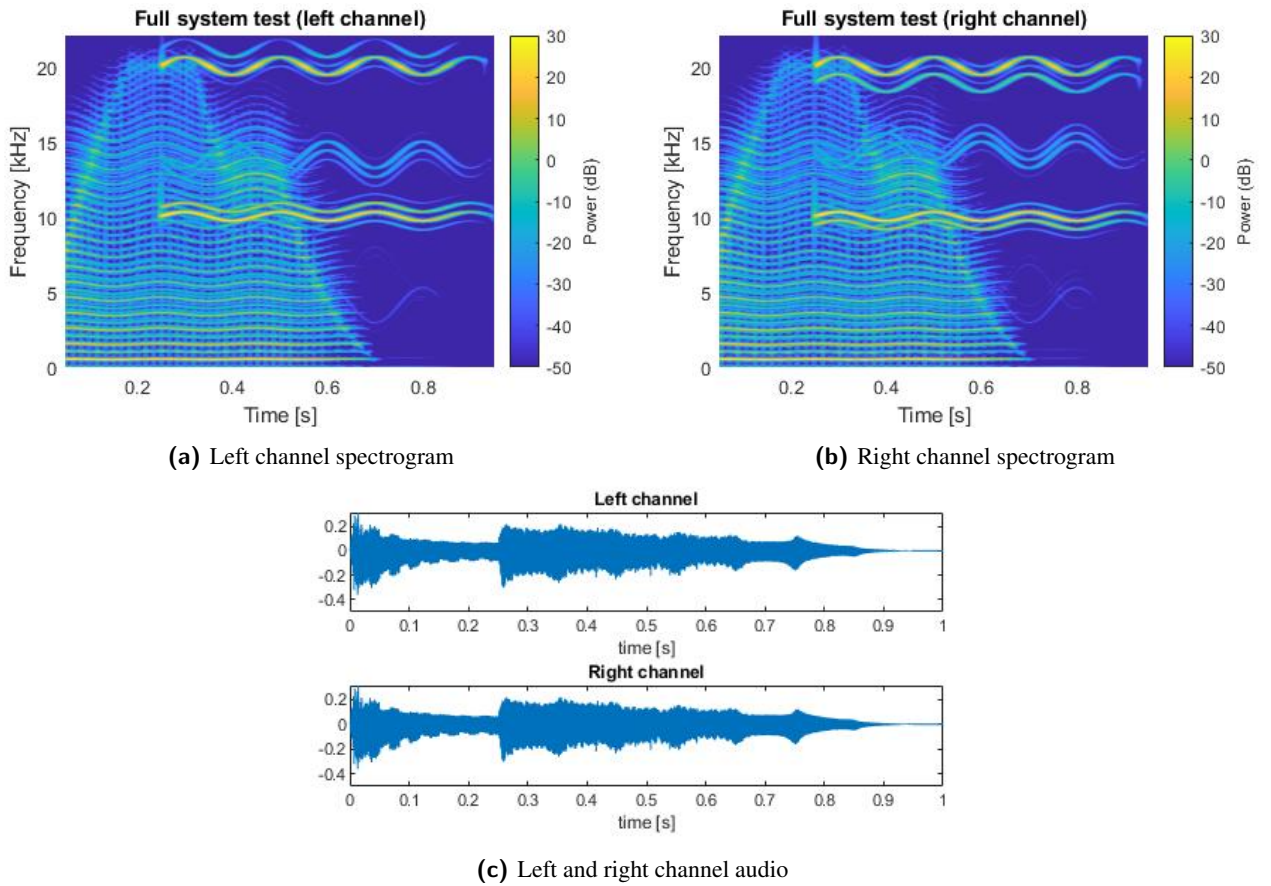
Figure 4.8: Full system test ($f_1 = 500$ Hz, $f_2 = 10$ kHz)

The spectrograms show correct filter envelope modulation, which is limited to 20 kHz as per equation 3.40. Vibrato with the correct deviation and modulation frequencies can clearly be observed for MIDI note 1, with some aliasing present due to waveshaping. Figure 4.8a shows the additional detuned oscillator to be higher in frequency, with reduced amplitude from the fundamental, whereas figure 4.8b shows the additional detuned oscillator to be lower in frequency. Only a single detuned oscillator is present per audio channel, as is expected for a stereo width of 1. Figure 4.8c adequately demonstrates the effect of the volume envelope, but is slightly obscured by amplitude fluctuations resulting from the high-Q filter sweep. The addition of the second generator is clearly be observed at 0.25 seconds.

4.8. Musical test

Using the MIDI input emulation as described in appendix B.1, audio for Beethoven's Moonlight Sonata is generated (played back at 120 BPM). The amount of generators are increased to 32 for this section. The SMF containing the note data can be found at [47]. The audio is publicly available at https://github.com/marcorad/skripsie/tree/main/wav, and on YouTube at https://youtu.be/um-3rZIdguA. This section will provide some qualitative comments on the generated audio, as there is no objective way to measure this.

5 different configuration parameters are used to generate 5 different audio files, and assigned a nickname according to how they sound. See table C.4 for the configuration details, and listing E.1 for the configuration code. Figure 4.9 shows the result of the fifth configuration, which is nicknamed "chiptune", after the sound of old computer games. Figure A.11 in the appendix shows the audio for the other configurations.

(a) The generated audio (b) Zoomed-in waveform
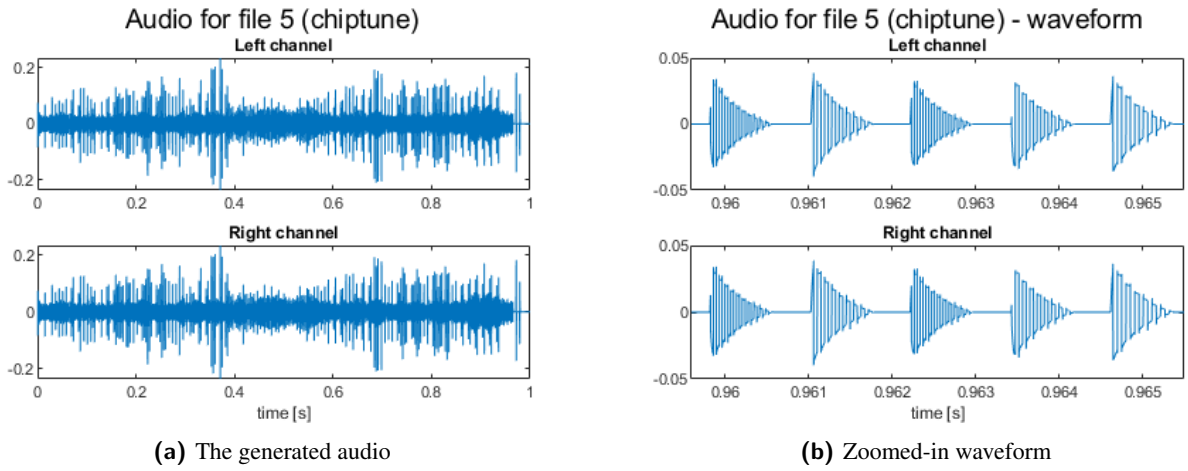
Figure 4.9: Audio generated from configuration 5 (chiptune)

The generated audio shows that this system can successfully create a wide variety of complex sounds, ranging from vintage to comical to theatrical. It is clear that the components of this system can be used to construct a commercially viable product.

There are some drawbacks and nuances in this system, that could be improved. The sudden addition and loss of harmonics, due to the harmonic indexing scheme, can be noticed when listening on high-quality monitors or headphones. This is not immediately noticeable to the average listener. It could be easily improved by using a different harmonic indexing scheme (see section 5.2 for more detail). Furthermore, the anti-aliasing technique used in waveshaping tends to sound "dull" for higher notes, especially for the hyperbolic tangent waveshaping function. This is expected, and can be combated through using 2x or 4x oversampling, which is explicitly neglected in section 3.6, due to processing considerations. However, the waveshaping still proves very useful and unique, especially with octave detunes applied in configuration 3. This drawback is discovered when the detuned oscillators are removed from configuration 3.

Three musicians are asked to provide commentary on the generated audio. Only positive feedback relating to the audio system directly is received, with them being unable to identify the aforementioned drawbacks until it is pointed out to them. A suggestion us made to include an exponentially-mapped volume curve for MIDI velocity to volume conversion, for greater dynamic range. See appendix F for a detailed summary of their feedback.

Chapter 5

Summary and Conclusion

5.1. Results achieved

The audio generation core for a hardware synthesiser is successfully designed and implemented in C. The following, with reference to subsection 1.3.2, is successfully achieved:

1. The basic waveforms (sine, triangle, sawtooth and square) are selected and interpolated according to user parameters, by using the wavetable component. The LUTs required for this task are effectively designed to combat aliasing, while allowing for an arbitrary amount of copies with increasing harmonics. This system supports any sampling frequency, and yields a bit depth of up to 24 bits, if required.

2. Two additional oscillators can be detuned from the base frequency oscillator, and panned so as to achieve a desired stereo width, using the aforementioned wavetable techniques.

3. An ADSR envelope state machine is designed that can produce the required piecewise-exponential behaviour, with re-triggering capabilities.

4. HP12, HP24, LP12, LP24 and BP24 filters, derived from analogue prototypes, are implemented, requiring only 2 table lookups and a single division operation to calculate the FIR coefficients.

5. The cutoff frequency of the available filters can be modulated by an ADSR envelope on a per-sample basis, and set relative to the fundamental frequency.

6. Stereo waveshaping, using the hyperbolic tangent and sinusoid functions, can be performed, with an effective anti-aliasing method being implemented, using a single LP12 filter, with a cutoff frequency set as a function of the gain into the waveshaper. Waveshaping is done efficiently using LUTs.

7. Sinusoidal vibrato at a specified frequency and intensity can be applied to a stereo waveform using the FM capabilities of the wavetable component.

8. Polyphony is achieved using a manager component that can play up to a fixed number of notes, which can do so efficiently in O(N) time. If a new note is triggered when the generator is at capacity, the oldest playing note is re-triggered. The manager produces stereo samples from active generators, and stores it into a buffer of arbitrary size, while providing adequate headroom (samples are within [-1,1] for normal operation, and clipped if necessary).

All the components designed in this thesis are building blocks for creating a more complex audio synthesis core. The building blocks can easily be rearranged in the signal chain to produce different effects, and can be used individually or for other applications. See appendix B.5 for detail on these alternatives and applications.

5.2. Further improvements and work

Designing a wavetable synthesiser can be a highly creative and product-dependent process, with unique features often being the selling point. This thesis only explores and designs the fundamental aspects of what can become a fully-fledged product. The first step in furthering design would be the implementation of the designed software on hardware. An ARM-based implementation is recommended, such as the STM32 Cortex M7 series. A dual-core processor may be desirable, by using the lower-speed core to process user input and handle the display, while using the high-speed core exclusively for DSP. Implementation on the Cortex A7-series MPUs may also be desirable if greater performance is required. Simulating and inspecting the ARM assembly could also be done, where various optimisation techniques can be applied to improve performance. A performance analysis can be done for specific hardware architectures, and optimised accordingly.

Additionally, the designed software can be implemented in VST design, where modifications can be made for concurrency, by sampling generator components on different threads. This could also be beneficial for some Cortex A7 MPUs. The techniques detailed in this thesis could potencially be implemented on an FPGA.

It should be noted that fully-fledged MIDI support is not integrated into this system. Allowing pitch-bend and hold-pedal functionality is considered critical for products. Pitch-bend and hold-pedal functionality would be a possible augmentation to this system, without requiring significant effort. A MIDI-control based system can also be added, that would allow the MIDI protocol to change user parameters, so that the instrument can be externally controlled.

Aside from hardware implementation, many additions can be made to the designed components. Among these additions are the implementation of more filters, such as the comb, or peak filter. The filters can also have more "character" added, by utilising saturation in the feedback path [48]. This can be achieved through including hyperbolic tangent waveshaping in the filtering function, which then saturates the output $y[n]$. This would incur further numerical simulation to avoid aliasing for feedback saturation. Furthermore, the addition of oversampling in combination with the described anti-aliasing waveshaping scheme can be introduced for more robust performance. The optimal thresholds to choose for harmonics in the proposed scheme (section 3.6) can be revisited. The design of the ADSR state-machine can be revisited for optimisation.

The proposed harmonic indexing scheme can be revisited, where a more memory-efficient way of storing LUTs with the required number of harmonics can be devised. This could perhaps be achieved by storing a single waveform with many harmonics, and then filtering and temporarily storing it to reduce harmonic content when required. This requires many design considerations, determined by the effect of linear interpolation in the frequency domain, discussed in section 3.17. This would increase processing requirements, and would be best achieved using a second core, or a multi-threaded system. Such a technique would be hardware dependent, which is why it is not considered in this thesis, which aims to provide a general foundation for these systems. The current technique only requires an external memory for LUT storage, which can be set up to contain many megabytes of waveforms, loaded by the MCU as required.

A system can be designed that would allow for arbitrary parameter modulation, by having a selection of modulator functions that the user can assign to modulate arbitrary paramaters. This would increase computational and implementation complexity significantly.

Finally, integrating other effects, such as a general EQ, delay, phaser and reverb effects, could be beneficial for the release of a finished product. Such effects would require a powerful processor if it is combined with this system, or it can be offloaded to an additional effects processor.

Bibliography

[1] *ADSR A-140*, Doepfer. [Online]. Available: https://doepfer.de/a100_man/A140_man.pdf

[2] *Configure the Coefficients for Digital Biquad Filters in TLV320AIC3xxx Family*, Texas Instruments, April 2010.

[3] MTU Physics, "Tuning: Frequencies for equal-tempered scale, A4 = 440 Hz," last accessed 20 July 2021. [Online]. Available: https://pages.mtu.edu/~suits/notefreqs.html

[4] ARM Developer, "FPU instruction set," last accessed 9 October 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0439/b/BEHJADED

[5] Seveen, "STM32 Synth," 2019, last accessed 16 October 2021. [Online]. Available: https://github.com/Seveen/stm32-synth

[6] audiojs, "List of common audio sample rates," 2017, last accessed 14 October 2021. [Online]. Available: https://github.com/audiojs/sample-rate

[7] Maxim Integrated, "Audio Data Converters - Parametric Search," 2021, last accessed 1 October 2021. [Online]. Available: https://www.maximintegrated.com/en/products/parametric/search.html?fam=audiocodecs&295=OR%7CAudio%20CODEC

[8] S. Lee, "This is the early history of the synthesizer," 2018, last accessed 2 October 2021. [Online]. Available: https://www.redbull.com/gb-en/electronic-music-early-history-of-the-synth

[9] AJH Synth, "Transistor ladder filter," last accessed 2 October 2021. [Online]. Available: https://ajhsynth.com/VCF.html

[10] Make Noise, "Maths," last accessed 2 October 2021. [Online]. Available: https://www.makenoisemusic.com/modules/maths

[11] MusicRadar, "The 16 best eurorack modules 2021: the right modules for any build, or expansion of your modular synthesizer system," 2021, last accessed 2 October 2021. [Online]. Available: https://www.musicradar.com/news/the-best-eurorack-modules-in-the-world

[12] Vintage Synth Explorer, "Yamaha DX7," last accessed 2 October 2021. [Online]. Available: https://www.vintagesynth.com/yamaha/dx7.php

[13] Xfer, "Serum: Advanced wavetable synthesizer," last accessed 3 October 2021. [Online]. Available: https://xferrecords.com/products/serum

[14] Native Instruments, "Massive," last accessed 3 October 2021. [Online]. Available: https://www.native-instruments.com/en/products/komplete/synths/massive/

[15] Arturia, "Pigments: Polychrome software synthesizer," last accessed 4 October 2021. [Online]. Available: https://www.arturia.com/products/analog-classics/pigments/overview#en

[16] D. Karras, "Sound synthesis theory," 2018, last accessed 3 October 2021. [Online]. Available: https://en.wikibooks.org/wiki/Sound_Synthesis_Theory

[17] Ableton, "Live," last accessed 1 October 2021. [Online]. Available: https://www.ableton.com/en/shop/live/

[18] Spectrasonics, "Omnisphere 2.8 - endless possibilities," last accessed 5 October 2021. [Online]. Available: https://www.spectrasonics.net/products/omnisphere/

[19] J. Smith, "Virtual acoustic musical instruments: Review and update," *Journal of New Music Research*, vol. 33, pp. 283–304, 09 2004.

[20] P. Mantione, "The fundamentals of physical modeling synthesis," 2019, last accessed 5 October 2021. [Online]. Available: https://theproaudiofiles.com/physical-modeling-synthesis/

[21] Native Instruments, "Kontakt 6 player," last accessed 5 October 2021. [Online]. Available: https://www.native-instruments.com/en/products/komplete/samplers/kontakt-6-player/

[22] Spitfire Audio, "Spitfire audio," last accessed 5 October 2021. [Online]. Available: https://www.spitfireaudio.com/shop/

[23] M. Apler, "Sound industry history: East vs. west coast synthesis: Moog, buchla, and finding a balance between familiarity and uncertainty in electronic instrument design," last accessed 7 October 2021. [Online]. Available: https://hii-mag.com/allposts/industry-historyeastwest

[24] Detroit Modular, "Eurorack: Modules," last accessed 7 October 2021. [Online]. Available: https://www.detroitmodular.com/eurorack.html

[25] Doepfer, "A-111-3 Micro Precision VCO / VCLFO," last accessed 7 October 2021. [Online]. Available: https://doepfer.de/A1113.htm

[26] IntelliJel, "The little filter that could," last accessed 7 October 2021. [Online]. Available: https://intellijel.com/shop/eurorack/uvcf/

[27] Nord, "Nord Stage 3," last accessed 7 October 2021. [Online]. Available: https://www.nordkeyboards.com/products/nord-stage-3

[28] *VCA Module*, MFB. [Online]. Available: http://mfberlin.de/wp-content/uploads/VCA_english.pdf

[29] S. Pigeon, "The non-linearities of the Human Ear," last accessed 7 October 2021. [Online]. Available: https://www.audiocheck.net/soundtests_nonlinear.php

[30] C. Schmidt-Jones, "Tuning systems," *Connexions*, 2005.

[31] HyperPhysics, "Equal temperament," last accessed 7 October 2021. [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/Music/et.html

[32] H. Robjohns, "Q. How does Mid-Sides encoding/decoding actually work?" 2017, last accessed 8 July 2021. [Online]. Available: https://www.soundonsound.com/sound-advice/q-how-does-mid-sides-recording-actually-work

[33] A. Habibi, "Comparison of nth-Order DPCM Encoder With Linear Transformations and Block Quantization Techniques," *IEEE Transactions on Communication Technology*, vol. 19, no. 6, pp. 948–956, 1971.

[34] R. N. Bracewell and R. N. Bracewell, *The Fourier transform and its applications*. McGraw-Hill New York, 1986, vol. 31999, ch. 9.

[35] Wikimedia Foundation, Inc., "Audio bit depth," 2021, last accessed 8 October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Audio_bit_depth

[36] R. H. Jack, T. Stockman, and A. McPherson, "Effect of latency on performer interaction and subjective quality assessment of a digital musical instrument," in *Proceedings of the audio mostly 2016*, 2016, pp. 116–123.

[37] T. Blu, P. Thevenaz, and M. Unser, "Linear interpolation revitalized," *IEEE Transactions on Image Processing*, vol. 13, no. 5, pp. 710–719, 2004.

[38] GeeksForGeeks, "Compute modulus division by a power-of-2-number," 2021, last accessed 22 July 2021. [Online]. Available: https://www.geeksforgeeks.org/compute-modulus-division-by-a-power-of-2-number/

[39] J. O. Smith, *Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications*, 2nd ed. W3K Publishing, 2007.

[40] Analog Devices, *Linear Circuit Design Handbook*. Newnes/Elsevier, 2007, ch. 8.

[41] J. Timoney and V. Lazzarini, "New perspectives on distortion synthesis for virtual analog oscillators," *Computer Music Journal*, vol. 34, pp. 28–40, 2010.

[42] T. Thomas and S. Sekhar, *Communication Theory*. Tata-McGraw Hill, 2005, p. 136.

[43] J. P. Nix, "Listener preferences for vibrato rate and extent in synthesized vocal samples," in *Proceedings of Meetings on Acoustics 169ASA*, vol. 23, no. 1. Acoustical Society of America, 2015, p. 035002.

[44] MIDI Association, "Summary of MIDI 1.0 Messages," 2021, last accessed 9 October 2021. [Online]. Available: https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message

[45] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Prentice-Hall Inc., 1999, pp. 468–471.

[46] D. Mellor, "Visualizing stereo information using Lissajous figures," 2016, last accessed 26 October 2021. [Online]. Available: https://www.adventures-in-audio.com/visualizing-stereo-information-using-lissajous-figures

[47] BitMidi, "Beethoven-Moonlight-Sonata.mid," 2018, last accessed 2 October 2021. [Online]. Available: https://bitmidi.com/beethoven-moonlight-sonata-mid

[48] J. Timoney and V. Lazzarini, "Saturation non-linearities for Virtual Analog filters," *Forum Acusticum*, 2011.

[49] C. Stuart, "C++ library for parsing Standard MIDI Files," 2021, last accessed 5 July 2021. [Online]. Available: https://midifile.sapp.org/

[50] Wikimedia Foundation, Inc., "Q (number format)," 2021, last accessed 10 October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Q_(number_format)

[51] M. Bhojasia, "Explain Logical and Arithmetic Shifts in C Language with Examples," last accessed 10 October 2021. [Online]. Available: https://www.sanfoundry.com/c-tutorials-logical-arithmetic-shifts-uses/

Appendix A

Additional figures

(a) Doepfer A-111-3   (b) IntelliJel UVCF   (c) MFB VCA   (d) Doepfer A-140
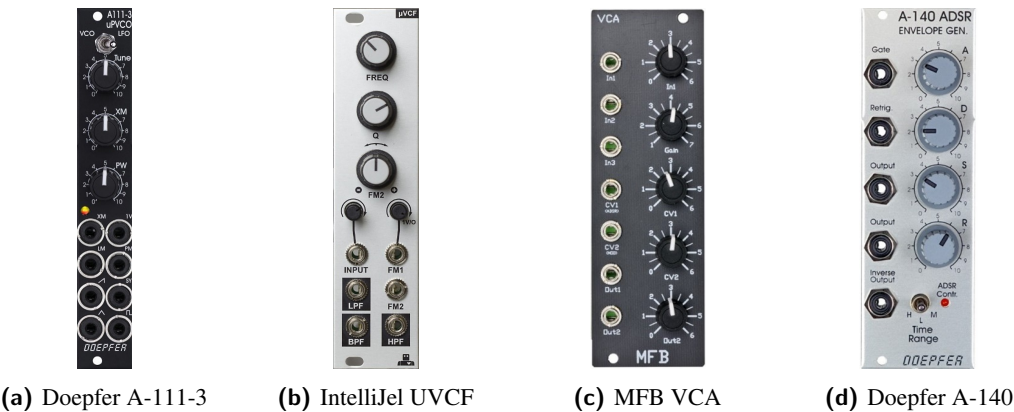
Figure A.1: Examples of fundamental modules
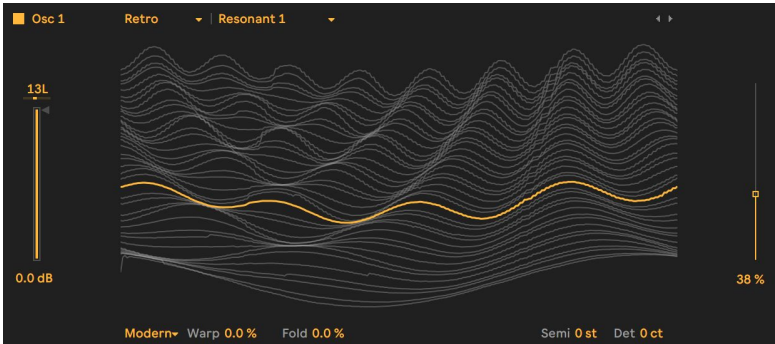
Figure A.2: Ableton's Wavetable VST

(a) A typical ADSR trigger with a gate signal   (b) A typical ADSR re-trigger
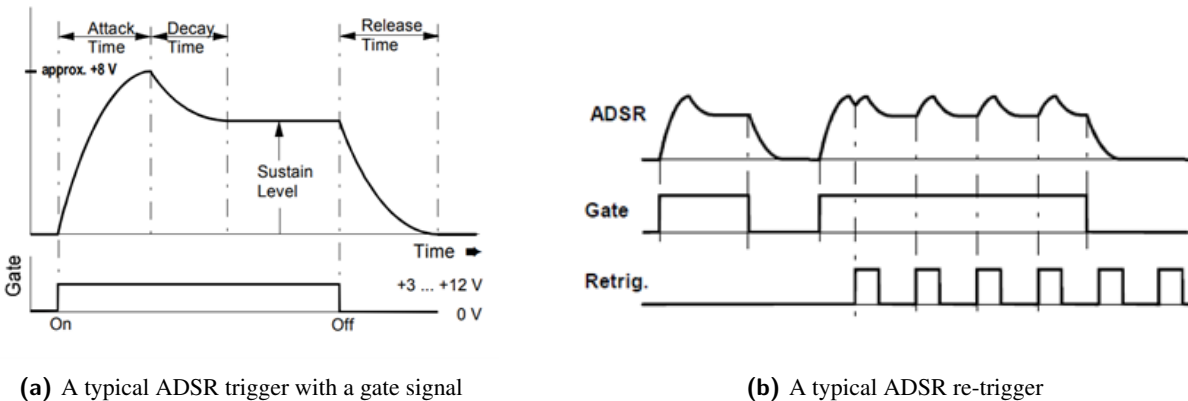
Figure A.3: Doepfer A-140 ADSR operation [1]
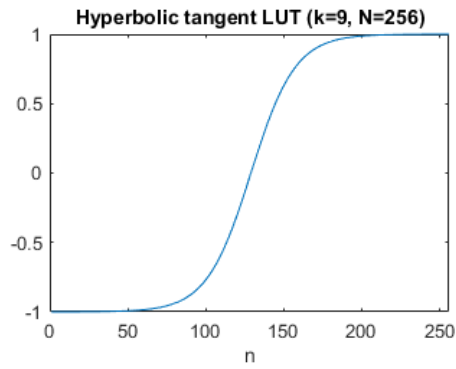
Figure A.4: Hyperbolic tangent LUT
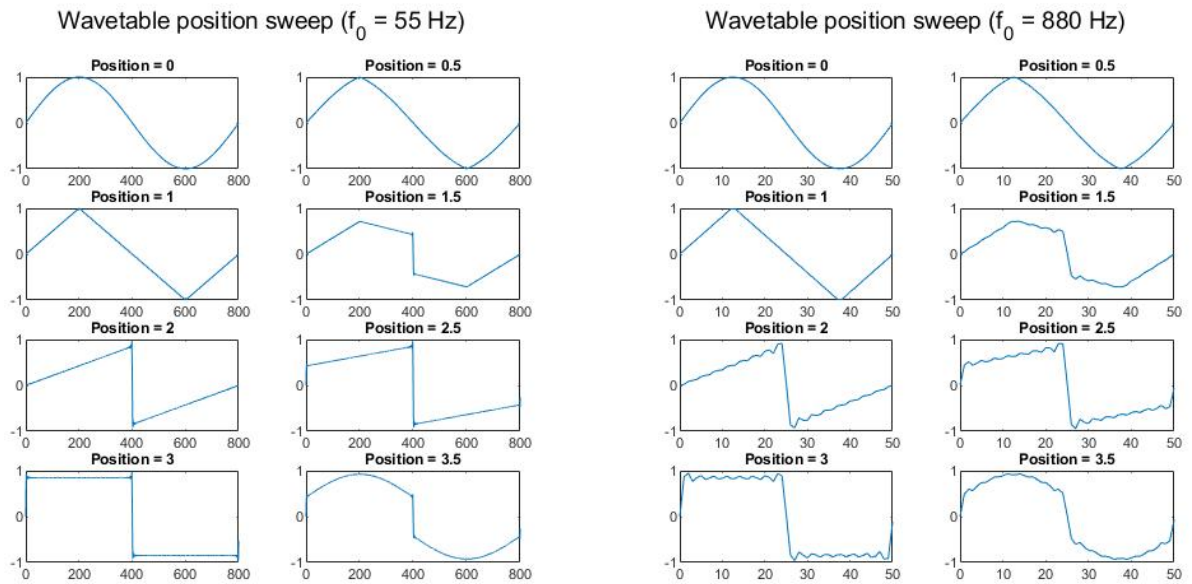
(a) Wavetable position sweep for A1 (55 Hz)

(b) Wavetable position sweep for A5 (880 Hz)

Figure A.5: Testing inter-wavetable interpolation at different frequencies

(a) LP12-filtered Guassian noise

(b) HP12-filtered Guassian noise

(c) LP24-filtered Guassian noise
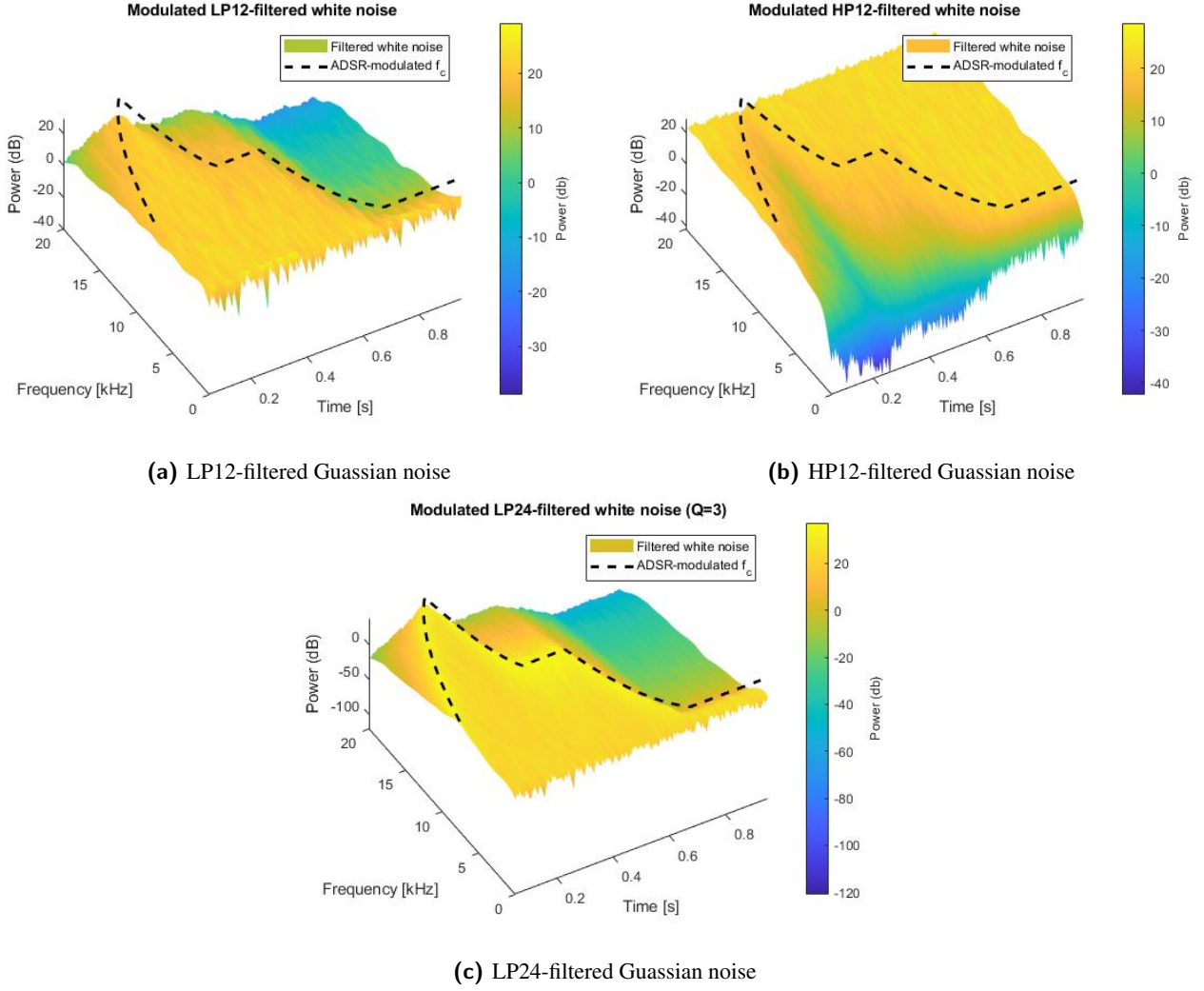
Figure A.6: Filtered white noise indicating the effect of time-varying filters

(a) Waveshaped chirp for $g = 2$, with anti-aliasing filter

(b) Waveshaped chirp for $g = 2$, without the anti-aliasing filter
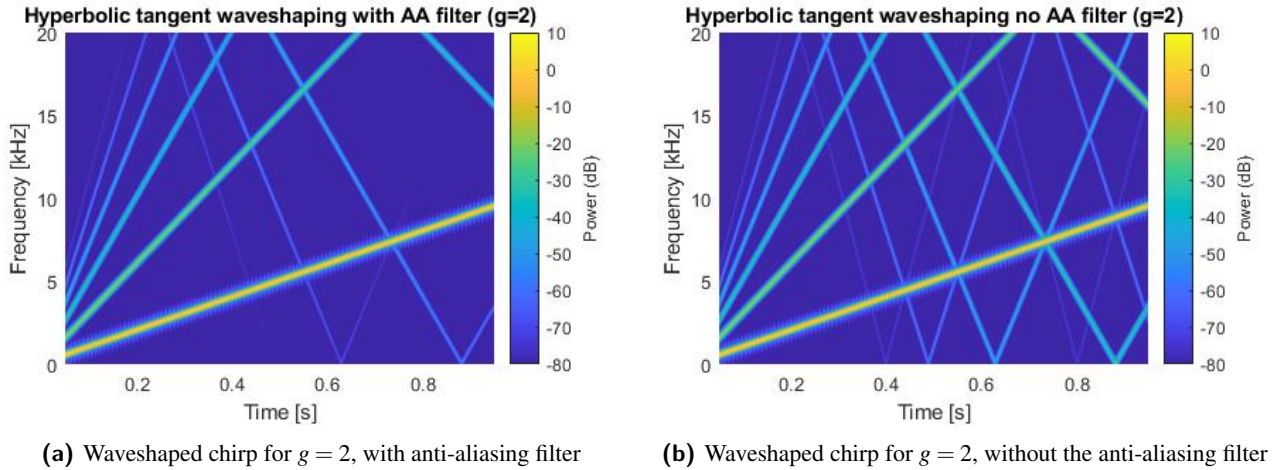
Figure A.7: Comparing the effect of the anti-aliasing LP12 filter for hyperbolic tangent waveshaping
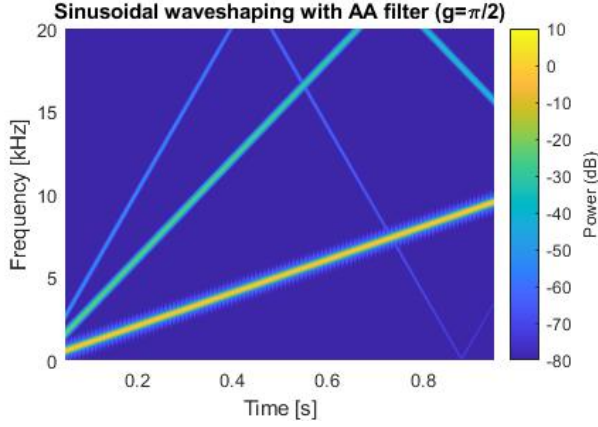
(a) Waveshaped chirp for $g = \frac{1}{2}\pi$, with anti-aliasing filter
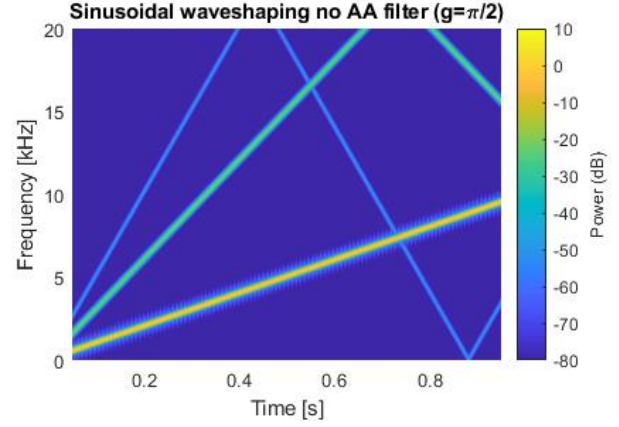
(b) Waveshaped chirp for $g = \frac{1}{2}\pi$, without the anti-aliasing filter

Figure A.8: Comparing the effect of the anti-aliasing LP12 filter for sinusoidal waveshaping
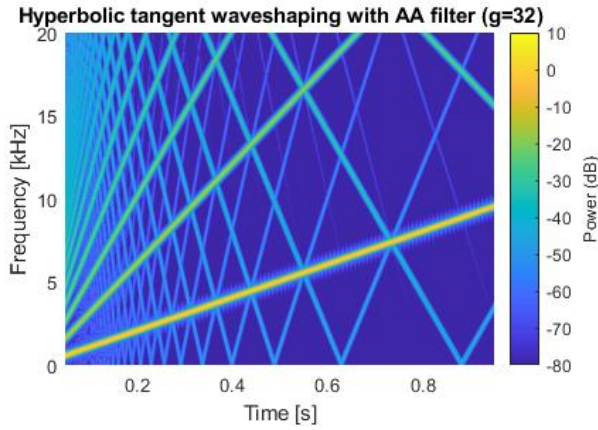
(a) Waveshaped chirp for $g = 32$, with anti-aliasing filter
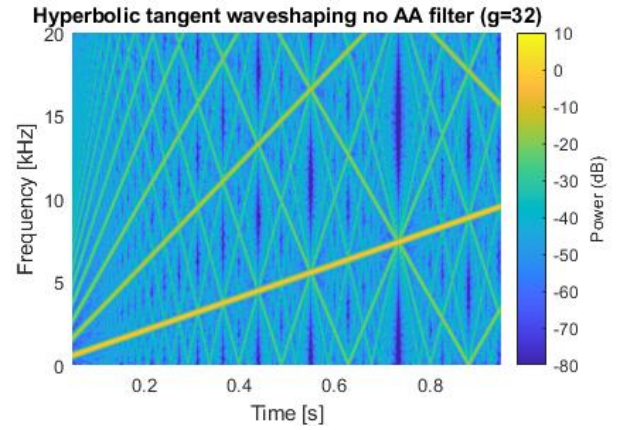
(b) Waveshaped chirp for $g = 32$, without the anti-aliasing filter

Figure A.9: Comparing the effect of the anti-aliasing LP12 filter for hyperbolic tangent waveshaping
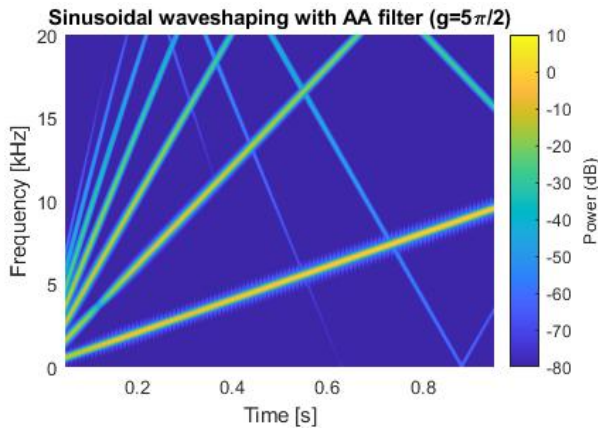
(a) Waveshaped chirp for $g = \frac{5}{2}\pi$, with anti-aliasing filter
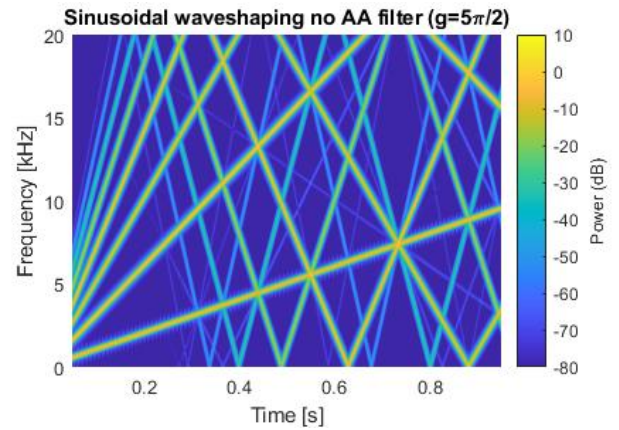
(b) Waveshaped chirp for $g = \frac{5}{2}\pi$, without the anti-aliasing filter

Figure A.10: Comparing the effect of the anti-aliasing LP12 filter for sinusoidal waveshaping

(a) Configuration 1 (string ensemble)

(b) Configuration 2 (wah-voice)

(c) Configuration 3 (hard harpsichord)

(d) Configuration 4 (vintage detune)
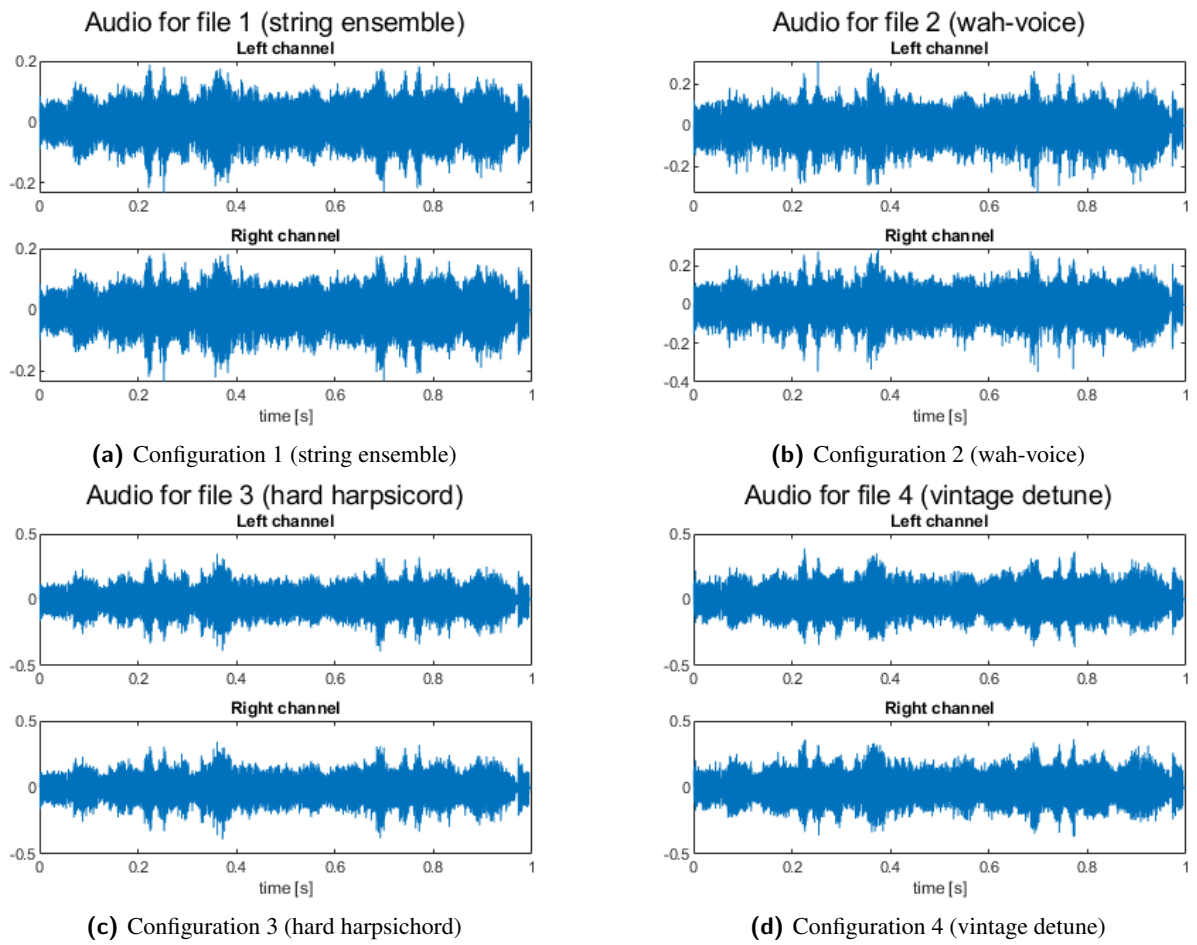
Figure A.11: Additional audio files generated for Beethoven's Moonlight Sonata

Appendix B

Additional sections and information

B.1. MIDI

The MIDI protocol is traditionally a communications standard specified for electronic inter-instrument communication. This protocol has been specified for a variety of mediums, which include USB 1.0 and 2.0, data storage in ".mid" files (SMF), and the original UART-based protocol that interfaces with a 5-pin MIDI (DIN) connector (of which only 3 pins are used).

A MIDI message consists of 3 bytes [44]. The first byte contains status information, which indicates the message type. Messages can be sent on one of 16 channels, which are identified in the status byte, if applicable. There are 5 message types:

1. Channel Voice: contains note information, such as note on/off, aftertouch, pressure and control change. This is the most important message for the application at hand.

2. Channel Mode: sound off; reset; all notes off.

3. System Common: manufacturer information; song select and position; and tune request.

4. System Real-Time: intended for timing and sequencing, which includes the timing clock (24 times per quarter note) and start and stop sequencing.

5. System Exclusive (Sysex): specific to the device.

The Midifile library [49] is used to convert SMFs into interpretable data that can be used to drive our system. The library is used to convert an SMF to note information that includes a starting and ending timestamp. To emulate the process that would occur on an MCU, the data is processed in fixed chunks of size 128, which is stored in a buffer and then written to a wav file. The code is shown in listing B.1 below.

All code is publicly available at https://github.com/marcorad/skripsie.

```
1  struct note {
2    uint8_t note;
3    float start_seconds;
4    float duration_seconds;
5    float vel;
6  };
7
8  std::vector<note> note_queue;
9
10
11 void load_notes(const std::string& file, uint8_t track, bool debug = false) {
12   note_queue.clear();
13   smf::MidiFile midi;
14   //midi.read("..\\midilib\\fireflies.mid");
15   midi.read(file);
16   //cout << "NUM TRACKS:" << midi.getNumTracks() << endl;
17
18   midi.linkEventPairs();
19   midi.linkNotePairs();
```

```
20    midi.doTimeAnalysis();
21    float T = 1.0f / FS * PLAYBACK_BUFFER_SIZE;
22    for (int i = 0; i < midi[track].size(); i++) {
23      if (!midi[track][i].isNoteOn()) {
24        continue;
25      }
26      auto note_on = midi[track][i];
27      note n = { note_on[1], note_on.seconds, midi[track][i].getDurationInSeconds(),
      note_on[2] };
28        note_queue.push_back(n);
29        if(debug) std::cout << note_names[(int)n.note] << " (" << (int)n.note << ")
      " <<
30          " @ t=[" << n.start_seconds << ", " << (n.start_seconds + n.duration_seconds)
       << "] (dt="
31          << n.duration_seconds << ")" << std::endl;
32    }
33    //sort according to trigger on
34    std::sort(note_queue.begin(), note_queue.end(), [](const note& n1, const note& n2){
35      return n1.start_seconds < n2.start_seconds;
36      });
37  }
38
39  //simulates the PLAYBACK_BUF_SIZE nature of dealing with a queue of note events
40  void write_midi_to_wav(gen_manager* gm, gen_config* gc, const std::string& name, bool
       debug = false) {
41    using namespace std;
42
43    float dt = (float)PLAYBACK_BUFFER_SIZE / FS;
44    float t = note_queue[0].start_seconds - 0.1f; //start time with 100ms space before
      starting
45    float start_t = t;
46    float end_t = note_queue.back().start_seconds + note_queue.back().duration_seconds
      + 1.0f; //end time with 1s space (for possible decays)
47    uint32_t N = (uint32_t)( (end_t - t) * FS );
48
49    float T = 1.0f / FS * PLAYBACK_BUFFER_SIZE; // time taken to play one buffer of
      samples
50
51    float* bufL = new float[N];
52    float* bufR = new float[N];
53
54    uint32_t sample_index = 0;
55
56    std::vector<note> playing = {};
57
58    unsigned int i = 0;
59    std::vector<note> trigger_off = {};
60
61    while (t <= end_t) {
62      //lambda to see if note must be triggered off
63      auto l = [&](const note& n) {
64        return n.start_seconds + n.duration_seconds <= t ;
65      };
66
```

```
67      //copy notes that must be triggered off
68      std::copy_if(playing.begin(), playing.end(), std::back_inserter(trigger_off), l);
69
70      //remove them
71      playing.erase(std::remove_if(playing.begin(), playing.end(), l), playing.end());
72
73      std::for_each(trigger_off.begin(), trigger_off.end(), [&](const note& n) {
74        //TRIGGER OFF
75        if (debug) cout << (n.start_seconds - start_t) << " note off: " << note_names[n
    .note] << endl;
76        gm_trigger_note_off(gm, n.note);
77        });
78      trigger_off.clear();
79
80      //find trigger on events
81      while (i < note_queue.size()) {
82        note n = note_queue[i];
83        if (n.start_seconds < t) {
84          playing.push_back(n);
85
86          //TRIGGER ON
87          if (debug) cout << (n.start_seconds - start_t) << " note on: " << note_names[
    n.note] << endl;
88          gm_trigger_note_on(gm, gc, n.note, n.vel);
89
90          i++;
91        }
92        else {
93          break;
94        }
95      }
96
97      //write samples
98      uint32_t sample_size = std::min<uint32_t>(PLAYBACK_BUFFER_SIZE, N - sample_index)
    ;
99      gm_write_n_samples(gm, gc, bufL + sample_index, bufR + sample_index, sample_size)
    ;
100
101     //update time and index
102     t += dt;
103     sample_index += sample_size;
104   }
105
106   //write to file
107   write_to_wav(name, bufL, bufR, N, FS);
108 }
```

Listing B.1: Simulating real-time MIDI input via SMF data.

Code was written to a wav file, using the correct wav file header information. 16-bit audio at 44.1 kHz is used. This is shown in listing B.2 below.

```
1 // WAVE PCM soundfile format (you can find more in http://soundfile.sapp.org/doc/
    WaveFormat/ )
2 typedef struct header_file
3 {
```

```cpp
4    char chunk_id[4] = { 'R','I','F','F'};
5    int chunk_size; //TO BE FILLED IN:  36 + SubChunk2Size
6    char format[4] = { 'W','A','V','E' };
7    char subchunk1_id[4] = { 'f','m','t',' ' };
8    int subchunk1_size = 16; //16 for PCM
9    short int audio_format = 1; // PCM = 1
10   short int num_channels = 2; //stereo
11   int sample_rate;        //TO BE FILLED IN: sample_rate denotes the sampling rate.
12   int byte_rate; //TO BE FILLED IN: SampleRate * NumChannels * BitsPerSample/8
13   short int block_align = 2 * 16 / 8; //NumChannels * BitsPerSample/8
14   short int bits_per_sample = 16; //bits per sample
15   char subchunk2_id[4] = { 'd','a','t','a' };
16   int subchunk2_size;      //TO BE FILLED IN: NumSamples * NumChannels * BitsPerSample
        /8
17 } header;
18
19 typedef struct header_file* header_p;
20
21 void write_to_wav(const std::string& name, float bufL[], float bufR[], int
       num_samples, int fs) {
22   using namespace std;
23   header head;
24   head.sample_rate = fs;
25   head.byte_rate = fs * 2 * 16 / 8;
26   head.subchunk2_size = num_samples * 2 * 16 / 8;
27   head.chunk_size = 36 + head.subchunk2_size;
28
29   ofstream f;
30   f.open("..\\wav\\" + name + ".wav", ios::binary);
31   f.write(reinterpret_cast<char*> (&head), sizeof(header));
32
33   for (int i = 0; i < num_samples; i++) {
34     short int s = (int)(bufL[i] * (float)INT16_MAX);
35     f.write(reinterpret_cast<char*> (&s), sizeof(short int));
36     s = (int)(bufR[i] * (float)INT16_MAX);
37     f.write(reinterpret_cast<char*> (&s), sizeof(short int));
38   }
39   f.close();
40 }
```

Listing B.2: Writing to a wav file.

B.2. A basic monophonic modular setup

Figure B.1 shows a block diagram of a typical modular setup. Many keyboard synthesisers follow this topology.

The system consists of a note controller, which sends modulating signals to all of the modules. The modules have user-defined parameters, that are also modulated by ADSR sources.

The VCO generates the audio signal, which is then attenuated accordingly by the VCA and the volume ADSR envelope. The gain-modulated signal then enters the filter, which has a cut-off parameter that is modulated by a filter ADSR envelope. Note that the VCF and VCA can be swapped in order. The ADSR has very low-frequency content, which makes the order of these modules mostly inconsequential. The transient response of the VCF will be affected by the order.
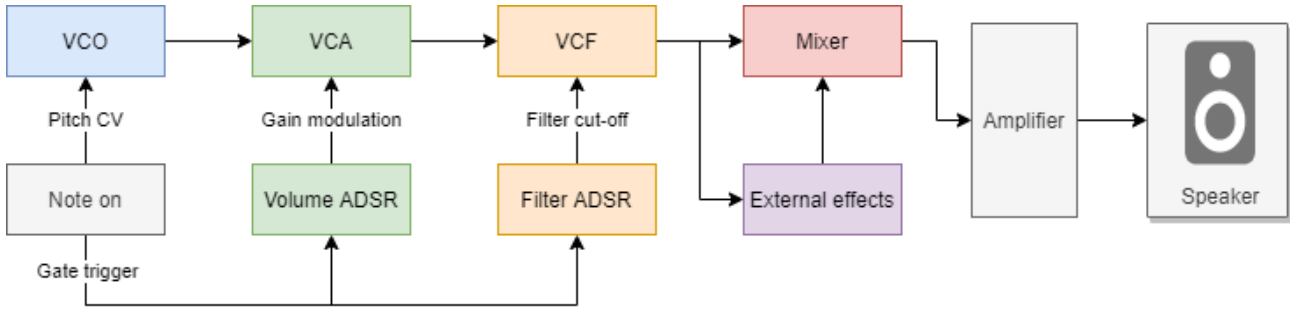
Figure B.1: Block diagram of a very basic monophonic modular setup.

The final modified signal is sent to a mixer, which can apply additional effects such as distortion, reverb or delay (either in series or parallel). These effects could be internally included in the synthesiser, or part of an FX loop. The amplification and speaker is not necessarily included, depending on the output of the synthesiser (could be a line-out, a digital signal within a DAW or internal speakers). It should be noted that many in-between steps are often included at the discretion of the designer. An example would be distortion/saturation between the VCO and VCA.

B.3. Regarding integer arithmetic using Q-numbers

The system in this thesis was partially written for integer arithmetic using Q-numbers, but then later rewritten due to a variety of factors.

Q-numbers is a way of representing fractional data using only integers, with an explicit denominator that is a power of 2. For example, a Q15 number is stored as $n_{Q15} = \frac{n}{2^{15}}$. This allows for storage of rational numbers, with a linear quantisation interval, whereas single-precision floating-point numbers have 23 bits of resolution for a power-of-2 associated exponent. Thus, floating point numbers have a varying quantisation over the numbers that they store, and can prove useful for dealing with small and large numbers alike.

Q-number multiplication is shown in the C-implementation below, which consists of casting, integer multiplication and bit-shift operations. The C-implementations for Q-number multiplication and division can be found at [50].

It should be noted that a 32-bit Q-number multiplication requires casts to a 64 bit integers and their multiplication. Thus, for allowing 24-bit audio resolution, arithmetic must be performed with 64-bit numbers, which is impractical on 32-bit processors and ends up being slower than floating-point FPU operations.

Reduced precision at lower values (such as values close to 0), also proved to be inaccurate when interpolating the sinusoidal LUT for filter coefficient calculation. The resulting filter coefficients at low cut-off frequencies were wildly inaccurate and caused unpredictable filter behaviour. Furthermore, the range is also limited by the Q-number. For example, a 16-bit signed Q15 number can only store values between -1 and +1 before overflowing. This can be problematic for very resonant filters, and requires the constant re-scaling of numbers so that it remains within range, which results in further precision loss and a higher noise floor. For signed numbers, it must also be guaranteed that bit-shifting compiles to an arithmetic shift ASM operation, which can be compiler dependent, making it unreliable for porting to different platforms [51].

An FPU floating point operation is less expensive than a Q-number multiplication, since an FPU operation requires 1 clock cycle (see table C.3) to load data from processor to FPU and 1 clock cycle to retrieve. Thus, a multiplication requires, at most, 3 clock cycles. This is reduced if data does not need to be shifted to and from the main processor.

Q-number multiplication performance can also be comparable to FPU multiplication, if saturation is not

included. However, the lack of 24-bit audio resolution for 32-bit systems without requiring 64-bit integers and the limited precision of Q-number for smaller values and its range restrictions, resulted in floating point being chosen for the system, requiring a refactor of the initial implementation that used Q-numbers.

B.4. Optimising harmonic indexing using a LUT

We are interested in creating a LUT for equation 3.14, shown below for convenience.

$$i = \min(\lfloor \log_2(\lfloor h_{max} \rfloor) \rfloor, K) - 1$$

We can reduce the memory consumption of the harmonic indexing LUT by 4, by considering the equation:

$$i = \lfloor \log_2(\lfloor \frac{h_{max}}{4} \rfloor) \rfloor + 1$$

This LUT can easily be constructed. Since we only 2^{b-1} harmonics within the buffer, we originally need a LUT of size 2^{b-1}. This technique allows for a LUT of size 2^{b-3}. For this thesis, we use $b = 9$, hence we originally need a LUT of size 256. For this technique, we only need a LUT of size 64, and extra indexing code. This technique is verified to yield correct results using Microsoft Excel.

The harmonic indexing equation then becomes:

$$i = \begin{cases} 0, \ h_{max} \leq 2 \\ 1, \ h_{max} \in (2,4] \\ \lfloor \log_2(\lfloor \frac{h_{max}}{4} \rfloor) \rfloor + 1, \ h_{max} > 4 \end{cases} \tag{B.1}$$

Note that larger memory reduction is possible, but would require more logic to handle the additional piece-wise functions added to i.

B.5. Alternative uses and signal chains for the designed components

The choice of processing order in the generator component for the design in this thesis is arbitrary, and depends on the designer. Order is significant, since this system is inherently non-linear, caused by waveshaping and ADSR volume modulation. Figure B.2 shows the signal chain as designed in this thesis, for reference. It should be noted that 3 oscillators is the minimum recommended for stereo width, but any odd number of oscillators greater than this can further add a greater stereo effect.
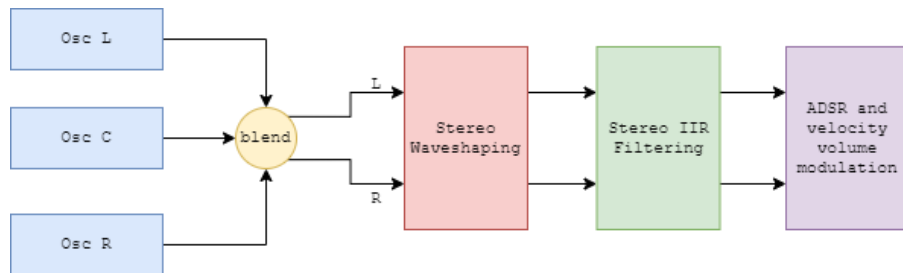
Figure B.2: The current signal chain, as designed in this thesis.

The placement of the waveshaping component can greatly influence the effect it has on the audio. From figure B.2, the left and right channels are waveshaped. This implies that the oscillators are coupled in the waveshaping process, i.e, a change in behaviour oscillators, such as by detuning our stereo width modification,

will greatly impact the results. As such, figure B.3 shows alternative placements for the waveshaping component. If the waveshaping is placed last, it will act as a form of distortion/saturation. This makes the results highly dependent on filter, volume and stereo settings. This effect is often desirable, but could require the introduction of more aliasing, as the performer could require a highly distorted signal.

Alternatively, the waveshaping can be placed after each oscillator, which would decouple any effects caused by stereo blending. This would lead to more predictable results, but comes at the cost of more waveshaping components, increasing computational requirements. This would not scale well in the addition of more oscillators in this system.
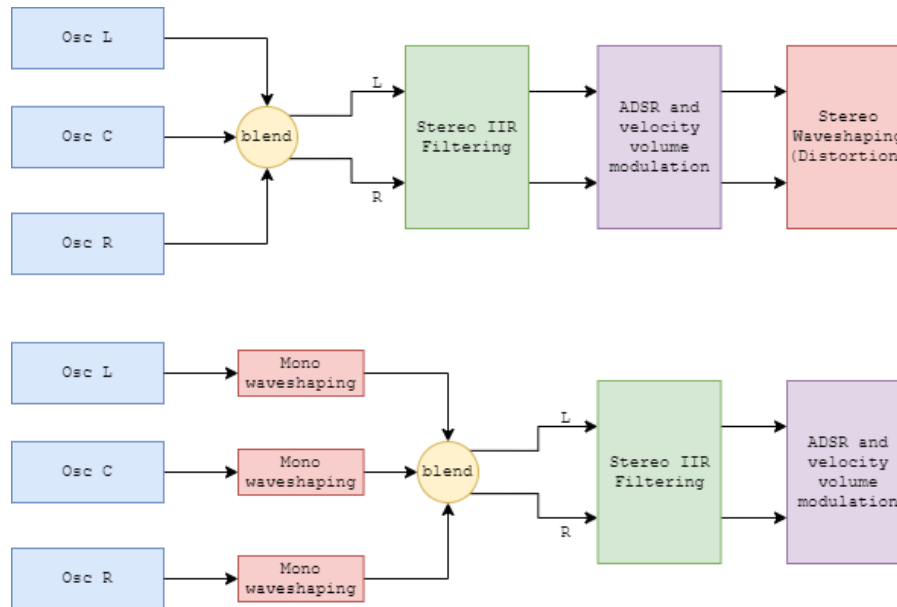
Figure B.3: Alternative signal chain placement for waveshaping.

Furthermore, hyperbolic tangent waveshaping can be used in the feedback loop of the filter, shown in figure B.4. This mimics the behaviour of analogue tube filters, which often becomes distorted for higher Q values. The non-linearity of tube filters can be reflected here, since it will add additional harmonics to the signal, which is often desired by performers that extensively utilises filters.
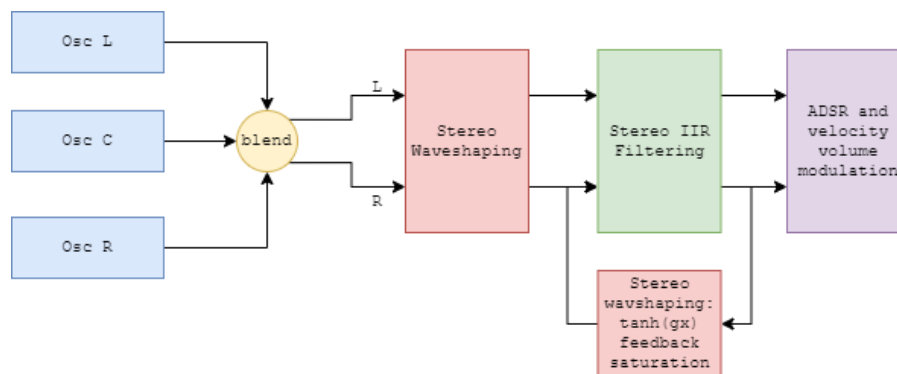
Figure B.4: Additional feedback saturation in the IIR filters.

Figures B.2, B.3 and B.4 can be combined in any way to also achieve another alternative signal chain. More filters, waveshapers and oscillators can be introduced to create a more complicated product, using the building blocks that were designed in this thesis.

Appendix C

Tables

Table C.1: Bilinear transform substitution [2]

S-domain	Z-domain
1	$(1 + 2z^{-1} + z^{-2})(1 - \cos(\omega_0))$
s	$(1 - z^{-2})\sin(\omega_0)$
s^2	$(1 - 2z^{-1} + z^{-2})(1 + \cos(\omega_0))$
$1 + s^2$	$2(1 - 2\cos(\omega_0)z^{-1} + z^{-2})$

Table C.2: MIDI note IDs and their frequencies [3]

MIDI (hexadecimal)	Note Name	Frequency (Hz)
0x0		8.18
0x1		8.66
0x2		9.18
0x3		9.72
0x4		10.3
0x5		10.91
0x6		11.56
0x7		12.25
0x8		12.98
0x9		13.75
0xA		14.57
0xB		15.43
0xC		16.35
0xD		17.32
0xE		18.35
0xF		19.45
0x10		20.6
0x11		21.83
0x12		23.12
0x13		24.5
0x14		25.96
0x15	A0	27.5
0x16	A#0/Bb0	29.14
0x17	B0	30.87
0x18	C1	32.7
0x19	C#1/Db1	34.65
0x1A	D1	36.71
0x1B	D#1/Eb1	38.89
0x1C	E1	41.2

0x1D	F1	43.65
0x1E	F#1/Gb1	46.25
0x1F	G1	49
0x20	G#1/Ab1	51.91
0x21	A1	55
0x22	A#1/Bb1	58.27
0x23	B1	61.74
0x24	C2	65.41
0x25	C#2/Db2	69.3
0x26	D2	73.42
0x27	D#2/Eb2	77.78
0x28	E2	82.41
0x29	F2	87.31
0x2A	F#2/Gb2	92.5
0x2B	G2	98
0x2C	G#2/Ab2	103.83
0x2D	A2	110
0x2E	A#2/Bb2	116.54
0x2F	B2	123.47
0x30	C3	130.81
0x31	C#3/Db3	138.59
0x32	D3	146.83
0x33	D#3/Eb3	155.56
0x34	E3	164.81
0x35	F3	174.61
0x36	F#3/Gb3	185
0x37	G3	196
0x38	G#3/Ab3	207.65
0x39	A3	220
0x3A	A#3/Bb3	233.08
0x3B	B3	246.94
0x3C	C4 (middle C)	261.63
0x3D	C#4/Db4	277.18
0x3E	D4	293.66
0x3F	D#4/Eb4	311.13
0x40	E4	329.63
0x41	F4	349.23
0x42	F#4/Gb4	369.99
0x43	G4	392
0x44	G#4/Ab4	415.3
0x45	A4 concert pitch	440
0x46	A#4/Bb4	466.16
0x47	B4	493.88
0x48	C5	523.25

0x49	C#5/Db5	554.37
0x4A	D5	587.33
0x4B	D#5/Eb5	622.25
0x4C	E5	659.26
0x4D	F5	698.46
0x4E	F#5/Gb5	739.99
0x4F	G5	783.99
0x50	G#5/Ab5	830.61
0x51	A5	880
0x52	A#5/Bb5	932.33
0x53	B5	987.77
0x54	C6	1046.5
0x55	C#6/Db6	1108.73
0x56	D6	1174.66
0x57	D#6/Eb6	1244.51
0x58	E6	1318.51
0x59	F6	1396.91
0x5A	F#6/Gb6	1479.98
0x5B	G6	1567.98
0x5C	G#6/Ab6	1661.22
0x5D	A6	1760
0x5E	A#6/Bb6	1864.66
0x5F	B6	1975.53
0x60	C7	2093
0x61	C#7/Db7	2217.46
0x62	D7	2349.32
0x63	D#7/Eb7	2489.02
0x64	E7	2637.02
0x65	F7	2793.83
0x66	F#7/Gb7	2959.96
0x67	G7	3135.96
0x68	G#7/Ab7	3322.44
0x69	A7	3520
0x6A	A#7/Bb7	3729.31
0x6B	B7	3951.07
0x6C	C8	4186.01
0x6D	C#8/Db8	4434.92
0x6E	D8	4698.64
0x6F	D#8/Eb8	4978.03
0x70	E8	5274.04
0x71	F8	5587.65
0x72	F#8/Gb8	5919.91
0x73	G8	6271.93
0x74	G#8/Ab8	6644.88

0x75	A8	7040
0x76	A#8/Bb8	7458.62
0x77	B8	7902.13
0x78	C9	8372.02
0x79	C#9/Db9	8869.84
0x7A	D9	9397.27
0x7B	D#9/Eb9	9956.06
0x7C	E9	10548.08
0x7D	F9	11175.3
0x7E	F#9/Gb9	11839.82
0x7F	G9	12543.85

Table C.3: ARM Cortex M4 and M7 FPU instruction set [4]

Operation	Assembler	Cycles
Absolute value	VABS.F32	1
Addition	VADD.F32	1
Compare	VCMP.F32	1
	VCMPE.F32	1
Convert	VCVT.F32	1
Divide	VDIV.F32	14
Load	VLDM.64	1+2*N
	VLDM.32	1+N
	VLDR.64	3
	VLDR.32	2
Move	VMOV	1
	VMOV	1
	VMOV	2
	VMRS	1
	VMSR	1
Multiply	VMUL.F32	1
	VMLA.F32	3
	VMLS.F32	3
	VNMLA.F32	3
	VNMLS.F32	3
Multiply (fused)	VFMA.F32	3
	VFMS.F32	3
	VFNMA.F32	3
	VFNMS.F32	3
Negate	VNEG.F32	1
	VNMUL.F32	1
Pop	VPOP.64	1+2*N
	VPOP.32	1+N
Push	VPUSH.64	1+2*N
	VPUSH.32	1+N
Square-root	VSQRT.F32	14
Store	VSTM.64	1+2*N
	VSTM.32	1+N
	VSTR.64	3
	VSTR.32	2
Subtract	VSUB.F32	1

Table C.4: Parameter settings for generated audio files from section 4.8.

Category	Parameter	Audio File				
		1	**2**	**3**	**4**	**5**
Volume	Attack [ms]	50	50	80	80	5
	Decay [ms]	50	200	200	200	100
	Sustain	0.8	0.5	0.75	0.75	0
	Release [ms]	60	10	80	80	
Filtering	Rel. start	3	1000	100	3	700
	Rel. end	500	100	10	1	7
	Attack [ms]	100	100	150	200	100
	Decay [ms]	100	100	200	200	100
	Sustain	1	1	1	1	1
	Release [ms]	10^6	10^6	10^6	10^6	10^6
	Q	0.7	2	0.7	3	2
	Type	LP12	LP24	LP24	HP24	LP24
Detune	Amount [cents]	10	0	1200	20	0
	Width	0.3		0.2	0.5	
	Volume	0.8		0.8	0.8	
Wavetable	Position	2	2	2.5	3.2	2.9
Vibrato	Intensity [cents]	15	0	10	0	0
	Frequency [Hz]	7		7		
Waveshaping	Type	tanh	sin	tanh	none	none
	Gain	1	2π	5		

Appendix D

Code listings

All code is publicly available at https://github.com/marcorad/skripsie.

```
1  float notes_freq_hz[] =
2  { 8.18f,8.66f,9.18f,9.72f,10.3f,10.91f,
3  11.56f,12.25f,12.98f,13.75f,14.57f,15.43f,
4  16.35f,17.32f,18.35f,19.45f,20.6f,21.83f,
5  23.12f,24.5f,25.96f,27.5f,29.14f,30.87f,32.7f,
6  34.65f,36.71f,38.89f,41.2f,43.65f,46.25f,49.0f,
7  51.91f,55.0f,58.27f,61.74f,65.41f,69.3f,73.42f,77.78f,82.41f,87.31f,92.5f,
8  98.0f,103.83f,110.0f,116.54f,123.47f,130.81f,138.59f,146.83f,155.56f,164.81f,
9  174.61f,185.f,196.0f,207.65f,220.0f,233.08f,246.94f,261.63f,277.18f,293.66f,
10 311.13f,329.63f,349.23f,369.99f,392.0f,415.3f,440.0f,466.16f,493.88f,523.25f,554.37f,
11 587.33f,622.25f,659.26f,698.46f,739.99f,783.99f,830.61f,880.0f,932.33f,987.77f,1046.5
      f,
12 1108.73f,1174.66f,1244.51f,1318.51f,1396.91f,1479.98f,1567.98f,1661.22f,1760.0f
      ,1864.66f,
13 1975.53f,2093.0f,2217.46f,2349.32f,2489.02f,2637.02f,2793.83f,2959.96f,3135.96f
      ,3322.44f,
14 3520.0f,3729.31f,3951.07f,4186.01f,4434.92f,4698.64f,4978.03f,5274.04f,5587.65f
      ,5919.91f,
15 6271.93f,6644.88f,7040.0f,7458.62f,7902.13f,8372.02f,8869.84f,9397.27f,9956.06f
      ,10548.08f,
16 11175.3f,11839.82f,12543.85f };
17
18 float cents_scaling_factor[] = {
19 1.00000000000000f,1.00057778950655f,1.00115591285382f,1.00173437023470,
20 1.00231316184217f,1.00289228786937f,1.00347174850950f,1.00405154395592,
21 1.00463167440205f,1.00521214004148f,1.00579294106785f,1.00637407767497,
22 1.00695555005672f,1.00753735840711f,1.00811950292026f,1.00870198379040,
23 1.00928480121187f,1.00986795537914f,1.01045144648676f,1.01103527472943,
24 1.01161944030192f,1.01220394339916f,1.01278878421615f,1.01337396294802,
25 1.01395947979003f,1.01454533493752f,1.01513152858597f,1.01571806093096,
26 1.01630493216819f,1.01689214249346f,1.01747969210269f,1.01806758119192,
27 1.01865580995729f,1.01924437859508f,1.01983328730164f,1.02042253627348,
28 1.02101212570719f,1.02160205579949f,1.02219232674721f,1.02278293874728,
29 1.02337389199677f,1.02396518669285f,1.02455682303280f,1.02514880121402,
30 1.02574112143402f,1.02633378389042f,1.02692678878098f,1.02752013630354,
31 1.02811382665607f,1.02870786003665f,1.02930223664349f,1.02989695667490,
32 1.03049202032930f,1.03108742780523f,1.03168317930136f,1.03227927501645,
33 1.03287571514939f,1.03347249989918f,1.03406962946493f,1.03466710404588,
34 1.03526492384138f,1.03586308905088f,1.03646159987396f,1.03706045651031,
35 1.03765965915975f,1.03825920802219f,1.03885910329766f,1.03945934518634,
36 1.04005993388848f,1.04066086960447f,1.04126215253481f,1.04186378288011,
37 1.04246576084112f,1.04306808661868f,1.04367076041375f,1.04427378242741,
38 1.04487715286087f,1.04548087191543f,1.04608493979253f,1.04668935669371,
39 1.04729412282063f,1.04789923837507f,1.04850470355893f,1.04911051857422,
40 1.04971668362307f,1.05032319890772f,1.05093006463054f,1.05153728099401,
41 1.05214484820072f,1.05275276645338f,1.05336103595484f,1.05396965690802,
```

```
42  1.05457862951601f ,1.0518795398198f ,1.05579763050924f ,1.05640765930119 ,
43  1.05701804056138f ,1.0576287449346f ,1.05823986130119f ,1.0585130118847 ,
44  1.05946309435930
45  };
46
47  float semitones_scaling_factor [] = {
48  1.00000000000000f ,1.05946309435930f ,1.12246204830937f ,1.18920711500272 ,
49  1.25992104989487f ,1.33483985417003f ,1.41421356237310f ,1.49830707687668 ,
50  1.58740105196820f ,1.68179283050743f ,1.78179743628068f ,1.88774862536339
51  };
52
53  float notes_digital_freq [128];
54
55  void init_notes_digital_freq_buffer () {
56    for (int i = 0; i < 128; i++) {
57      notes_digital_freq[i] = notes_freq_hz[i] / FS; //calc the buffer to allow for
      changes in FS
58    }
59  }
60
61  //only positive
62  float get_detune_factor_semitones_lut(int st) {
63    int i = abs(st);
64    int oct = i / 12;
65    i = i - oct * 12;
66    return ((float)(1<<oct)) * semitones_scaling_factor[i];
67  }
68
69  float get_detune_factor_cents_lut(float cents) {
70    float xc = fabs(cents);
71    float xs = floorf(xc * 0.01f); //amount of integer semitones
72    xc = ( xc - 100.0f * xs); //cents in the range [0,100)
73    float stf = get_detune_factor_semitones_lut((int)xs); //scaling from octaves
74    float cf = lut_lookup_no_wrap(cents_scaling_factor, xc); //scaling from cents,
      interpolated
75    float f = stf * cf; //cumulative scaling factor
76    return cents < 0.0f ? 1.0f / f : f; //scaling up or down
77  }
78
79  float get_detune_factor(float cents) {
80    return get_detune_factor_cents_lut(cents); // div by 100 to achieve 2^(1/1200)
81  }
82
83  float detune_cents(float orig_freq, float cents) {
84    return orig_freq * get_detune_factor(cents); // div by 100 to achieve 2^(1/1200)
85  }
```

Listing D.1: Efficient frequency scaling using the equal temperament tuning system

```
1  float square_sample(float pos, int k){
2      if (k % 2 == 0) return 0.0f; //only odd harmonics
3      float m = (float)(k);
4      return sinf(TWO_PI * m *  pos) / m;
5  }
6
```

```
7  float saw_sample(float pos, int k){
8      float sign = k % 2 == 0 ? 1.0f : -1.0f;
9      return -sinf(TWO_PI * (float)k *  pos) * sign / (float) k;
10 }
11
12 float tri_sample(float pos, int k){
13     if (k % 2 == 0) return 0.0f; //only odd harmonics
14     int i = (k - 1)/2; //account for different summation limit
15     float m = (float)(2*i + 1); //equals k
16     float sign = (i) % 2 == 0 ? 1.0f : -1.0f;
17     return sinf(TWO_PI * m *  pos) * sign / m / m;
18 }
19
20 float sin_sample(float pos, int k){
21     return sinf(TWO_PI * pos);
22 }
23
24 void load_wave(float buf[],  float (*wave)(float, int), int num_harmonics){
25     float max = -INFINITY;
26     float min = INFINITY;
27     float* vals = new float[LUT_SIZE];
28     for(int i = 0; i < LUT_SIZE; i++){
29         float pos = (float)i / (float) LUT_SIZE;
30         float val = 0.0f;
31         for(int k = 1; k <= num_harmonics; k++){
32             val += wave(pos, k);
33         }
34         *(vals+i) = val;
35         if (val < min) min = val;
36         if (val > max) max = val;
37     }
38     float offset = min + (max-min)/2.0f;
39     float s =2.0f/(max-min);
40
41      for(int i = 0; i < LUT_SIZE; i++){
42         //normalise to range [-1.0, 1.0]
43         float y = (vals[i] - offset) * s;
44         buf[i] = y;
45     }
46 }
47
48 void load_exp_decay(float exp[], uint16_t exp_size) {
49     float p = 0.8f; //fraction of final value K
50     float K = 1 / p; //final value, a[N] = pK
51     float N = (float)EXP_LUT_SIZE - 1.0f; //a[N] = 1 at final sample
52     float R = powf(1 - p, 1 / N); //base of exponent
53     for (int n = 0; n < exp_size; n++) {
54         float val = K * (1.0f - powf(R, (float)n)); //a[n]
55         exp[n] = val;
56     }
57 }
58
59 float basic_luts[4][8][LUT_SIZE];
60
```

```
61  void load_basic_luts() {
62      for (int i = 0; i < 8; i++) {
63          int n_harmonics = 1 << (i + 1); // 2 4 8 16 32 64 128 256
64          load_wave(basic_luts[SINE][i], sin_sample, n_harmonics); //exception for sin
    to save memory?
65          load_wave(basic_luts[TRIANGLE][i], tri_sample, n_harmonics);
66          load_wave(basic_luts[SAWTOOTH][i], saw_sample, n_harmonics);
67          load_wave(basic_luts[SQUARE][i], square_sample, n_harmonics);
68      }
69  }
70
71  float lut_exp[EXP_LUT_SIZE];
72
73  void init_basic_luts() {
74      load_basic_luts();
75      load_exp_decay(lut_exp, EXP_LUT_SIZE);
76      load_tanh();
77  }
```

Listing D.2: Generating the waveforms

```
1   #define TANH_LUT_SIZE_P1 (TANH_LUT_SIZE + 1)
2   #define tanh_grad0 9.0f //can be adjusted. This is the gradient at 0. This will
    determine what the edges of the LUT looks like
3   #define tanh_grad0_inv (1.0f / tanh_grad0)
4
5   float lut_tanh[TANH_LUT_SIZE_P1];
6
7   void load_tanh() {
8       float norm = 1.0/tanhf(tanh_grad0 * (1.0f - 0.5f)); //tanh at the max index, to
    ensure the lut begins and ends at +-1.0f
9       for (int i = 0; i < TANH_LUT_SIZE; i++)
10      {
11          float x = (float)i / (float)TANH_LUT_SIZE;
12          lut_tanh[i] = tanhf(tanh_grad0 * (x - 0.5f))* norm;
13      }
14      lut_tanh[TANH_LUT_SIZE] = 1.0f;
15  }
16
17  //saturate a number. returns a lut lookup of tanh(t)
18  inline float tanh_lookup(float t) {
19      float x = t * tanh_grad0_inv + 0.5f; //offset to be centered around 0.5 and
    cancel out gradient gain
20      //hard clip, to approximate tanh behaviour as x -> +-inf
21      if (x < 0.0f) return -1.0f;
22      if (x > 1.0f) return 1.0f;
23      //otherwise return approximated tanh(in)
24      return lut_lookup_no_wrap(lut_tanh, x * ((float)TANH_LUT_SIZE)); //makes sure no
    wrapping occurs, recall that that the last value is 1
25  }
26
27  inline float waveshape_calc_tanh_norm(float gain) {
28      return 1.0f / tanh_lookup(gain);
29  }
30
```

```
31  inline float waveshape_calc_sine_norm(float gain) {
32      if (gain >= 1.0f) return 1.0f;
33      return 1.0f / sin_lookup(0.25f * (gain));
34  }
35
36  //norm must be 1/tanh(gain) = 1.0f / tanh_lookup(gain)
37  float waveshape_tanh(float x, float gain) {
38      return tanh_lookup(gain * x);
39  }
40
41  //no modification
42  float waveshape_none(float x, float gain) {
43      return x;
44  }
45
46  float waveshape_hard_clipper(float x, float gain) {
47      return clamp(gain * x, -1.0f, 1.0f);
48  }
49
50  float waveshape_sine(float x, float gain) {
51      float t = fract(gain * x * PI_INV * 0.5f);
52      return sin_lookup(t);
53  }
```

Listing D.3: Waveshaping LUT functions

```
1  uint8_t harmonic_indices[] = {1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,
2  5,5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,7}; //
       floor(log(n/4)), where n is index
3
4  //configure in samples/sec and determin correct number of harmonics to avoid aliasing
5  inline void wt_config_digital_freq(wavetable* wt, float freq) {
6    wt->stride = freq * (float)LUT_SIZE; //set strides, with no FM applied
7    wt->base_stride = wt->stride;
8    uint16_t harmonics = (uint16_t)(0.5f / freq);
9    harmonics = harmonics > (LUT_SIZE >> 1) ? (LUT_SIZE >> 1) : harmonics; //clamp at
       max allowed by buffer
10   uint8_t harmonic_index = harmonics < 4 ? 0 : harmonic_indices[(harmonics >> 2) -
       1]; //divide harmonic index by 4, see excel, minus 1 for shifting excel index to
       0
11   wt->harmonic_index = harmonic_index;
12 }
```

Listing D.4: Configuring wavetable frequency

```
1  //ALL FILTERS HAVE f0 IN DIGITAL FREQUENCY
2
3  inline void iir_calc_lp24_coeff(IIR_coeff* filter, float f0, float q) {
4    float cosw = cos_lookup(f0);
5    float sinw = sin_lookup(f0);
6    float twoq = 2.0f * q;
7    //denominator
8    float a0_recip = 1.0f / (twoq + sinw); //devision is unavoidable here
9    float a1 = 2.0 * twoq * cosw; //negated version of what is in table, to account for
       d1 = -a1/a0
```

```c
10    float a2 = sinw - twoq; //negated version of what is in table, to account for d2 =
        -a2/a0
11    float b0, b1, b2;
12    b1 = twoq * (1.0f - cosw);
13    b0 = b1 * 0.5f;
14    b2 = b0;
15    float n0 = b0 * a0_recip;
16    float n1 = b1 * a0_recip;
17    float n2 = b2 * a0_recip;
18    float d1 = a1 * a0_recip;
19    float d2 = a2 * a0_recip;
20    filter->n0 = n0;
21    filter->n1 = n1;
22    filter->n2 = n2;
23    filter->d1 = d1;
24    filter->d2 = d2;
25 }
26
27 //HP calcs, f0 in samples/sec
28 inline void iir_calc_hp24_coeff(IIR_coeff* filter, float f0, float q) {
29    float cosw = cos_lookup(f0);
30    float sinw = sin_lookup(f0);
31    float twoq = 2.0f * q;
32    //denominator
33    float a0_recip = 1.0f / (twoq + sinw); //devision is unavoidable here
34    float a1 = 2.0 * twoq * cosw; //negated version of what is in table, to account for
        d1 = -a1/a0
35    float a2 = sinw - twoq; //negated version of what is in table, to account for d2 =
        -a2/a0
36    float b0, b1, b2;
37    b1 = -twoq * (1.0f + cosw);
38    b0 = b1 * -0.5f;
39    b2 = b0;
40    float n0 = b0 * a0_recip;
41    float n1 = b1 * a0_recip;
42    float n2 = b2 * a0_recip;
43    float d1 = a1 * a0_recip;
44    float d2 = a2 * a0_recip;
45    filter->n0 = n0;
46    filter->n1 = n1;
47    filter->n2 = n2;
48    filter->d1 = d1;
49    filter->d2 = d2;
50 }
51
52 //BP calcs, f0 in samples/sec
53 inline void iir_calc_bp12_coeff(IIR_coeff* filter, float f0, float q) {
54    float cosw = cos_lookup(f0);
55    float sinw = sin_lookup(f0);
56    float twoq = 2.0f * q;
57    //denominator
58    float a0_recip = 1.0f / (twoq + sinw); //devision is unavoidable here
59    float a1 = 2.0 * twoq * cosw; //negated version of what is in table, to account for
        d1 = -a1/a0
```

```
60    float a2 = sinw - twoq; //negated version of what is in table, to account for d2 =
         -a2/a0
61    float b0, b1, b2;
62    b1 = 0.0f;
63    b0 = q * sinw;
64    b2 = -b0;
65    float n0 = b0 * a0_recip;
66    float n1 = b1 * a0_recip;
67    float n2 = b2 * a0_recip;
68    float d1 = a1 * a0_recip;
69    float d2 = a2 * a0_recip;
70    filter->n0 = n0;
71    filter->n1 = n1;
72    filter->n2 = n2;
73    filter->d1 = d1;
74    filter->d2 = d2;
75 }
76
77 //LP calcs, f0 in samples/sec
78 inline void iir_calc_lp12_coeff(IIR_coeff* filter, float f0, float q) {
79    float cosw = cos_lookup(f0);
80    float sinw = sin_lookup(f0);
81    float one_minus_cosw = 1.0f - cosw;
82    //denominator
83    float a0_recip = 1.0f / (one_minus_cosw + sinw); //devision is unavoidable here
84    float a1 = 2.0f * one_minus_cosw; //negated version of what is in table, to account
         for d1 = -a1/a0
85    float a2 = one_minus_cosw - sinw; //negated version of what is in table, to account
         for d2 = -a2/a0
86    float b0, b1, b2;
87    b0 = one_minus_cosw;
88    b2 = b0;
89    b1 = 2.0f * b0;
90    float n0 = b0 * a0_recip;
91    float n1 = b1 * a0_recip;
92    float n2 = b2 * a0_recip;
93    float d1 = -a1 * a0_recip; //NEGATE!!
94    float d2 = -a2 * a0_recip;
95    filter->n0 = n0;
96    filter->n1 = n1;
97    filter->n2 = n2;
98    filter->d1 = d1;
99    filter->d2 = d2;
100 }
101
102 //HP calcs, f0 in samples/sec
103 inline void iir_calc_hp12_coeff(IIR_coeff* filter, float f0, float q) {
104    float cosw = cos_lookup(f0);
105    float sinw = sin_lookup(f0);
106    float one_minus_cosw = 1.0f - cosw;
107    //denominator
108    float a0_recip = 1.0f / (one_minus_cosw + sinw); //devision is unavoidable here
109    float a1 = 2.0 * one_minus_cosw; //negated version of what is in table, to account
         for d1 = -a1/a0
```

```
110    float a2 = one_minus_cosw - sinw; //negated version of what is in table, to account
           for d2 = -a2/a0
111    float b0, b1, b2;
112    b0 = sinw;
113    b2 = -b0;
114    b1 = 0.0f;
115    float n0 = b0 * a0_recip;
116    float n1 = b1 * a0_recip;
117    float n2 = b2 * a0_recip;
118    float d1 = -a1 * a0_recip;
119    float d2 = -a2 * a0_recip;
120    filter->n0 = n0;
121    filter->n1 = n1;
122    filter->n2 = n2;
123    filter->d1 = d1;
124    filter->d2 = d2;
125 }
126
127 inline void iir_calc_no_coeff(IIR_coeff* filter, float f0, float q) {
128    *filter = { 0.0f,0.0f,0.0f,0.0f,0.0f };
129    filter->n0 = 1.0f;
130 }
```

Listing D.5: Calculating the coefficients for the filters

```
1 inline float iir_filter_sample(IIR_coeff* coeff, IIR_prev_values* prev, float x) {
2    float y = coeff->n0 * x;
3    y += coeff->n1 * prev->xm1; //MULAC ops
4    y += coeff->n2 * prev->xm2;
5    y += coeff->d1 * prev->ym1;
6    y += coeff->d2 * prev->ym2;
7
8    //shift prev values
9    prev->ym2 = prev->ym1;
10    prev->ym1 = y;
11    prev->xm2 = prev->xm1;
12    prev->xm1 = x;
13    return prev->ym1;
14 }
```

Listing D.6: Filtering a signal

```
1 //approximates cos(2*pi*f) for f [0,0.75)
2 inline float cos_lookup(float f) {
3     return lut_lookup(basic_luts[SINE][0] + LUT_SIZE / 4, LUT_SIZE, f * (float)
    LUT_SIZE);
4 }
5
6 //approximates sin(2*pi*f) for f [0,1)
7 inline float sin_lookup(float f) {
8     return lut_lookup(basic_luts[SINE][0], LUT_SIZE, f * (float)LUT_SIZE);
9 }
```

Listing D.7: Trigonometric lookup functions

```
1 //MUST CHECK BEFOREHAND IF STATE IS NOT_PLAYING
```

```
2  inline float adsr_sample(ADSR* adsr, float params[]) {
3    if (adsr->state == SUSTAIN) { //expecting notes to spend most time in sustain
4      return params[SUSTAIN];
5    }
6
7    if ((adsr->phase < (float)EXP_LUT_SIZE - 1.0f)) //next phase from buffer wrap
8    {
9      float lookup = lut_lookup_no_wrap(lut_exp, adsr->phase);
10     adsr->prev_sample = adsr->scale * lookup + adsr->offset;
11   }
12   else {
13     adsr->state += 1;
14     adsr->phase = 0.0f;
15     if (adsr->state == DECAY) {
16       adsr->scale = params[SUSTAIN] - adsr->prev_sample;
17       adsr->offset = adsr->prev_sample;
18     }
19     else if (adsr->state == SUSTAIN) {
20       adsr->prev_sample = params[SUSTAIN];
21       return params[SUSTAIN];
22     }
23   }
24
25   if (adsr->state != NOT_PLAYING) {
26     adsr->phase += params[adsr->state];
27     return adsr->prev_sample;
28   }
29   else {
30     adsr->prev_sample = 0.0f;
31   }
32
33   return adsr->prev_sample;
34 }
35
36
37 //ADSR times in seconds
38 inline void adsr_config(float params[], float attack, float decay, float sustain,
     float release) {
39   float min = 1.0f / 0.001f / FS * (float)EXP_LUT_SIZE;
40   //strides
41   params[ATTACK] = attack != 0.0f ? (1.0f / attack / FS * (float)EXP_LUT_SIZE) : min;
       //minimum 1 ms
42   params[DECAY] = decay != 0.0f ? (1.0f / decay / FS * (float)EXP_LUT_SIZE) : min;
43   params[RELEASE] = release != 0.0f ? (1.0f / release / FS * (float)EXP_LUT_SIZE) :
     min;
44   //sustain level
45   params[SUSTAIN] = sustain;
46 }
47
48
49 //reset ADSR to trigger on a note
50 inline void adsr_trigger_on(ADSR* adsr) {
51   adsr->state = ATTACK;
52   adsr->phase = 0.0f;
```

```
53   adsr->scale = 1.0f - adsr->prev_sample; //attack from previous sample (for
       retrigger purposes)
54   adsr->offset = adsr->state != NOT_PLAYING ? adsr->prev_sample : 0.0f;
55 }
56
57 //reset ADSR to trigger into release
58 inline void adsr_trigger_off(ADSR* adsr) {
59   adsr->state = RELEASE;
60   adsr->phase = 0.0f;
61   adsr->scale = -adsr->prev_sample; //will always decay from previous sample
62   adsr->offset = adsr->prev_sample;
63 }
```

Listing D.8: Sampling from an ADSR envelope state-machine

```
1 inline void gen_vibrato(generator* g, gen_config* gc) {
2   float osc = (wt_sample_no_interpolation(&g->wt_vibrato, 0));
3   float ampl = (gc->vibrato_factor - 1.0f) * g->base_freq;
4   float vib = osc * ampl;
5   wt_apply_fm(&g->wt_left, vib);
6   wt_apply_fm(&g->wt_right, vib);
7   wt_apply_fm(&g->wt_center, vib);
8 }
```

Listing D.9: Applying vibrato FM

```
1 inline float gen_waveshape_sample(IIR_prev_values* pv, gen_config* gc, float x) {
2   float ylp = iir_filter_sample(&gc->filter_sat_AA, pv, x);
3   float yws = (*gc->waveshape)(ylp, gc->gain);
4   return (x - ylp) + yws;
5 }
6
7 //sample the voice
8 //CHECK IF THE VOLUME ENVELOPE IS DONE AND RESET THE FILTERS AFTERWARDS
9 inline void gen_sample(generator* g, gen_config* gc, float* buf_L, float* buf_R) {
10   //apply vibrato using FM
11   gen_vibrato(g, gc);
12   //sample L, R, C channels
13   float sc = wt_sample(&g->wt_center, gc);
14   float sl = wt_sample(&g->wt_left, gc);
15   float sr = wt_sample(&g->wt_right, gc);
16   //blend detuned samples
17   float width = 1.0f - gc->detune_width; //invert to get volume in other channel
18   //detune_width of 1 seperates sr and sl into L and R channels
19   //detune_width of 0 is mono
20   float L = sc + gc->detune_volume * (sl + width * sr);
21   float R = sc + gc->detune_volume * (sr + width * sl);
22   //find f0 using envelope
23   float f0 = g->filter_freq_start + adsr_sample(&g->envelope_filter_cutoff, gc->
       filt_adsr_params) * g->filter_envelope_amplitude;
24   f0 = clamp(f0, DIGITAL_FREQ_20HZ, DIGITAL_FREQ_20KHZ); //limit to audible range (
       otherwise there are clicks)
25   //calculate filter coefficients
26   (*(gc->filter_coeff_func))(&g->filter_coeff, f0, gc->filter_Q);
27   //waveshape L and R
28   L = gen_waveshape_sample(&g->filter_sat_pv_L, gc, L);
```

```
29    R = gen_waveshape_sample(&g->filter_sat_pv_R, gc, R);
30    //apply volume and envelope
31    //this is done before filtering so that we can partially mitigate the transient
         response, since filtering is linear
32    float volume = adsr_sample(&g->envelope_volume, gc->vol_adsr_params) * g->velocity;
33    L = L * volume;
34    R = R * volume;
35    //filter channels
36    L = iir_filter_sample(&g->filter_coeff, &g->filter_left_pv, L);
37    R = iir_filter_sample(&g->filter_coeff, &g->filter_right_pv, R);
38    //output
39    *buf_L = L;
40    *buf_R = R;
41 }
```

Listing D.10: Sampling from a generator

```
1  generator* note_played_hash_table[128]; //keep track of which note is played, for O
       (1) time on on-off triggers.
2
3  struct gen_manager {
4    generator generators[NUM_GENERATORS];
5    generator* available[NUM_GENERATORS];
6    int8_t available_head = NUM_GENERATORS;
7    generator* in_use[NUM_GENERATORS];
8    uint8_t in_use_head = 0;
9  };
10
11 //DO BEFORE PLAYING
12 inline void gm_init(gen_manager* gm) {
13   //make all available
14   for (int i = 0; i < NUM_GENERATORS; i++)
15   {
16     gm->available[i] = &gm->generators[i];
17   }
18   gm->available_head = NUM_GENERATORS;
19   gm->in_use_head = 0;
20   //none are in use
21   for (int i = 0; i < NUM_GENERATORS; i++)
22   {
23     gm->in_use[i] = nullptr;
24   }
25   //none are playing notes
26   for (int i = 0; i < 128; i++) {
27     note_played_hash_table[i] = nullptr;
28   }
29 }
30
31 inline void gm_add_to_in_use(gen_manager* gm, generator* g) {
32   gm->in_use[gm->in_use_head] = g;
33   gm->in_use_head++;
34 }
35
36 inline generator* gm_pop_off_back_from_in_use(gen_manager* gm) {
37   generator* g = gm->in_use[0];
```

```cpp
38    //shift queue
39    for (int i = 0; i < gm->in_use_head; i++)
40    {
41      gm->in_use[i] = gm->in_use[i + 1];
42    }
43    gm->in_use_head--;
44    return g;
45  }
46
47  inline void gm_add_to_available(gen_manager* gm, generator* g) {
48    gm->available[gm->available_head] = g;
49    gm->available_head++;
50  }
51
52  inline generator* gm_pop_off_front_from_available(gen_manager* gm) {
53    gm->available_head--;
54    return gm->available[gm->available_head];
55  }
56
57  inline void gm_set_gen_playing_note(generator* g, uint8_t midi_note) {
58    note_played_hash_table[midi_note] = g;
59  }
60
61  //get an available or gen, or the oldest playing gen if none are available
62  inline generator* gm_get_gen(gen_manager* gm) {
63    generator* g;
64    if (!gm->available_head == 0) {
65      //pop off front of available
66      g = gm_pop_off_front_from_available(gm);
67    }
68    else {//none are available
69      g = gm_pop_off_back_from_in_use(gm);
70      gm_set_gen_playing_note(nullptr, g->midi_note); //make sure that the retrigger of
        a note-off will not affect the new note
71    }
72
73    gm_add_to_in_use(gm, g);
74
75    return g;
76  }
77
78  //add all non-playing gens to the available list in O(N) time.
79  inline void gm_make_not_playing_available(gen_manager* gm) {
80    int count = 0;
81    for (int i = 0; i < gm->in_use_head; i++)
82    {
83      generator* g = gm->in_use[i];
84      if (!gen_is_playing(g)) {
85        count++;
86        gm_add_to_available(gm, g);
87      }
88      else {
89        gm->in_use[i - count] = g;
90      }
```

```
91      }
92      gm->in_use_head -= count;
93  }
94
95  inline generator* gm_get_gen_playing_note(uint8_t midi_note) {
96      return note_played_hash_table[midi_note];
97  }
98
99  float L_temp[PLAYBACK_BUFFER_SIZE];
100 float R_temp[PLAYBACK_BUFFER_SIZE];
101 inline void gm_write_n_samples(gen_manager* gm, gen_config* gc, float bufL[], float
        bufR[], uint32_t N) {
102     for (int i = 0; i < N; i++) //init to zero for addition later
103     {
104         bufL[i] = 0.0f;
105         bufR[i] = 0.0f;
106     }
107     for (int i = 0; i < gm->in_use_head; i++) //for all active voices
108     {
109         gen_accumulate_n_samples(gm->in_use[i], gc, bufL, bufR, N, 0.125f); //add to
            buffer, then mulac with 1/8th for headroom
110     }
111     for (int i = 0; i < N; i++) //clamp to +- 1
112     {
113         float L = bufL[i], R = bufR[i];
114         bufL[i] = clamp(L, -1.0f, 1.0f);
115         bufR[i] = clamp(R, -1.0f, 1.0f);
116     }
117     //make generators that finished decay phase available
118     gm_make_not_playing_available(gm);
119 }
120
121 //trigger on using the MIDI note frequency
122 inline void gm_trigger_note_on(gen_manager* gm, gen_config* gc, uint8_t note, uint8_t
        vel){
123     generator* g = gm_get_gen_playing_note(note);
124     if (g == nullptr) g = gm_get_gen(gm); //prevents retriggers of notes before a note
        off (faulty MIDI)
125     gen_freq(g, gc, notes_digital_freq[note], vel);
126     g->midi_note = note;
127     gm_set_gen_playing_note(g, note);
128     gen_note_on(g);
129 }
130
131 //does the same as above, but with an arbitrary digital frequency
132 inline void gm_trigger_note_on_freq(gen_manager* gm, gen_config* gc, uint8_t note,
        uint8_t vel, float freq) {
133     generator* g = gm_get_gen_playing_note(note);
134     if (g == nullptr) g = gm_get_gen(gm); //prevents any weirdness in retriggers of
        notes before a note off
135     gen_freq(g, gc, freq, vel);
136     g->midi_note = note;
137     gm_set_gen_playing_note(g, note);
138     gen_note_on(g);
```

```
139 }
140
141 inline void gm_trigger_note_off(gen_manager* gm, uint8_t note) {
142   generator* g = gm_get_gen_playing_note(note);
143   gm_set_gen_playing_note(nullptr, note); //not playing the note anymore
144   if (g != nullptr) gen_note_off(g); //prevents any weirdness in note off triggers if
          it's not actually on
145 }
146
147 //apply vibrato config to generators
148 inline void gm_apply_vibrato_config(gen_manager* gm, gen_config* gc) {
149   for (int i = 0; i < NUM_GENERATORS; i++) {
150     gen_apply_vibrato_config(&gm->generators[i], gc);
151   }
152 }
```

Listing D.11: Generator manager functions

Appendix E

Test code and Audio Logs

All code is publicly available at https://github.com/marcorad/skripsie.

```cpp
using namespace std;

//white noise generator
void generate_wn(float* buf, int N) {
  unsigned seed = 42; //LOL
  std::default_random_engine generator(seed);
  auto dist = std::normal_distribution<float>(0.0f, 1.0f);

  for (int i = 0; i < N; i++)
  {
    buf[i] = dist(generator);
  }
}

//chirp generator (in digital frequency)
void generate_linear_chirp_frequencies(float* buf, int N, float start, float end) {
  for (int i = 0; i < N; i++)
  {
    float x = (float)i / (float)N;
    float f = start + (end-start) * x;
    buf[i] = f;
  }
}

void add_wn(float* wn, int N, float u, float s) {
  srand(time(NULL));
  unsigned seed = rand();
  std::default_random_engine generator(seed);
  auto dist = std::normal_distribution<float>(u, s);

  for (int i = 0; i < N; i++)
  {
    wn[i] += dist(generator);
  }
}

void test_adsr(float* buf, int N) {
  ADSR adsr;
  int off = 0;
  int k = N / 8;
  float t = k / FS;
  float params[4];
  adsr_config(params, t, t, 0.5f, t);

  for (int i = 0; i < k / 3; i++)
  {
```

```
47      buf[i] = 0.0f;
48    }
49    off += k / 3;
50    adsr_trigger_on(&adsr);
51
52    for (int i = off; i < off + k + k/2; i++)
53    {
54      buf[i] = adsr_sample(&adsr, params);
55    }
56    adsr_trigger_on(&adsr);
57    off += k + k / 2;
58    for (int i = off; i < off + 3 * k; i++)
59    {
60      buf[i] = adsr_sample(&adsr, params);
61    }
62    off += 3* k;
63    adsr_trigger_off(&adsr);
64    for (int i = off; i < off + N - off; i++)
65    {
66      buf[i] = adsr_sample(&adsr, params);
67    }
68  }
69
70  void test_wavetable(float* buf, int N, float pos) {
71    //wavetables at different pos and freq
72    float f0 = notes_digital_freq[A0];
73    int off = 0;
74    int n = N / 8;
75    gen_config gc;
76    gc.wt_pos = pos;
77
78    wavetable wt;
79
80    for (float f = f0; f <= notes_digital_freq[A8]; f *= 2.0f) //sweep octaves
81    {
82      wt.phase = 0;
83      wt_config_digital_freq(&wt, f);
84      for (int i = off;  i < off + n; i++)
85      {
86        buf[i] = wt_sample(&wt, &gc);
87      }
88      off += n;
89    }
90  }
91
92  void test_vibrato(float* buf, int N) {
93    float f0 = 10000.0f/FS;
94    float cents = 500.0f;
95    generator g;
96    gen_config gc;
97    gen_config_default(&gc);
98
99    gen_config_vibrato(&gc, cents, 2.0f / FS);
100
```

```
101    gen_apply_vibrato_config(&g, &gc);
102    gen_freq(&g, &gc, f0, 127);
103    gen_note_on(&g);
104
105    for (int i = 0; i < N; i++)
106    {
107      float temp;
108      gen_sample(&g, &gc, buf + i, &temp);
109    }
110  }
111
112  void test_filter(float* wn, float* out, float* adsr_out, int N, float a, float d,
          float s, float r, float freqstart, float freqend, float q, void (*
          filter_coeff_func)(IIR_coeff*, float, float)) {
113    ADSR adsr;
114    IIR_prev_values pv = {0,0,0,0};
115    IIR_coeff coeff;
116    float params[4];
117    adsr_config(params, a, d, s, r);
118    adsr_trigger_on(&adsr);
119
120    for (int i = 0; i < N/2; i++)
121    {
122      float f0 = freqstart + adsr_sample(&adsr, params) * (freqend - freqstart);
123      (*filter_coeff_func)(&coeff, f0, q);
124      out[i] = iir_filter_sample(&coeff, &pv, wn[i]);
125      adsr_out[i] = f0;
126    }
127
128    adsr_trigger_off(&adsr);
129
130    for (int i = N/2; i < N; i++)
131    {
132      float f0 = freqstart + adsr_sample(&adsr, params) * (freqend - freqstart);
133      (*filter_coeff_func)(&coeff, f0, q);
134      out[i] = iir_filter_sample(&coeff, &pv, wn[i]);
135      adsr_out[i] = f0;
136    }
137  }
138
139  void test_waveshape(float* buf_tanh, float* buf_sin, int N, float* freq, float g_tanh
          , float g_sin, bool use_aa = true) {
140    generator g;
141    gen_config gc;
142    gen_config_default(&gc);
143    gen_apply_vibrato_config(&g, &gc);
144    gen_note_on(&g);
145    gen_config_tanh_saturator(&gc, g_tanh);
146    gen_config_filter_envelope(&gc, 0.1f, 0.1f, 1.0f, 0.1f, 512.0f, 512.0f, 0.7071f);
147    gen_config_wavetables(&gc, 0.0f, 0.0f, 0.0f, 0.0f);
148    if(!use_aa)
149      iir_calc_lp12_coeff(&gc.filter_sat_AA, DIGITAL_FREQ_20KHZ, 1.0f);
150    for (int i = 0; i < N; i++)
151    {
```

```
152       float temp;
153       gen_freq(&g, &gc, freq[i], 127);
154       gen_sample(&g, &gc, buf_tanh + i, &temp);
155     }
156
157     gen_config_sine_saturator(&gc, g_sin);
158     gen_reset_AA_pv(&g); //reset pv from previous step
159     if (!use_aa)
160       iir_calc_lp12_coeff(&gc.filter_sat_AA, DIGITAL_FREQ_20KHZ, 1.0f);
161     for (int i = 0; i < N; i++)
162     {
163       float temp;
164       gen_freq(&g, &gc, freq[i], 127);
165       gen_sample(&g, &gc, buf_sin + i, &temp);
166     }
167 }
168
169 void test_wavetable_sweep(float* buf, int N, float* freq, float pos) {
170     wavetable wt;
171     gen_config gc;
172
173     gc.wt_pos = pos;
174
175     for (int i = 0; i < N; i++)
176     {
177       wt_config_digital_freq(&wt, freq[i]);
178       buf[i] = wt_sample(&wt, &gc);
179     }
180 }
181
182 void test_stereo_width(float* bufL, float* bufR, int N, float width) {
183     float f0 = 100.0f / FS;
184     float cents = 1200.0f;
185     generator g;
186     gen_config gc;
187     gen_config_default(&gc);
188     gen_apply_vibrato_config(&g, &gc);
189     gen_config_wavetables(&gc, 0.0f, cents, 1.0f, width);
190     gen_freq(&g, &gc, f0, 127);
191     gen_note_on(&g);
192     gen_write_n_samples(&g, &gc, bufL, bufR, N);
193 }
194
195 void test_gm_overload(float buf[], int N) {
196     gen_manager gm;
197     gen_config gc;
198     gen_config_default(&gc);
199     gm_init(&gm);
200     gc.filter_coeff_func = &iir_calc_no_coeff;
201     gen_config_wavetables(&gc, 0.0f, 0.0f, 0.0f, 0.0f);
202     float* temp = new float[N];
203
204     float freq[16];
205     for (int i = 0; i < 8; i++)
```

```
206    {
207      freq[i] = (1000.0f + i * 2000.0f) / FS; //1k, 3k, ..., 15k
208      freq[i+8] = (2000.0f + i * 2000.0f) / FS; //2k, 4k, ..., 16k
209    }
210
211    int L = N / 16;
212    int end = L * 16;
213    int note = 0;
214    for (int off = 0; off < end; off += L)
215    {
216      gm_trigger_note_on_freq(&gm, &gc, note, 127, freq[note]);
217      gm_write_n_samples(&gm, &gc, buf + off, temp + off, L);
218      note++;
219    }
220
221    //fill remaining with 0
222    for (int i = 0; i < N-end; i++)
223    {
224      buf[end + i] = 0.0f;
225    }
226 }
227
228
229
230 void test_gm_operation(float buf[], int N) {
231    gen_manager gm;
232    gen_config gc;
233    gen_config_default(&gc);
234    gm_init(&gm);
235    gc.filter_coeff_func = &iir_calc_no_coeff;
236    gen_config_wavetables(&gc, 0.0f, 0.0f, 0.0f, 0.0f);
237
238    float* temp = new float[N];
239
240    float freq[8];
241    for (int i = 0; i < 8; i++)
242    {
243      freq[i] = (500.0f + i * 2500.0f) / FS; //0k5, 3k, 5k5, ..., 18k
244    }
245
246    int L = N / 16;
247    int end = L * 16;
248    int note = 0;
249    int count = 0;
250    //trigger all on, then trigger all off by LIFO
251    for (int off = 0; off < end; off += L)
252    {
253      if (count < 8)
254      {
255        gm_trigger_note_on_freq(&gm, &gc, note, 127, freq[note]);
256        note++;
257      }
258      else {
259        gm_trigger_note_off(&gm, note);
```

```
260       note--;
261     }
262     gm_write_n_samples(&gm, &gc, buf + off, temp + off, L);
263     count++;
264   }
265
266   //fill remaining with 0
267   for (int i = 0; i < N - end; i++)
268   {
269     buf[end + i] = 0.0f;
270   }
271 }
272
273 void test_overall_system(float bufL[], float bufR[], int N) {
274   gen_manager gm;
275   gen_config gc;
276   gen_config_default(&gc);
277   gm_init(&gm);
278   gen_config_wavetables(&gc, 2.5f, 100.0f, 0.25f, 1.0f);
279   gen_config_filter(&gc, &iir_calc_lp24_coeff);
280   gen_config_filter_envelope(&gc, 0.3f, 0.1f, 0.5f, 0.2f, 1.0f, 50.0f, 7.0f);
281   gen_config_tanh_saturator(&gc, 2.0f);
282   gen_config_vibrato(&gc, 50.0f, 5.0f / FS);
283   gm_apply_vibrato_config(&gm, &gc);
284   gen_config_volume_envelope(&gc, 0.01f, 0.2f, 0.5f, 0.2f);
285
286   float f1 = 500.0f / FS, f2 = 10000.0f / FS ;
287
288   int n = N / 16;
289   int n1 = 4 * n;
290   int n2 = 8 * n;
291   int n3 = 12 * n;
292   int n4 = 16 * n;
293
294   gm_trigger_note_on_freq(&gm, &gc, 0, 127, f1);
295   int begin = 0;
296   int size = n1;
297   gm_write_n_samples(&gm, &gc, bufL + begin, bufR + begin, size);
298
299   begin = n1;
300   size = n1;
301   gm_trigger_note_on_freq(&gm, &gc, 1, 127, f2);
302   gm_write_n_samples(&gm, &gc, bufL + begin, bufR + begin, size);
303
304   begin = n2;
305   size = n3 - n2;
306   gm_trigger_note_off(&gm, 0);
307   gm_write_n_samples(&gm, &gc, bufL + begin, bufR + begin, size);
308
309   begin = n3;
310   size = n4 - n3;
311   gm_trigger_note_off(&gm, 1);
312   gm_write_n_samples(&gm, &gc, bufL + begin, bufR + begin, size);
313
```

```
314    for (int i = n4; i < N; i++)
315    {
316      bufL[i] = 0.0f;
317      bufR[i] = 0.0f;
318    }
319 }
320
321 void test_freq_scaling() {
322    cout << "Detune factor 125.57: " << get_detune_factor_cents_lut(125.57f) << endl;
323 }
324
325 void print_to_file(float arr[], int size, const string& name) {
326    ofstream f;
327    f.open(name);
328    f.precision(10);
329    for (int i = 0; i < size; i++) {
330      f << arr[i] << endl;
331    }
332    f.close();
333 }
334
335
336 void print_iir_coeff(IIR_coeff& f) {
337    cout << "[" << f.n0 << ", " << f.n1 << ", " << f.n2 << "], [ 1, -" << f.d1 << ", -"
         << f.d2 << "]" << endl;
338 }
339
340 void print_basic_waveforms(){
341    string names[] = { "sine", "triangle", "sawtooth", "square" };
342    for (int i = 0; i < 4; i++) {
343      for (int j = 0; j < 8; j++) {
344        print_to_file(basic_luts[i][j], LUT_SIZE, "..\\basic waveforms\\" + names[i] +
       "\\" + to_string(j) + ".txt");
345      }
346    }
347 }
348
349
350 void run_tests() {
351    int N = 44100;
352    float* wn = new float[N];
353    generate_wn(wn, N);
354
355    float* chirp_freq = new float[N];
356    generate_linear_chirp_frequencies(chirp_freq, N, DIGITAL_FREQ_20HZ,
         DIGITAL_FREQ_20KHZ*0.5f); //20 to 10k
357
358    float* out = new float[N];
359    float* out2 = new float[N];
360    float* adsr = new float[N];
361
362    //FILTERS
363    cout << "_____" << endl;
364    cout << "STARTING FILTER TESTS" << endl;
```

```cpp
365    cout << "_____" << endl;
366
367    cout << "HP24" << endl;
368    print_to_file(wn, N, "..//testfiles//filter//white.txt");
369    test_filter(wn, out, adsr, N, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f / FS,
          5.0f, &iir_calc_hp24_coeff);
370    print_to_file(out, N, "..//testfiles//filter//hp24.txt");
371    cout << "LP24" << endl;
372    test_filter(wn, out, adsr, N, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f / FS,
          5.0f, &iir_calc_lp24_coeff);
373    print_to_file(out, N, "..//testfiles//filter//lp24.txt");
374    cout << "BP12" << endl;
375    test_filter(wn, out, adsr, N, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f / FS,
          5.0f, &iir_calc_bp12_coeff);
376    print_to_file(out, N, "..//testfiles//filter//bp12.txt");
377    cout << "HP12" << endl;
378    test_filter(wn, out, adsr, 44100, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f /
          FS, 5.0f, &iir_calc_hp12_coeff);
379    print_to_file(out, N, "..//testfiles//filter//hp12.txt");
380    cout << "LP12" << endl;
381    test_filter(wn, out, adsr, 44100, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f /
          FS, 5.0f, &iir_calc_lp12_coeff);
382    print_to_file(out, N, "..//testfiles//filter//lp12.txt");
383    cout << "WRITING ENVELOPE" << endl;
384    print_to_file(adsr, N, "..//testfiles//filter//adsr.txt");
385    int i = 0;
386
387    cout << endl;
388    cout << "_____" << endl;
389    cout << "STARTING WAVETABLE TESTS" << endl;
390    cout << "_____" << endl;
391    //WAVETABLE
392    for (float p = 0.0; p < 4.0f; p += 0.5f)
393    {
394      cout << "POS = " << p << endl;
395      test_wavetable(out, N, p);
396      print_to_file(out, N, "..//testfiles//wavetable//" + to_string(i) + ".txt");
397      i++;
398    }
399
400    cout << "SQUARE CHIRP" << endl;
401    test_wavetable_sweep(out, N, chirp_freq, 3.0f);
402    print_to_file(out, N, "..//testfiles//wavetable//square chirp.txt");
403
404    cout <<  endl;
405    cout << "_____" << endl;
406    cout << "STARTING GENERATOR TESTS" << endl;
407    cout << "_____" << endl;
408    //GENERATOR
409    cout << "VIBRATO" << endl;
410    test_vibrato(out, N);
411    print_to_file(out, N, "..//testfiles//generator//vibrato.txt");
412
413    float w[] = { 0.0f, 0.33f, 0.67f, 1.0f };
```

```
414
415   for (int i = 0; i < 4; i++)
416   {
417     cout << "WIDTH = " << w[i] << endl;
418     test_stereo_width(out, out2, N, w[i]);
419     print_to_file(out, N, "..//testfiles//generator//width L " + to_string(i) + ".txt
        ");
420     print_to_file(out2, N, "..//testfiles//generator//width R " + to_string(i) + ".
        txt");
421   }
422
423   cout << endl;
424   cout << "_____" << endl;
425   cout << "STARTING WAVESHAPER TESTS" << endl;
426   cout << "_____" << endl;
427
428   print_to_file(lut_tanh, TANH_LUT_SIZE_P1 , "..//testfiles//waveshape//tanh lut.txt"
        );
429
430   //WAVESHAPE
431   for (int i = 0; i < 5; i++)
432   {
433     float ip1 = (float)i + 1.0f;
434     float g_sin = PI / 2.0f * ((float)i + 1.0f);
435     float g_tanh = ip1 * ip1 * 2.0f;
436
437     cout << "G_TANH = " << g_tanh << " G_SIN = " << g_sin << endl;
438
439     test_waveshape(out, out2, N, chirp_freq, g_tanh, g_sin, true);
440     print_to_file(out, N, "..//testfiles//waveshape//tanh " + to_string(i) + ".txt");
441     print_to_file(out2, N, "..//testfiles//waveshape//sin " + to_string(i) + ".txt");
442
443     test_waveshape(out, out2, N, chirp_freq, g_tanh, g_sin, false);
444     print_to_file(out, N, "..//testfiles//waveshape//tanh " + to_string(i) + " no aa.
        txt");
445     print_to_file(out2, N, "..//testfiles//waveshape//sin " + to_string(i) + " no aa.
        txt");
446   }
447
448   cout << endl;
449   cout << "_____" << endl;
450   cout << "STARTING ADSR TESTS" << endl;
451   cout << "_____" << endl;
452   cout << "ADSR WITH RETRIGGER" << endl;
453   test_adsr(out, N);
454   print_to_file(out, N, "..//testfiles//adsr//adsr retrigger.txt");
455
456   cout << endl;
457   cout << "_____" << endl;
458   cout << "STARTING MANAGER TESTS" << endl;
459   cout << "_____" << endl;
460   cout << "TESTING MANAGER OPERATION" << endl;
461   test_gm_operation(out, N);
462   print_to_file(out, N, "..//testfiles//manager//operation.txt");
```

```
463
464   cout << "TESTING MANAGER OVERLOAD" << endl;
465   test_gm_overload(out, N);
466   print_to_file(out, N, "..//testfiles//manager//overload.txt");
467
468   cout << "TESTING OVERALL SYSTEM" << endl;
469   test_overall_system(out, out2, N);
470   print_to_file(out, N, "..//testfiles//manager//overall L.txt");
471   print_to_file(out2, N, "..//testfiles//manager//overall R.txt");
472 }
473
474
475 int main() {
476   cout << "INITIALISING" << endl;
477   init_basic_luts();
478   init_notes_digital_freq_buffer();
479
480   int size = FS*3;
481   int off = 3000;
482   float* L = new float[size];
483   float* R = new float[size];
484
485   gen_manager gm;
486   gen_config gc;
487
488   gm_init(&gm);
489
490   cout << "LOADING MOONLIGHT SONATA" << endl;
491   load_notes("..\\midifiles\\moonlight sonata.mid", 0, true);
492
493   cout << endl << "_____" << endl;
494   cout << "CONFIGURATION 1" << endl;
495   gen_config_volume_envelope(&gc, 0.05f, 0.05f, 0.8f, 0.06f);
496   gen_config_wavetables(&gc, 2.0f, 10.0f, 0.8f, 0.3f);
497   gen_config_filter_envelope(&gc, 0.1f, 0.1f, 1.0f, 1000.0f, 3.0f, 500.0f, 0.7f);
498   gen_config_filter(&gc, &iir_calc_lp12_coeff);
499   gen_config_vibrato(&gc, 15.0f, 7.0f / FS);
500   gen_config_tanh_saturator(&gc, 1.0f);
501   gm_apply_vibrato_config(&gm, &gc);
502   cout << "WRITING TO WAV" << endl;
503   write_midi_to_wav(&gm, &gc, "moonlight 1", false);
504   cout << "_____" << endl;
505
506   cout << "CONFIGURATION 2" << endl;
507   gen_config_volume_envelope(&gc, 0.05f, 0.2f, 0.5f, 0.01f);
508   gen_config_wavetables(&gc, 2.0f, 0.0f, 0.8f, 0.8f);
509   gen_config_filter_envelope(&gc, 0.15f, 0.1f, 1.0f, 1000.0f, 100.0f, 3.0f, 2.0f);
510   gen_config_filter(&gc, &iir_calc_lp24_coeff);
511   gen_config_vibrato(&gc, 0.0f, 7.0f / FS);
512   gen_config_sine_saturator(&gc, 2 * PI);
513   gm_apply_vibrato_config(&gm, &gc);
514   cout << "WRITING TO WAV" << endl;
515   write_midi_to_wav(&gm, &gc, "moonlight 2", false);
516   cout << "_____" << endl;
```

```cpp
517
518    cout << "CONFIGURATION 3" << endl;
519    gen_config_volume_envelope(&gc, 0.08f, 0.2f, 0.75f, 0.08f);
520    gen_config_wavetables(&gc, 2.5f, 1200.0f, 0.8f, 0.2f);
521    gen_config_filter_envelope(&gc, 0.15f, 0.2f, 1.0f, 1000.0f, 100.0f, 10.0f, 0.7071f)
          ;
522    gen_config_filter(&gc, &iir_calc_lp24_coeff);
523    gen_config_vibrato(&gc, 10.0f, 7.0f / FS);
524    gen_config_tanh_saturator(&gc, 5.0f);
525    gm_apply_vibrato_config(&gm, &gc);
526    cout << "WRITING TO WAV" << endl;
527    write_midi_to_wav(&gm, &gc, "moonlight 3", false);
528    cout << "_____" << endl;
529
530    cout << "CONFIGURATION 4" << endl;
531    gen_config_volume_envelope(&gc, 0.08f, 0.2f, 0.75f, 0.08f);
532    gen_config_wavetables(&gc, 3.2f, 20.0f, 0.8f, 0.5f);
533    gen_config_filter_envelope(&gc, 0.2f, 0.2f, 1.0f, 1000.0f, 1.0f, 3.0f, 3.0f);
534    gen_config_filter(&gc, &iir_calc_hp24_coeff);
535    gen_config_vibrato(&gc, 0.0f, 7.0f / FS);
536    gen_config_no_saturator(&gc);
537    gm_apply_vibrato_config(&gm, &gc);
538    cout << "WRITING TO WAV" << endl;
539    write_midi_to_wav(&gm, &gc, "moonlight 4", false);
540    cout << "_____" << endl;
541
542    cout << "CONFIGURATION 5" << endl;
543    gen_config_volume_envelope(&gc, 0.005f, 0.1f, 0.0f, 0.08f);
544    gen_config_wavetables(&gc, 2.9f, 0.0f, 0.0f, 0.25f);
545    gen_config_filter_envelope(&gc, 0.1f, 0.1f, 1.0f, 1000.0f, 700.0f, 7.0f, 2.0f);
546    gen_config_filter(&gc, &iir_calc_lp24_coeff);
547    gen_config_vibrato(&gc, 0.0f, 7.0f / FS);
548    gen_config_no_saturator(&gc);
549    gm_apply_vibrato_config(&gm, &gc);
550    cout << "WRITING TO WAV" << endl;
551    write_midi_to_wav(&gm, &gc, "moonlight 5", false);
552    cout << "_____" << endl;
553
554    run_tests();
555    return 0;
556 }
```

Listing E.1: Code used to generate test data for chapter 4

Appendix F

Musician feedback

Gerainn Everson

Gerainn is a guitarist, pianist, vocalist and band member who often performs in his spare time. Gerainn mostly noticed the lack of human dynamics in the audio, which is a result of the MIDI used, and not the system. However, he would prefer greater dynamic contrast, which can be achieved through exponentially scaling MIDI velocity values.

He enjoyed the variety of sounds that are achievable with the system, but commented on some of the settings not being according to his taste (long envelope release times and too intense vibrato on track 3), which is not a reflection on the system itself. He enjoyed tracks 1 and 5 the most, and only noticed the harmonic indexing limitations once it was pointed out to him.

Gina Theron

Gina is a drummer, accordionist, vocalist and band member who enjoys playing in her spare time. She also noticed the lack of human dynamics and lack of dynamic contrast, like Gerainn. She only had feedback on the parameters not being according to her taste (track 3 vibrato too fast). She thoroughly enjoyed track 4 and 5. She only noticed the harmonic indexing drawbacks on a second listen, after it was pointed out.

Mike Donaldson

Mike is a guitarist who has performed numerous times with a variety of people. He had similar feedback to Gina and Gerainn regarding dynamics, but enjoyed all the tracks. He did not notice the harmonic indexing drawbacks, and found it difficult to hear, even after it was pointed out.

Appendix G
Project Planning Schedule

Week	Date	Planned work	Work done		Supervisor feedback
June/July holidays		Implement initial software system.	Implemented basic system using Q-numbers.		
1	09-Aug-21	Refactor to floating point.	Refactored Q-number code to floating point.	11-Aug-21	Initial meeting to discuss Skripsie approach
2	16-Aug-21	Start block diagrams. Debug.	Halfway done block diagrams. Debugged current code. More refactoring.	18-Aug-21	Agreement between skripsie student and study leader regarding mutual responsibilities
3	23-Aug-21	Block diagrams. Design documentation.	Almost done with block diagrams. Design documentation started.	20-Aug-21	Meeting to discuss mathematical description of skripsie topic
4	30-Aug-21	Start literature overview.	Literature overview started. Started work on waveshaping.	29-Aug-21	Alternatives to STM32 implementation mentioned.
5	06-Sep-21	Conduct basic tests. Work on full system.	Conducted some tests. Debugging. MATLAB simulation of waveshaping.	06-Sep-21	Mixed version of 4 tracks resulting from the software. Spectograms of a tanh(G*sin(2pi*f)) distortion, with a frequency sweep of 20 to 20kHz. Anti-aliasing strategy. Mentioned possible scope reduction.
6	13-Sep-21	Documentation and debugging. Work on MIDI library integration.	Documentation. More system debugging. Full MIDI library integration.		
7	20-Sep-21	Refactor code to avoid data duplication. Documentation.	Refactored code. Rewrote parts for efficiency. Added waveshaping support.	27-Sep-21	Code for predifined LUTs. Master design for software. Software is complex, intensive, and can fill up the entire report - SCOPE CHANGED TO EXCLUDE STM32 IMPLEMENTATION.
8	27-Sep-21	Start system tests. MATLAB analysis code.	MATLAB analysis code. Started basic system tests. Shortened report. Report length too long.		
9	04-Oct-21	Debugging.	Debugging. System is mostly operational.		
10	11-Oct-21	Debug. Generate test audio. Identify further problems.	Generated test audio. Better system interface added. Better waveshaping.	12-Oct-21	Additional audio track. Skripsie progress. Report format not up to standard. Must change report orginisation.
11	18-Oct-21	Reorganise report.	Reorganised report. Continued work on documentation. Shortened report.		
12	25-Oct-21	System tests and analysis.	System testing and documentation.		
13	01-Nov-21	System tests and analysis. Work on testing documentation	Conducted interviews for final test. Updated figures and testing documentation.		
14	08-Nov-21	Report writing.	Finished report. Commence editing, and prepare presentation.	09-Nov-21	Feedback on the entirety of skripsie project
15	15-Nov-21	Hand in. Work on poster and video.			

Appendix H

Outcomes Compliance

ELO 1: Problem solving

Identify, formulate, analyse and solve complex engineering problems creatively and innovatively

A variety of problems had to be addressed in designing this system. This includes: developing own techniques to combat aliasing; planning and implementing efficient code; considering all design aspects that could influence the results; specifying the requirements, scope and DSP signal chain of the system; using existing knowledge to analyse and derive equations. The entirety of chapter 3 demonstrates this ELO.

ELO 2: Application of scientific and engineering knowledge

Apply knowledge of mathematics, natural sciences, engineering fundamentals and an engineering speciality to solve complex engineering problems.

Discrete mathematics, which includes Z-domain analysis and DTFTs, is used. Fourier series, calculus, conventional mathematics and human-hearing considerations are present. Musical knowledge is integrated with engineering knowledge. Frequency domain knowledge is used to perform numerical analysis. Extensive use of linear interpolation and LUTs, which are common in microcontroller implementations for a variety of applications, is present. The use of common data structure, which includes queues, stacks, and hash tables, are used to the system's advantage. The entirety of chapters 3 and 4 reflect this.

ELO 3: Engineering Design

Perform creative, procedural and non-procedural design and synthesis of components, systems, engineering works, products or processes.

Creative design is implemented in designing the DSP chain of this system. Rigorous software design, hardware considerations and state-machines are present in chapter 3. Filters are designed for both obvious and non-obvious audio results (i.e. anti-aliasing filtering). Various schemes are used to make the system computationally efficient, and required the translation of derived equations to software. Common data structures, which includes queues, stacks, and hashtables, have a custom implementation to leverage system performance.

ELO 4: Investigations, experiments and data analysis

Demonstrate competence to design and conduct investigations and experiments.

Creative ways to measure system performance is used, since the designed system is complex and difficult to measure unless it is configured in a certain way. This includes the extensive use of spectrograms, white noise filtering, frequency sweeps (chirps) and a lissajous (XY) plot. Custom C code is written to properly configure and test the system for all operating conditions. Chapter 4 demonstrates this ELO.

ELO 5: Engineering methods, skills and tools, including Information Technology

Demonstrate competence to use appropriate engineering methods, skills and tools, including those based on information technology.

Microsoft Visual Studio, MATLAB and Github are the primary tools used in this project. Advanced use of MATLAB, and custom test code written in C, demonstrate extensive use of engineering skills and tools. Chapters 3 and 4 reflects this ELO.

ELO 6: Professional and technical communication

Demonstrate competence to communicate effectively, both orally and in writing, with engineering audiences and the community at large.

This report is well-documented and highly technical, requiring extensive knowledge of mathematics, with almost no information left out. Code is made publicly available. A presentation video is created.

ELO 8: Individual work

Demonstrate competence to work effectively as an individual.

All work performed is individual, planned by a schedule. Work was efficiently performed over the required time period, without falling behind on schedule.

ELO 9: Independent Learning Ability

Demonstrate competence to engage in independent learning through well-developed learning skills.

Many new skills were acquired, which includes filter implementation, design specification, musical integration, MATLAB analyses, numerical analysis, efficient C implementations and using an external MIDI library.