University of Leipzig
Faculty of Mathematics and Computer Science
Department of Computer Science

Master's Thesis

Programming a remote controllable real-time FM audio synthesizer in Rust

Author
Andreas Linz

Supervisor
Prof. Dr. Mario Hlawitschka

Second Reader
Prof. Dr. Gerik Scheuermann

Submitted by **Andreas Linz** to the *Department of Computer Science* in partial fulfilment of the requirements for the degree of
Master of Science — Computer Science
Leipzig, 8. Januar 2017

Version: `v0.9.1-16-g6d4d939`

To my mother.
We all hope that you'll get well, soon.

Contents

Acknowledgements

I would like to express my appreciation to the following people for their their help and support:

- My supervisor Prof. Dr. Mario Hlawitschka for his DSP lecture which originally sparked my interest in this topic, his advice and patience when the finishing of this thesis was delayed.
- My family, especially my parents and sister, for their unconditional support during this research and the rest of my academic career. Without your help, I may have given up.
- Lucas, René, Marcus E., Mario L., Katharina, Florian, Carolin, Tobias S. and Tobias J. (in no particular order) for many fruitful discussions, giving me feedback and sometimes another perspective on the topic.

In addition, I would like to thank spreadshirt for giving me the flexibility in my working schedule that was needed to finish this thesis. Moreover, finishing this thesis would not have been possible without spreadshirt's financial support.

Colophon

The following open-source tools were used to create this document:

- pandoc
- LaTeX 2_ε, XeLaTeX and Ti*k*Z
- GNU make
- vim

Abstract

English

Software Audiosynthesizers have gained in popularity over the past 10 years and it is impossible to imagine professional or home studios without them. This popularity is mostly justified by the high computing power, which is available everywhere on PCs and mobile devices and makes real-time audio synthesis usable. The aim of this work is the detailed description of basic synthesizer components and the investigation of suitable algorithms and techniques for their realization.

In the first part, input protocols are considered and the fundamental building blocks of a synthiser are introduced. Subsequently, different synthesis techniques are explained and the decision for the choice of the subtractive synthesis is explained.

The second part deals with signal processing topics of the individual synthesizer components. The basics of digital filters are explained and FIR and IIR filters are compared. Thereafter, the audio quality and efficiency of various waveform synthesis methods is evaluated, i. a. band-limited impulse trains (BLIT) and wavetables. Implementation details are explained in chapter 7 and the components of the synthesizer application are evaluated.

It was found through the investigation of the synthesizer implementation that the selected techniques and algorithms have high audio quality and low calculation costs, in particular Rust has proven to be a very suitable choice for the development of real-time applications.

German

Software Audiosynthesizer haben in den letzten 10 Jahren enorm an Popularität gewonnen und sind in vielen Profi- und Heimstudios nicht mehr wegzudenken. Diese Popularität ist durch die hohe Rechenleistung begründet, welche auf PCs und mobilen Geräten überall zur Verfügung steht und Echtzeitaudiosynthese nutzbar macht. Das Ziel dieser Arbeit ist die ausführliche Beschreibung grundlegender Synthesizerkomponenten und die Untersuchung geeigneter Algorithmen und Techniken für deren Realisierung.

Im ersten Teil der Arbeit werden Eingabeprotokolle betrachtet und die fundamentalen Bausteine eines Synthesizers eingeführt. Anschließend werden verschiedene Synthesetechniken erläutert und die Entscheidung für die Wahl der subtraktiven Synthese begründet.

Der zweite Teil beschäftigt sich mit der Signalverarbeitung innerhalb der einzelnen Synthesizerkomponenten. Die Grundlagen von digitalen Filtern werden erläutert und FIR mit IIR Filtern verglichen. Nachfolgend wird die Audioqualität und Effizienz verschiedener Wellenformsynthesemethoden, u.a. bandbeschränkte Impulsfolgen (BLIT) und Wavetables, evaluiert. Implementierungsdetails werden im 7. Kapitel erläutert und die Komponenten des entwickelten Synthesizers ausgewertet.

Es zeigte sich durch Untersuchung der Synthesizerimplementation, dass die ausgewählten Techniken und Algorithmen eine hohe Audioqualität bei gleichzeitig niedrigen Berechnungskosten haben, insbesondere Rust hat sich als sehr geeignete Wahl für die Entwicklung von Echtzeitanwendungen erwiesen.

Andreas Linz
University of Leipzig
Faculty of Mathematics and Computer Science—Computer Science
Programming a remote controllable real-time FM audio synthesizer in Rust
8. Januar 2017

1 Introduction

Electronic musical synthesizers have a long history that started in 1906 with the *Telharmonium* [Roa96, p. 83], a very large device with rotating tone generators that emitted pure sinusoidal waves. It took nearly 50 years until the first experiments with digital sound synthesis were made in 1957 at Bell Telephone Laboratories with the development of Music I and II programs by Max V. Mathews [Roa96, p. 88]. Another 20 years later, in the late 70s, the first real-time FM synthesizers became commercially available, e.g. the Fairlight CMI in 1979. The first affordable digital hardware FM synthesizer, Yamaha's DX7, was introduced in 1983 and caused the decline of analog synthesizers because digital hardware became cheaper from year to year. Due to increasing computing power, real-time synthesis was made possible on general purpose computers in the early 2000s without the need for special DSP processors or other expensive audio hardware. Since then, a large number of vastly different software synthesizers was developed, both commercial as well as free or open source projects.

Despite the huge amount of computing power that is recently available even on commodity hardware, the development is still a challenging task because musicians expect an nearly instantaneous response from their instrument and dropped audio frames cause click sound which could make a whole recording useless. Therefore, audio software must make very efficient use of processing and power while

achieving the best possible amount of audio quality.

The set of problems that arise when developing a software synthesizer is quite large. It includes timing and synchronization problems that occur because the short time windows in which computations are required to finish. Another class of problems are sound artifacts like aliasing which must be avoided by choosing appropriate algorithms.

Audio software is usually implemented in programming languages with manual memory managment (unmanaged languages) like C or C++ for which a great number of frameworks already exist. Rust, on the other hand, is a promising new programming language that claims to be thread-safe, guarantees memory safety (unlike C and C++) and has little runtime cost because a garbage collector is not required. This makes Rust a great candidate for the development of modern audio software. But, due to the language's ecosystem still being in its early days, a lot of libraries had to be implemented from scratch.

The aim of this thesis is to evaluate available methods and algorithms for the use in a real-time polyphonic FM synthesizer and to implement a prototype that can be played with conventional audio controller hardware. Every piece of code that was implemented for this thesis is being open sourced, see chapter 7 for details.

1.1 Scope of this Thesis

It's not required for the reader to have any prior knowledge of soft- and hardware tools used in music production environments. However, a basic knowledge of signal processing and some familiarity with computer programming is beneficial to grasp the presented concepts.

2 User Interface

Interaction with the software synthesizer is done through its *user interface*. The user interface serves three senses, which are sight, touch and hearing. This section concentrates on the first two, the *visual* and the *haptic* component. However, a synthesizer can be played solely through haptic controls and audio feedback. Visual indicators for the synthesizer's parameters, e.g. through a display, are convenient, but not necessary for the playing musician.

The application supports the two most common musical control signal protocols, *MIDI* and *Open Sound Control* (OSC). Adding MIDI support is highly beneficial, because it enables the synthesizer to be played with any—of the vast amount of available—MIDI hardware controllers (fig. 2.1 shows an example of such a device). On the other hand, Open Sound Control software like liine's Lemur [Lii16] provides an editor to create or customize a software defined controller for a multitouch device like a smartphone or tablet.

2.1 MIDI

The Musical Instrument Digital Interface (MIDI) specification stipulates a hardware interconnection scheme and a method for data communication [Roa96, p. 972],

Figure 2.1: Edirol PCR-300 MIDI controller keyboard

but only the protocol specification is of interest for this work. Most modern MIDI hardware is connected via USB anyway. The *MIDI 1.0 Specification* [Ass14] provides a high level description of the MIDI protocol:

> *The Musical Instrument Digital Interface (MIDI) protocol provides a standardized and efficient means of conveying musical performance information as electronic data. MIDI information is transmitted in "MIDI messages", which can be thought of as instructions which tell a music synthesizer how to play a piece of music. [Ass14, p. 1]*

Transmitting *control data* is the purpose of the MIDI protocol, and not, like it is sometimes confused, to transmit audio data[1]. Control data can be thought of as the press of a key, turning a knob, or an instruction to change the clock speed of a song.

The work on the MIDI specification began in 1981 by a consortium of Japanese

[1] It is possible to transmit audio data over MIDI by using *System Exclusive* (SysEx) messages, but this can not be done in real-time and is often used to replace or update samples or wavetables in hardware synthesizers.

and American synthesizer manufacturers, the MIDI Manufacturers Association (MMA). In August 1983, the version 1.0 was published [Roa96, p. 974]. This year, 2016, the MMA established The MIDI Association (TMA). The TMA should support the global community of MIDI users and establish midi.org [Ass16] as a central source for information about MIDI. MIDI is used in nearly every music electronic device, like synthesizers, samplers, digital audio effects, and music software, due to its simple protocol structure and long time of existence.

2.1.1 MIDI Protocol

The MIDI protocol specifies a standard transmission rate of $31\,250\,\mathrm{bit\,s^{-1}}$. This may seem like an unusual choice for the transmission rate, but it was derived by dividing the common clock frequency of 1 MHz by 32 [Roa96, p. 976]. It uses an 8b/10b encoding, i.e. 8 bit of data are transmitted as a 10 bit word. A data byte is enclosed by a start and stop bit which in turn results in the 10 bit encoding. Asynchronous serial communication is used to transfer *MIDI messages*, thus the start and stop bit.

A MIDI message is composed of a *status byte* which is followed by up to two[2] *data byte*s. Both types are differentiated by their most significant bit (MSB), `1` for status- and `0` for data bytes. Consequently, the usable payload size is reduced to 7 bit, in other words, values can range from 0 to 127.

The structure of a MIDI status byte is as follows `0TTTCCCC` , where `T` denotes the three message type bits and `C` the remaining four bits that indicate the channel

[2] System Exclusive (SysEx) messages can be made up of more than two data-bytes, in fact they are build by a sequence of data bytes followed by an *End of Exclusive* (EOX) message to mark the end of the stream. This type of message does not contain any musical control data, in general it is used to upload binary data, like firmware updates or samples, to a MIDI device.

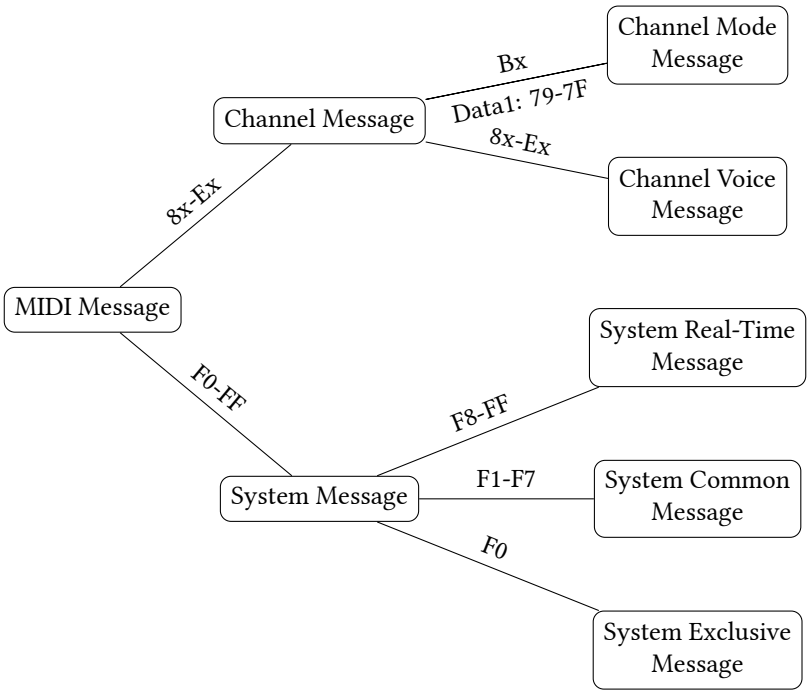

Figure 2.2: Classification of MIDI messages.

number. Hence, there are sixteen different channels addressable. MIDI channels allow to route different logical streams over one physical MIDI connection, e.g. to reach a different, daisy-chained MIDI device or to control different timbres of a multitimbral synthesizer.

MIDI messages are divided in two categories, *channel* and *system* messages. Only the latter contain musical control information and therefore are of interest for this thesis. Figure 2.2 illustrates the classification, status byte values are shown as edge labels where x illustrates *don't care*. *Channel Mode Messages* define the instrument's response to Voice Messages [Ass14, p. 36], i.e. listen on all channels (omni mode), or switch between mono- and polyphonic mode (multiple simultaneous voices).

Table 2.1: Types of MIDI Voice Messages.

Type	Status	Data1	Data2	Description
Note-Off	8x	Key #	Velocity	Key released.
Note-On	9x	Key #	Velocity	Key press from a triggering device.
Polyphonic Key Pressure	Ax	Key #	Pressure	Aftertouch event.
Control Change	Bx	Ctrl. #	Value	Move of a controller other than a key (e.g. Knob, Slider).
Program Change	Cx	Program #	—	Instruction to load specified preset.
Channel Pressure	Dx	Pressure	—	Aftertouch event.
Pitch Bend	Ex	MSB	LSB	Altering pitch (14-bit resolution).

2.1.2 MIDI Pitch

Table 2.1 gives an overview of the types on voice messages. Corresponding Note-On and Off messages do not necessarily follow one after another, therefore, to relate associated messages, pitch information is contained in the Note-Off as well. Pitch is encoded as a 7 bit value in note messages, hence there is a range of 128 pitches or about 10 octaves. MIDI's pitch representation was designed with an *chromatic western music scale* in mind. A *chromatic scale* has 12 pitches per octave with one semitone difference between each pitch, that is a ratio of $2^{1/12}$ between successive notes. An interval of one octave is equivalent to a doubling or halving (in the negative case) in frequency. Instruments in *western music* are usually *equal-tempered*, i.e. all semitones have the same size. MIDI pitches are considered to be equal-tempered and range from C0 (*c* in the lowest octave) to a G10 (*g* in the 10th octave). Middle C, pitch number 60 (C5), is used as reference.

$$
\begin{aligned}
f &= f_{\text{tune}} \cdot 2^{(p - p_{\text{ref}})/12} \\
p &= p_{\text{ref}} + 12 \cdot \log_2(f / f_{\text{tune}})
\end{aligned}
\tag{2.1}
$$

Equation (2.1) shows how to calculate the frequency f for a given MIDI pitch p, and vice versa, where f_{tune} is the tuning frequency and p_{ref} is the reference pitch number. Musical instruments are commonly tuned to the *Concert A*, the note A above middle C or MIDI pitch 69. The default tuning of Concert A is 440 Hz [Com75]. The following example shows how to calculate the frequency for middle C by using the *Concert A* tuned to 440 Hz as reference pitch in eq. (2.1):

$$
\begin{aligned}
f &= 440\,\text{Hz} \cdot 2^{(60-69)/12} \\
&= 440\,\text{Hz} \cdot 2^{-9/12} \\
&\approx 261.626\,\text{Hz}
\end{aligned}
$$

2.1.3 Timing Problems

Playing two or more notes a the same time, i.e. playing a chord, can lead to timing problems because of MIDI's low bandwidth.

$$
\begin{aligned}
t_{\text{Note-On}} &= 3 \cdot \left(31\,250\,\text{bit}\,\text{s}^{-1} / 10\,\text{bit}\right)^{-1} \\
&= 0.000\,96\,\text{s} = 0.96\,\text{ms}
\end{aligned}
$$

The time to transmit a single note-on event $t_{\text{Note-On}}$ takes ≈ 1 ms, this means that the last transmitted note of an n-key chord arrives with $n \cdot 0.96$ ms delay, e.g. the last note of a pentachord (5 keys) will be received $5 \cdot 0.96\,\text{ms} = 4.8\,\text{ms}$ later than

the first one. This may result in a *comb filter*[3] like distortion of the synthesized chord sound.

2.2 Open Sound Control

The *UC Berkeley Center for New Music and Audio Technology* (CNMAT) originally developed, and continues to research, Open Sound Control. In 2002, OSC's 1.0 specification was released. It provides the following definition [Wri02]:

> *Open Sound Control (OSC) is an open, transport-independent, message-based protocol developed for communication among computers, sound synthesizers, and other multimedia devices.*

The protocol is not limited to being used with audio or multimedia devices, however, it is often used as a high-speed network replacement for MIDI. Referring to OSC as a *message format* is more accurate, since error-handling, synchronization or negotiation methods are not specified. Therefore, OSC can be compared to formats like JSON or XML. A draft of the OSC 1.1 specification was published in a 2009 paper [FS09] only adding minor, backward compatible changes. UDP is often used as the transport layer to avoid the time required to establish a connection by TCP's three-way handshake. A connection less transport is sufficient because OSC sender and receiver are almost always in physical proximity and connected through the same LAN.

[3] A comb filter adds a delayed copy of the signal to itself causing addition or subtraction in the signal. The filters frequency response shows regularly spaced notches, might resemble the shape of a comb.

Table 2.2: Overview of OSC 1.0 and 1.1 data types.

Tag	Description	1.0 Required	1.1 Required
i	32 bit two's complement integer	✓	✓
f	IEEE 754 single precision (32 bit)	✓	✓
s	null-terminated sequence of ASCII characters	✓	✓
b	binary blob with size information	✓	✓
t	OSC-timetag in NTP format		✓
T/F	boolean values: true, false		✓
N	Nil		✓
I	Infinitum (1.0)/Impulse(1.1) used as event trigger		✓
d	IEEE 754 double precision (64 bit)		
h	64 bit big-endian two's complement integer		
S	alternate string type		
c	ASCII character		
r	RGBA color (8 bit per channel)		
m	4 B MIDI message (from MSB to LSB): port, status, data1, data2		
[,]	Array delimiters		

2.2.1 OSC Data Types

An overview of the predefined data types for both, OSC 1.0 and 1.1, is shown in table 2.2. The byte order of OSC's integer, float and timetags is big-endian. OSC's unit of transmission is called *OSC Packet*. The EBNF grammar for OSC packets is described by fig. 2.4. Fields of an OSC packet have to be aligned to multiples of 4-byte and are zero-padded, thus the size of such a packet is also a multiple of four. The packets contents can either be an *OSC Message* or *OSC Bundle*. An OSC message starts with an *address pattern* followed by zero or more *arguments* to be applied to the *OSC Method* matched by the pattern. Address pattern can contain basic regular expression with single-/multi-character `?/*` wildcards, range `[A-Z]`

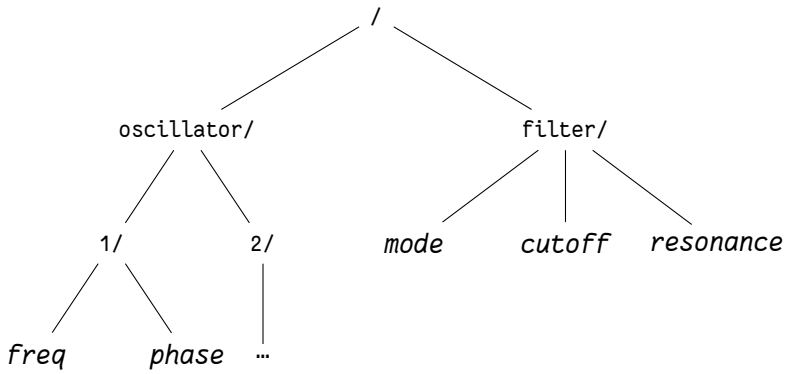

Figure 2.3: OSC Address Space example.

and list matches `{foo, bar}`, hence multiple OSC Methods can be triggered with a single OSC Message. An OSC Receiver's[4] address space forms a tree structure with branch nodes called *OSC Containers* and leaves are named *OSC Methods*. Methods are *italicized* in the tree structure example of fig. 2.3. The address of an OSC method starts with a `/`, followed by any container name along the path in order from the root of the tree, joined by forward slashes `/` and the method's name, e.g. `/oscillator/1/phase`.

2.2.2 Comparison to MIDI

Both protocols provide a number of benefits and limitations in comparison to each other. The following list shows them for OSC compared to MIDI:

+ OSC's data-types allow a much higher resolution for control values. They also provide symbolic types like booleans or *Nil* to represent an empty value.

+ The definition of *custom data-types* is allowed, therefore OSC applications

[4] The term *OSC Receiver* and *OSC Server* is interchangeable. This also applies to *OSC Sender* and *OSC Client*. OSC applications often act as server and receiver, hence no clear distinction between those roles can be made.

Figure 2.4: Grammar of an OSC packet described as EBNF (ISO14977 syntax [Int96, p. 14]).

```
packet          = size, content ;
size            = (* 4-byte aligned packet content field length *) ;
content         = message | bundle ;
message         = address, ",", { type-tag }, { argument } ;
address         = "/", osc-string - ( "'" | "#" | "*" | "," | "/" |
                                      "?" | "[" | "]" | "{" | "}" ) ;
osc-string      = { ASCII }, "0" ;
type-tag        = "i" | "f" | "s" | "b" | "h" | "t" | "d" | "S" |
                  "c" | "r" | "m" | "T" | "F" | "N" | "I" |
                  "{", {type-tag}, "}" ;
argument        = (* binary representation of the argument *) ;
bundle          = "#bundle", OSC-timetag , { bundle-element } ;
bundle-element  = size, content ;
```

must be made robust against unknown ones.

+ The *bandwidth* is orders of magnitudes larger than MIDI's, but it depends on the type of network used. A common choice are ad-hoc Wi-Fi connections between OSC receiver and sender because the player (sender) and the instrument (receiver) are in local proximity to each other. This, in turn, results in an acceptable network *latency* in the single digit millisecond range.

+ Control events can be send simultaneously as an OSC bundle, e.g. note events of a chord.

+ Events can be timed with an resolution of ≈ 200 picoseconds [Wri02].

+ OSC can be used to tunnel MIDI messages over a network connection.

− There is no standard for discovering OSC devices in a network, thus addresses must be configured manually which is cumbersome.

— Unlike MIDI, there is no standard namespace for interfacing with an OSC device, although, a proposal for a standard exists [Ehr13].

— The number of applications that support OSC is very limited.

3 Synthesizer Fundamentals

This chapter outlines the fundamental elements of a synthesizer and briefly describes the fundamental methods of sound generation.

3.1 Oscillator

Oscillators are the fundamental building blocks of a synthesizer's sound generation engine. They serve the purpose of emitting a periodic waveform. An oscillator is controlled through its *frequency* and *amplitude* parameters. In the context of a synthesizer there are additional controls for starting *phase*, the point at which the waveforms begins, and *type of waveform* to emit.

The amplitude parameter sets the *peak amplitude* for the signal, i.e. the absolute value of the waveforms highest amplitude. Frequency is usually specified as number of waveform cycles per second (Hz) but in the software implementation stored as *phase increment* (angular frequency) for each sample step. The software oscillators output is a sequence of samples at equidistant intervals T. Let $f_s = 1/T$ be the sample rate and f the frequency in Hz (s^{-1}), then the phase increment ω is calculated like this:

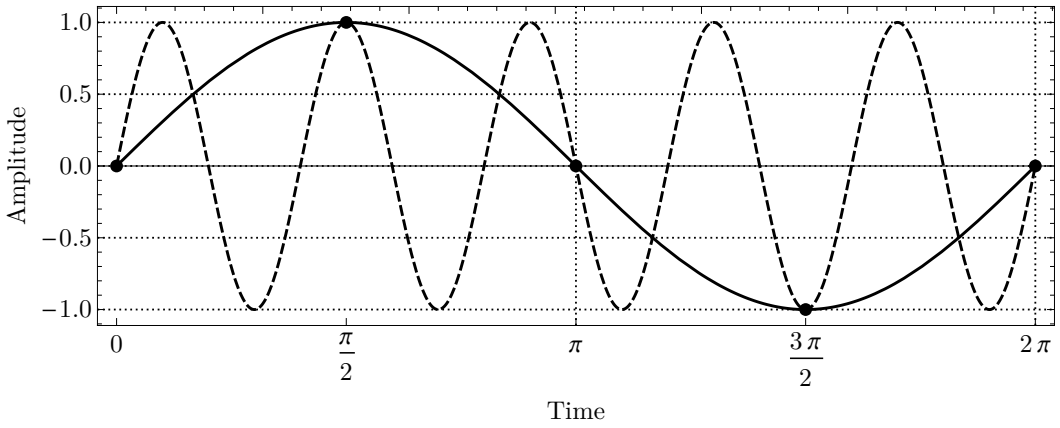

Figure 3.1: Two sinusoids with angular frequencies $\omega_1 = \pi/2$, $\omega_2 = 3/2\,\pi$ sampled in intervals of $T = \pi/2$. Both sinusoids produce the same sampled signal due to aliasing.

$$\omega = \frac{2\pi f}{f_s} \tag{3.1}$$

The oscillators highest frequency is limited to $f_s/2$ or $\omega = \pi$, which is called *Nyquist frequency*. In general, the (Nyquist-Shannon) *sampling-theorem* states that a signal can be exactly reconstructed from its digitization if its entire frequency spectrum lies below the Nyquist frequency [Ben08, p. 244]. In other words, it ensures that there are at least two sample points for any period of a frequency component contained in the sampled signal.

3.1.1 Aliasing

Figure 3.1 shows two sinusoids with frequencies $\omega_1 = \pi/2$ and $\omega_2 = 3/2\,\pi$ that are sampled at sample rate $\omega_s = \pi/2$. Clearly, ω_2 is above the Nyquist frequency $\omega_{\text{Ny}} = \pi$, thus ω_2 is *foldover* at ω_{Ny} which results in a frequency of ω_2

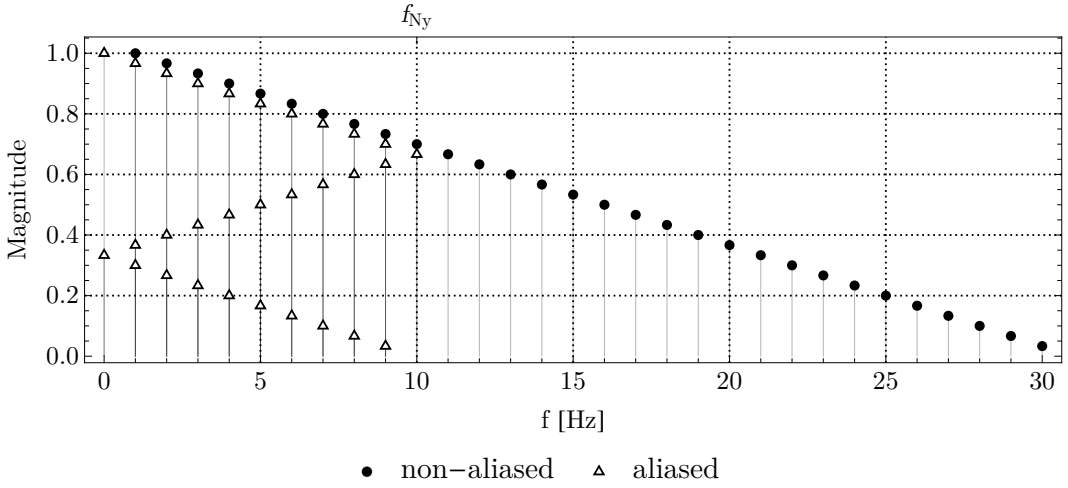

Figure 3.2: Spectra for waveforms sampled at a) $f_{\text{Ny}} = 30$ and b) $f_{\text{Ny}} = 10$ which is $1/3$ of the highest frequency contained, hence foldover (aliasing) occurs.

mod $\omega_{\text{Ny}} = \pi/2$ that is equal to ω_1, therefore ω_2 is an *alias* of ω_1, so both signals are indistinguishable after the sampling process. This effect is called *aliasing* or *foldover* and is an inevitable result of sampling or sample rate conversion, hence signal components with frequencies above Nyquist must be removed or reduced before fed into the sampling process. The effect of foldover in the frequency spectrum is shown in fig. 3.2.

The alias f_a for a frequency f and a sampling frequency f_s can be calculated as shown in eq. (3.2)[1], where $N = \lfloor f/f_s \rfloor$

$$f_a = \begin{cases} |N\, f_s - f| & \text{if } N \text{ is even} \\ |(N+1) f_s - f| & \text{otherwise} \end{cases} \tag{3.2}$$

[1]Dashow presented a method for generating non-harmonic spectra using foldover frequencies [Das78, p. 82].

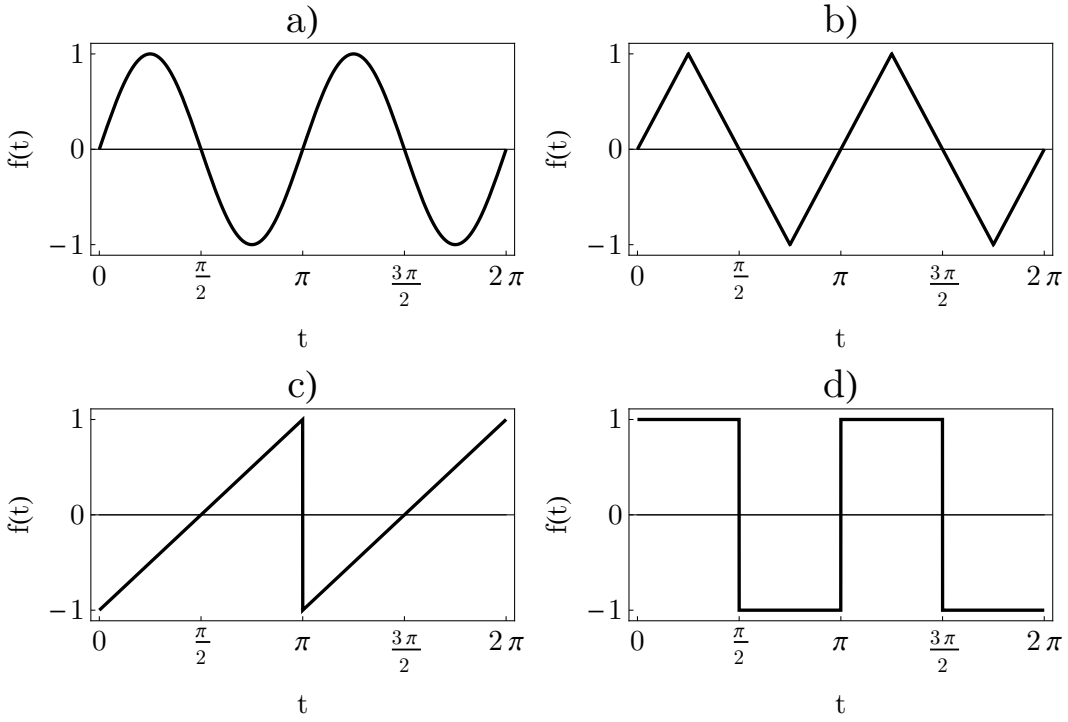

Figure 3.3: Common (non bandlimited) waveforms supported by most synthesizers: a) sine wave, b) triangle wave, c) ramp/sawtooth, d) square wave.

3.2 Waveforms

Synthesis techniques like subtractive synthesis require a source signal with rich harmonic content, hence providing only sine wave oscillators is not sufficient. Oscillator waveforms typically used in subtractive synthesis are *triangle*, *sawtooth* and *square wave* the so called *trivial* or *geometric waveforms* [Pek14, p. 5] because of their well-defined shape consisting of piece-wise linear or constant segments. Non bandlimited versions of those waveforms are shown in fig. 3.3. To prevent aliasing artifacts caused by discontinuities in the waveform (square and sawtooth) or its slope (triangle) it is required to create bandlimited versions of those waveforms.

3.2.1 Fourier Synthesis

Fourier synthesis is used to create bandlimited versions of those complex waveforms. It uses the properties of *Fourier series* representation of arbitrary but periodic waveforms [Loy11, p. 103]:

> *Any periodic vibration, [...], can be built up from sinusoids whose frequencies are integer multiplies of a fundamental frequency, by choosing the proper amplitudes and phases.*

This means that any periodic waveform can be constructed by specifying the power of each its *harmonics*, where a *harmonic* is an integer multiple of the waveforms fundamental frequency. The Fourier series is defined as (eq. (3.3))

$$f(t) = a_0/2 + \sum_{k=1}^{\infty} a_k \cos(2\pi k t + \phi_k) + \sum_{k=1}^{\infty} b_k \sin(2\pi k t + \phi_k) \tag{3.3}$$

where a_k and b_k are coefficients for the strength of the k-th harmonic. The harmonics phase is specified ϕ_k and $\omega_k = 2\pi k$ sets its angular frequency. By substituting[2] eulers equation (eq. (3.4))

$$e^{i\phi} = \cos(\phi) + i \sin(\phi) \tag{3.4}$$

into eq. (3.3) it can be written in complex form:

$$f(t) = \sum_{k=-\infty}^{\infty} c_n e^{i\, 2\pi\, k\, t + \phi_k} \tag{3.5}$$

[2] Eulers identities: $\cos = (e^{i\phi} + e^{-i\phi})/2$ and $\sin = (e^{i\phi} + e^{-i\phi})/(2i)$.

.

The summation range has changed for the complex Fourier series eq. (3.5) to $-\infty$ and ∞, thus there are now negative frequencies. To retrieve a real-valued signal from the complex Fourier series it is required to take the conjugate transpose of each value, i.e. to specify the coefficients c_k as pairs with a $-c_k$ for each positive coefficient.

The complex coefficients c_k can be converted from the a_k, b_k's with the following equation:

$$c_{\pm k} = 1/2\,(a_k \pm i\,b_k) \tag{3.6}$$

3.2.2 Bandlimited Waveforms

Bandlimited sawtooth, triangle and square waveforms can be created by means of Fourier synthesis. Sawtooth, in contrast to square and triangle waves, contains odd and even harmonics which makes them a great source signal because of their rich harmonic content. Square and triangle waves consist solely of odd harmonic partials. The complex Fourier series for sawtooth, triangle and square waves is shown in eq. (3.7), eq. (3.8) and eq. (3.9) where the sum is zero for $n = 0$. The number of harmonics contained in the waveform is determined by summation limits and fig. 3.4 illustrates evaluated Fourier series for sawtooth (a) and square wave (b) at increasing numbers of harmonics partials.

$$x_{\text{saw}}(t) = \sum_{n=-\infty,\, n\neq 0}^{\infty} -1^n \frac{e^{-i\,2\pi\,n\,t}}{n\pi} \tag{3.7}$$

$$x_{\text{square}}(t) = 2 \sum_{n=-\infty,\, n\neq 0}^{\infty} \frac{e^{-i\,2\pi\,(2n-1)\,t}}{n\pi} \tag{3.8}$$

$$x_{\text{triangle}}(t) = 4 \sum_{n=-\infty,\, n\neq 0}^{\infty} \frac{e^{-i\,2\pi\,(2n-1)\,t}}{(n\pi)^2} \tag{3.9}$$

3.3 Non-linearity of hearing

The intensity of a sound is perceived logarithmically by human hearing [WDJ97, p. 27]. Therefore, the *ratio* between two sound intensities is important and *not* the difference as in the case of linear perceived phenomenons. The ratio of two physical quantities, e.g. signal amplitudes, is measured in dB (decibel), a dimensionless logarithmic unit. Distinction should be made between the ratio of signal energy which is expressed by

$$10 \log_{10} \left(\frac{a}{b}\right) \tag{3.10}$$

and the ratio of signal power eq. (3.11).

$$20 \log_{10} \left(\frac{a}{b}\right) \tag{3.11}$$

with $a, b \in \mathbb{R}$. The difference in the scaling factors is based on the definition of signal energy where the square of the signals amplitude is taken, see eq. (3.12).

$$\int_{-\infty}^{\infty} |x(t)|^2 \, dt \tag{3.12}$$

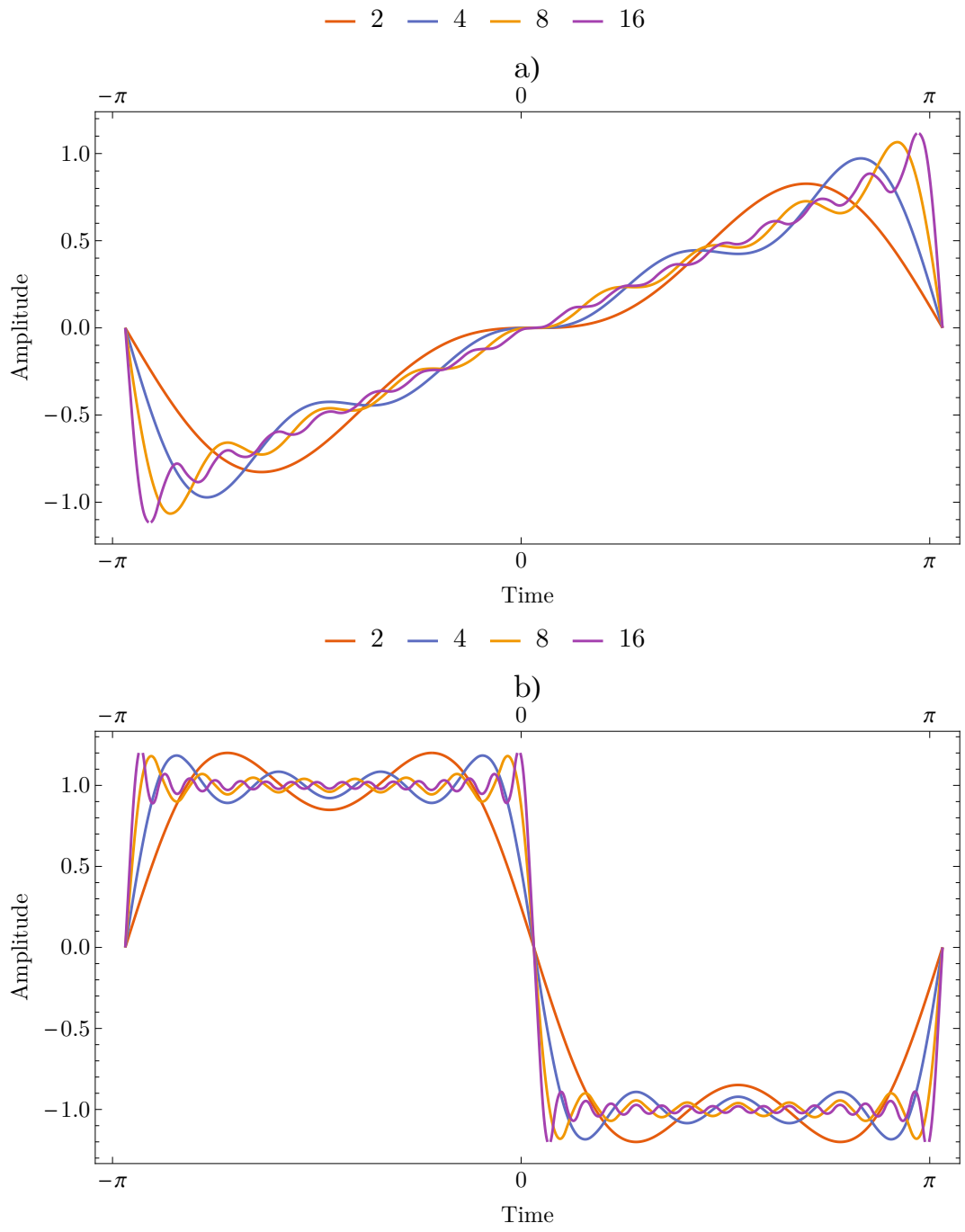

Figure 3.4: One cycle of bandlimited a) sawtooth and b) square waveforms with increasing number of harmonics.

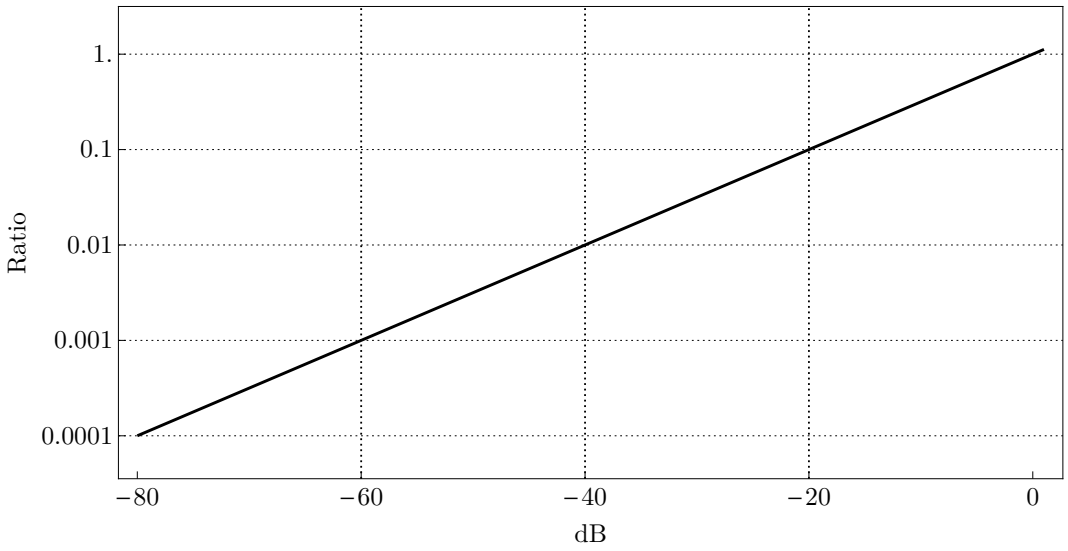

Figure 3.5: Relationship between dB and corresponding amplitude ratios on a logarithmically scaled y-axis.

Figure 3.5 illustrates the relationship between dB levels (x-axis) and corresponding amplitude ratios on a logarithmically scaled y-axis. The pressure levels audible by human ears range from 0.00002 $\mathrm{N\,m^{-2}}$ to 200 $\mathrm{N\,m^{-2}}$ [WDJ97, p. 26], which is seven orders of magnitude larger than the audibility threshold and clearly shows that logarithmic scale is better suited than a linear one to represent sound level ratios.

3.4 Envelope Generators

The sound produced by an musical instrument is usually not static and changes in amplitude or spectral content over time. To simulate these *time-varying* waveforms a function of time, the envelope generator, is used to controls parameters of an oscillator or other parts of a synthesizer's sound engine, e.g. the cutoff of a frequency filter. In fig. 3.6 the time-varying behavior of sounds produced by musical

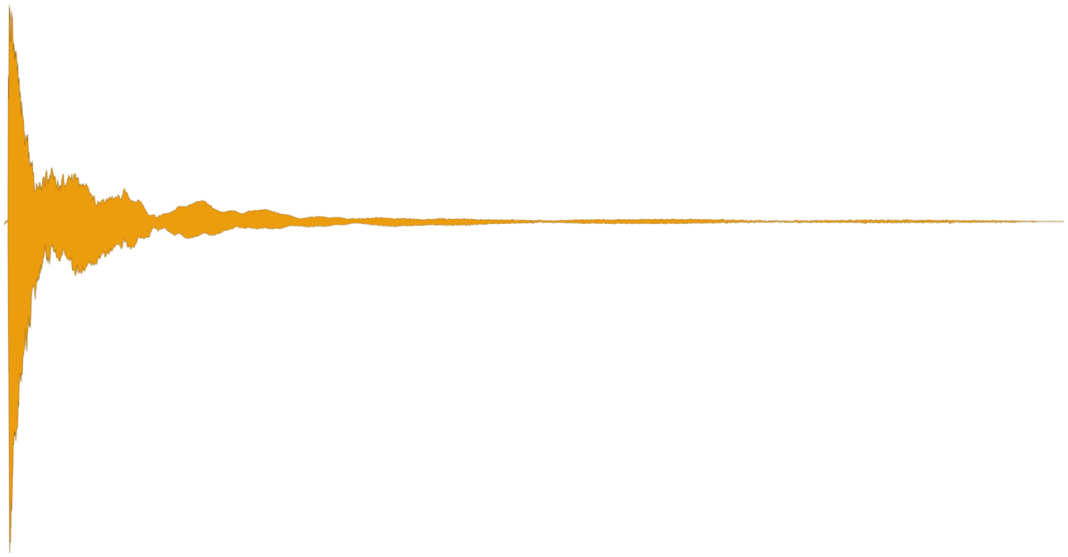

Figure 3.6: Waveform plot of sampled C_4 note played on a piano [13].

instruments is illustrated by an example of waveform plot of a C_4 note played on piano.

There are various types of envelope generators that range from simple two stage models, for fading the sound in and out, to ones which have an arbitrary number of stages and envelope shapes. A commonly used model with a reasonable amount of controllable parameters is the so called ADSR envelope, which stands for the four different stages of the envelope which are *Attack*, *Decay*, *Sustain* and *Release*. Because of the non-linearity of human hearing, as discussed in section 3.3, it is not sufficient to linearly ramp values between those four stages because this would not yield a smooth change in perceived loudness.

Puckette proposes three different amplitude envelope transfer functions [Puc06,

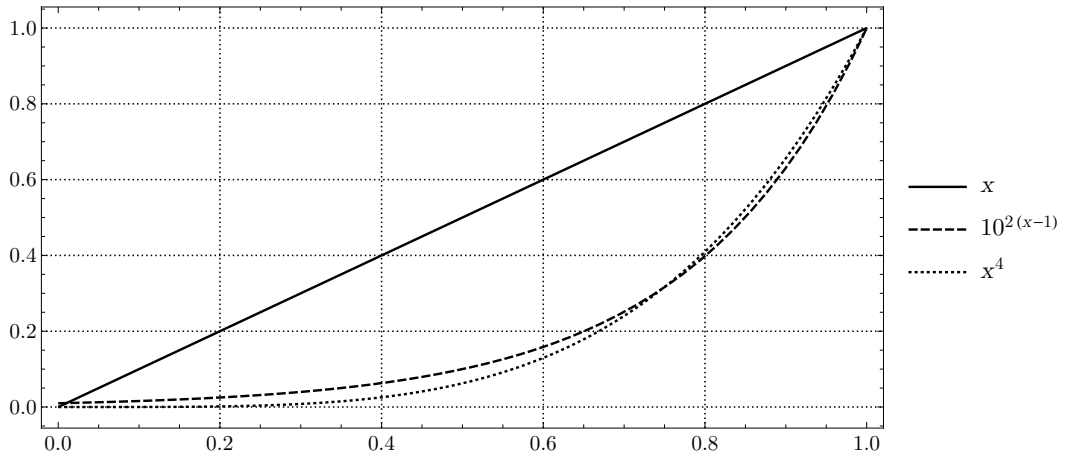

Figure 3.7: Three amplitude envelope transfer functions for input values in the range of [0, 1] as proposed by [Puc06, p. 94].

p. 94] (see fig. 3.7) where $10^{2(x-1)}$ converts from dB to linear and the quartic curve x^4 approximates the exponential dB curve while being computationally less expensive and reaching true zero at $x = 0$.

An ADSR generator's output is fully determined by five parameters, that are the output *level* and *duration* of the attack stage, decay duration, sustain level and duration of release. Figure 3.8 shows the output and stages for an envelope generator with exponential transfer function. The generator will start the output on an event like a key press and will reside in the sustain stage as long as the key is still pressed. If the key is released the generator will switch to the release stage independent of its state at the time of the event.

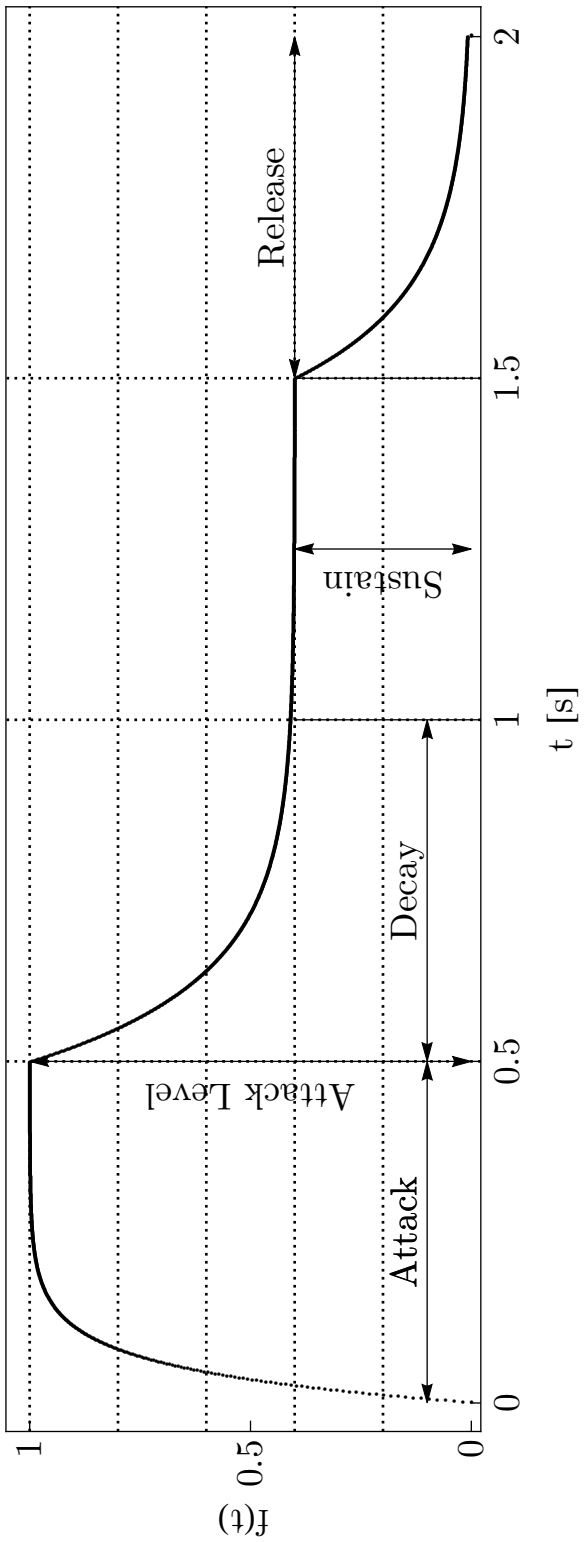

Figure 3.8: An ADSR envelope with equal duration of 0.5s for each stage and attack and sustain levels of 1 and 0.5.

4 Synthesis Techniques

Since the invention of the first electrical organ in 1894 [Roa96, p. 83], the *Telharmonium* built by Thaddeus Cahill, a lot of different synthesizing techniques have been developed. Those techniques can be roughly divided into two broad categories, techniques for *mimicking the sound of traditional instruments*, like

- Karplus-Strong synthesis [KS83], a simple technique for simulating plucked-string or drum sounds
- physical modeling synthesis which uses a mathematical model of an instrument to generate sounds

or techniques for generating arbitrary sounds, possibly not reproducible by a physical instrument, like *Additive*, *Subtractive* and *FM* synthesis. Developing a physical simulation of a traditional instrument is not the aim of this thesis, therefore the latter techniques will be described in this section.

4.1 Additive Synthesis

Additive synthesis is one of the oldest sound synthesizing techniques. It uses separate sinusoidal oscillators to generate a complex sound from its partials. As the name suggests, the output of each oscillator is added up to obtain the resulting

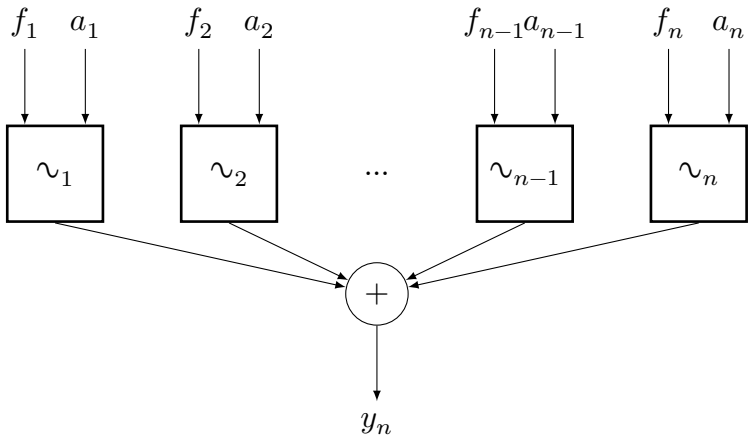

Figure 4.1: Basic structure of an additive synthesizer.

output signal. The basic structure of an additive synthesizer is shown in fig. 4.1, where $\sim_i$ denotes a sinusoidal oscillator with frequency input f_i and amplitude input a_i.

An advantage of additive synthesis is its great versatility, because virtually any sound can be synthesized, given a sufficient amount of oscillators. This comes with two major downsides:

1. this method is computationally expensive.
2. it is hard to control because there are at least as twice as many parameters as there are oscillators.

Additionally, to be able to simulate real or time-varying artificial sounds there must be functions that control those parameters over time, e.g. to reduce the amplitude of higher frequency partials when the sound decays. The coefficients obtained by Fourier analysis of a real sound (e.g. a sample of played key on a piano) can be used as parameters to reconstruct this sound through additive synthesis, this process is sometimes called *Fourier recomposition* [WDJ97, p. 88].

4.2 Frequency Modulation (FM) Synthesis

Frequency Modulation (FM) was originally used in telecommunications to encode information on a carrier wave by modulating the waves instantaneous frequency, e.g. for radio broadcast. In 1973, Chowning presented a new application of this well-known process to control spectral components of an audio signal with great simplicity [Cho73, p. 1]. Contrary to its well-understood use for radio transmission, both the *carrier* and the *modulating frequency* are inside the audio band. The audio spectrum is formed by the carrier wave and side frequencies which are introduced through frequency modulation. Modulation of the carrier wave is determined by two factors:

- the *frequency of the modulating wave* m_f sets rate at which the instantaneous frequency of the carrier varies.
- the *amount of modulation* m_a which is equal to the modulating waves amplitude.

If both the carrier as well as the modulator, are sinusoids then the instantaneous frequency maybe be calculated as follows [Cho73, p. 2]:

$$y(t) = A \sin\left(c_f t + I \sin(m_f t)\right) \tag{4.1}$$

where A is the *peak amplitude*, c_f is the carrier wave's frequency and $I = m_a/m_f$ is the ratio of modulation amount to modulation frequency also called *modulation index*. A table of waveforms generated by different values of m_f and m_a is shown in fig. 4.2.

For $I = 0$ there is no modulation, but non-zero values will result in frequencies

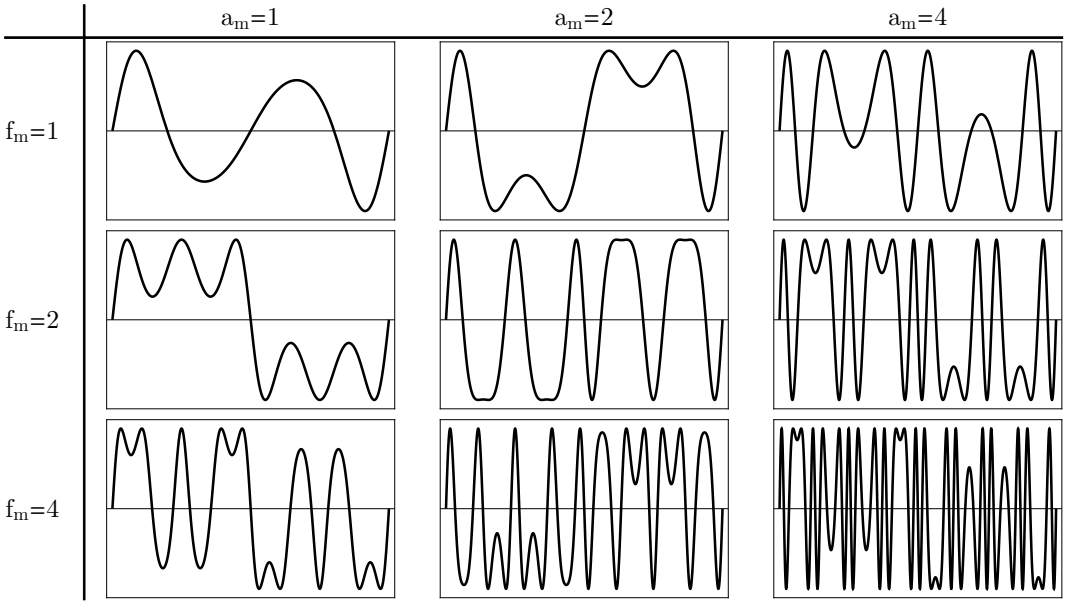

Figure 4.2: One cycle of a FM modulated sine wave for different modulation intensities a_m and modulator frequencies f_m.

occurring below and above the carrier frequency at intervals of the modulating frequency. Chowning describes the relation of modulation index and occurrence of side frequencies like this [Cho73, p. 2]:

> *The number of side frequencies which occur is related to the modulation index in such a way that as I increases from zero, energy is "stolen" from the carrier and distributed among an increasing number of side frequencies.*

This behavior is shown in fig. 4.3 for different modulation indices by constant modulation and carrier frequency. Negative amplitudes for frequency components indicate *phase inversion*[1].

Specific carrier and modulation frequency ratios and modulation index values will

[1] $-\sin(\phi) = \sin(-\phi)$

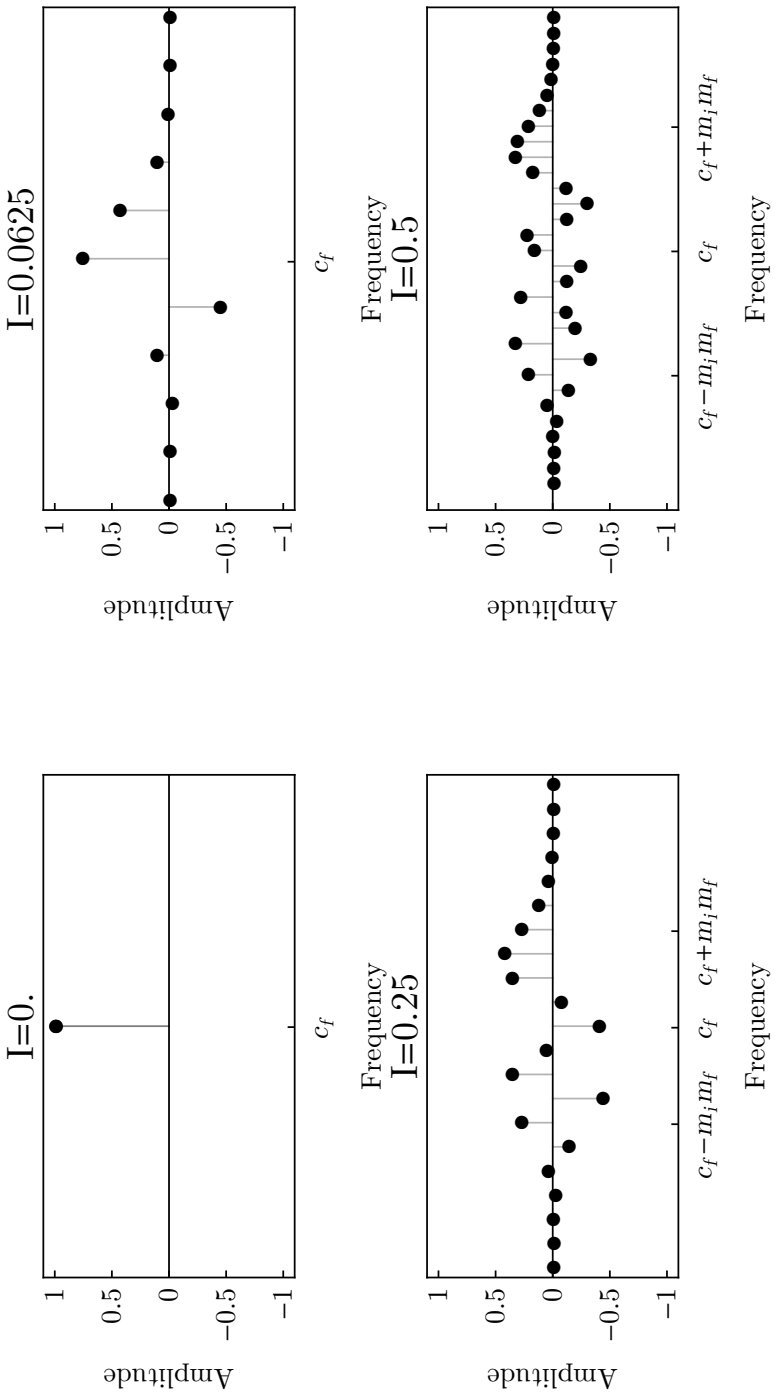

Figure 4.3: Magnitude spectra for frequency modulated carrier frequency c_f at different modulation indices I and constant modulation frequency m_f. Bandwidth increases symmetrically around c_f with I.

produce sideband frequencies that fall into the negative spectrum. Those negative frequency components will be reflected (aliased) around 0 Hz. Reflected sideband components will either increase or—if they are phase inverted—decrease the energy in the spectrum.

Harmonic spectra[2] will be generated if the ratio of carrier and modulation frequency is a rational number. Ratios that are irrational numbers, e.g. $1/\sqrt{2}$, will result in inharmonic spectra because the reflected sideband frequencies will fall inbetween the positive frequency components.

Carrier and sideband component amplitudes can be determined analytically by evaluating n-th order Bessel functions J_i of the first kind with the modulation index as argument. A quick estimation for the resulting bandwidth of different modulation indices is shown in fig. 4.4 by evaluating Bessel functions J_0 through J_{15} for those indices.

The most basic FM synthesizer *algorithm* consists of two *operators* (which are just oscillators in FM terminology), a modulator and a carrier, where the modulators output is summed with the carriers fundamental frequency. An FM algorithm is described by how its operators are connected among each other. Figure 4.5 shows the structure of the most basic FM algorithm, a simple pair of operators.

FM's advantages lie in the simplicity of control, the small computational effort that is required and the great amount of flexibility, resulting from arranging operators in different algorithms. On the other hand, FM synthesis is likely to introduce undesirable aliasing of higher frequencies which should be taken into account when implementing the algorithm.

[2] Overtones are an integer multiple of waves the fundamental frequency.

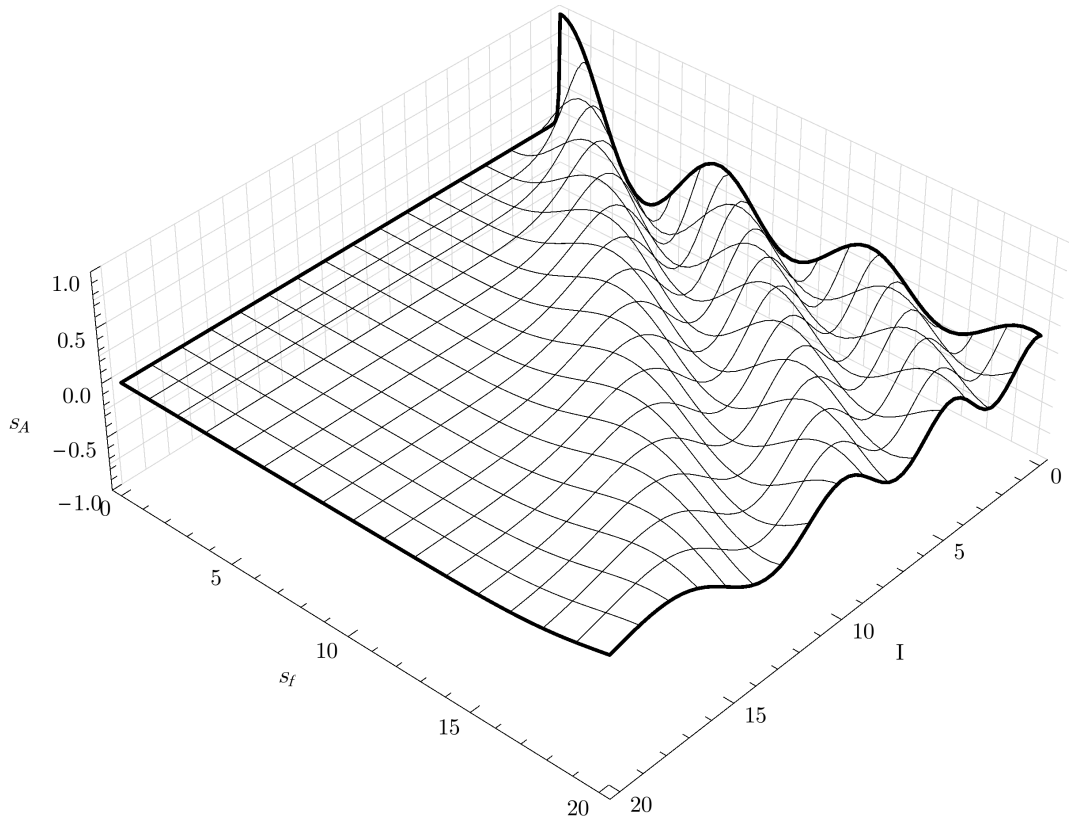

Figure 4.4: Bandwidth estimation for modulation indices I ranging from 0 through 20 by evaluating Bessel functions J_0 through J_{15} showing resulting sideband frequencies s_f and amplitudes s_A [Cho73, p. 5].

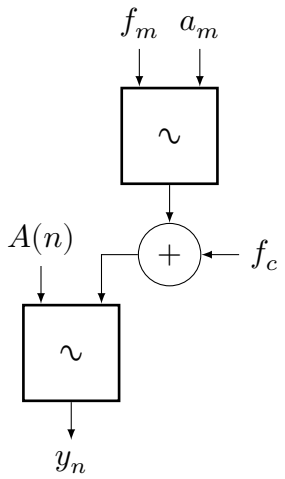

Figure 4.5: Most basic FM algorithm, a pair of operators with one modulator and carrier.

4.3 Subtractive Synthesis

Subtractive Synthesis is—like the name implies—the opposite of *Additive Synthesis* and creates musical tones by *removing* parts of the frequency spectrum of a source signal. A *filter* thereby amplifies or attenuates regions of the spectrum while the source signal passes through. Spectrally rich signals like noise, sawtooth or square waves are well suited for the use as sound sources. The following section introduces filters by means of audio signal processing, describes commonly used filter types in musical synthesizers and discusses FIR and IIR filters.

5 Digital Filters

Filter is a broad term that has many different and often very general definitions. Such a general definition is given by Smith III in [Smi85]:

> *Any medium through which the music signal passes, whatever its form, can be regarded as a filter.*

Surprisingly, even "Terminology in Digital Signal Processing" [Rab+72] uses the term without prior specification. In this thesis a more specific definition will be used [PM07, p. 326]:

> *The linear time-invariant system, through its frequency response function, attenuates some frequency components of the input signal and amplifies other frequency components. Thus the system acts as a filter to the input signal.*

This is analog to the description given in section 4.3, despite the terms *linear time-invariant* (LTI) system and *frequency response function* not having been explained.

5.1 Linear Time-Invariant Systems

LTI systems are used as filters because "no new spectral components are introduced" [Smi16] by them. The time-invariance property is not overly restrictive because it also holds for filters that change slowly over time. This is a very convenient property because, if musicians were not allowed to change parameters of a subtractive synthesizer's filter while playing, the result would be static and uninteresting sounds.

5.1.1 Linearity and Time-Invariance

A system is *linear* if the superposition principle eq. (5.1) holds.

$$F[a_1 x_1(n) + a_2 x_2(n)] = a_1 F[x_1(n)] + a_2 F[x_2(n)] \tag{5.1}$$

In other words, the response of system F applied to two (or more) stimuli $x_{1,2}$ is equal to the sum of responses of the system applied to each stimulus individually, for any real valued scalars $a_{1,2}$ and points in time n. This also shows the scaling the property of linear systems, i.e. scaling of a systems input results in an identical scaling of the response. A system is *time invariant*, if

$$F[x(n-k)] = y(n-k), \tag{5.2}$$

for any time shift k. Thus, the response of the system applied to a stimulus delayed by k units of time is equal to the systems response delayed for the same amount. Hence, if a system obeys both properties, linearity and time-invariance, then it is

called an LTI system. Such a system is characterized completely by its *impulse* or *frequency response*.

5.1.2 LTI filters

Linear time-invariant digital filters (systems), in the following simply called *filters*, may be written as *difference equation*

$$y[n] = \sum_{i=0}^{M} b_i x[n-i] - \sum_{j=1}^{N} a_j y[n-j] \tag{5.3}$$

where x denotes the input signal[1], y the output signal, and the filter's *coefficients* are the constants a_j and b_i. A signal therefore is a *sequence* of real numbers denoted as a function of integer index $x[n]$ where n denotes the *n*-th sample. Coefficients a_j, b_i must be in $\mathbb{R}$ to obtain a *real valued filter* that has a real valued output for any given real valued input signal. Another requirement of the filter is to be *causal*, i.e. it does not depend on future values and only uses past input and output values to calculate its current output value, otherwise the filter can not be realized. A filters *order* is the maximum sample delay used ($\max(M, N)$ in eq. (5.3)) and in general the higher a filters order the steeper its transition slope.

Another way of representing a digital filter is by its rational system- or *transfer function* $H(z)$ in the z-domain as shown for a *causal* filter in eq. (5.4) where $z = Ae^{i\phi}$ is some complex exponential with amplitude A and phase ϕ that acts as time-shift of j samples. This z-domain representation will be required when designing a digital IIR filters based on an analog prototype.

[1] The term sequence and signal will be used interchangeably.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}} \tag{5.4}$$

The z-domain representation of a discrete-time signal $x[n]$ is defined as the bilateral transform:

$$X(z) = \mathcal{Z}\{x[n]\} = \sum_{n=-\infty}^{\infty} x[n] z^{-n}. \tag{5.5}$$

LTI filters can be divided into two types, first feedforward or *finite impulse response* (FIR) filters which only use previous input values (a_js are zero) and second feedback or *infinite impulse response* (IIR) filters that also use previous output values to calculate the present filter output $y[n]$.

5.2 Impulse Response

Another way of representing a LTI system in the time domain is its response to a signal impulse, called the systems *impulse response*.

> *[A one-sample impulse] contains energy at all frequencies that can be represented at the given sampling frequency. Hence, a general way of characterizing a filter is to view its response to a one-sample pulse[...] [Roa96, p. 400].*

The hereby used input signal is the Kronecker delta function $\delta(n)$ eq. (5.6) which is one if $n = 0$ and zero otherwise.

$$\delta(n) = \begin{cases} 1, & n = 0, \\ 0, & otherwise \end{cases} \tag{5.6}$$

Applying the LTI system on the impulse signal $\delta(n)$ yields the systems impulse response denoted as $h[n]$. If the impulse response does not reach zero over time the system is said to be *unstable*. Any LTI system is fully described by its impulse response. *Convolving* an input signal $x[n]$ with the impulse response $h[n]$ yields the systems time-domain output for that signal [PM07, p. 73]:

$$y[n] = x * h = \sum_{k=-\infty}^{\infty} x[k]h[n-k] = \sum_{k=-\infty}^{\infty} x[n-k]h[n] \tag{5.7}$$

Clearly, convolution $*$ is a commutative operation. This property can be used to optimize possible implementations [Pav13].

5.3 Frequency Response

Evaluating the systems transfer function eq. (5.4) on the unit circle, i.e. set z to $e^{i\omega T}$ where T is the sampling interval, yields the LTI systems frequency response which is the frequency spectrum of the output divided by the frequency spectrum of the input.

It is easy to show that evaluating the (bilateral) z-transform on the unit-circle will find the spectrum because setting $z = e^{i\omega T}$ in eq. (5.5) results in the definition of the bilateral discrete-time Fourier transform:

$$X\left(e^{i\omega T}\right) = \sum_{n=-\infty}^{\infty} x[n]e^{-i\omega nT} \tag{5.8}$$

.

In the following only *causal* sequences are of interest, thus the *unilateral* versions of z-transform and discrete-time Fourier transform are used in which the summation index starts at $n = 0$. Also, the sampling index T is set to 1 for simplicity. Another way of obtaining an LTI systems frequency response is by applying the Fourier transform on the systems impulse response $h(z)$ [PM07, p. 301]:

$$H(\omega) = \sum_{n=-\infty}^{\infty} h[n]e^{-i\omega n} \tag{5.9}$$

It is sufficient to evaluate the frequency response function only for $\omega \in [-\pi, \pi)$ because all frequencies are mapped into a single cycle of the unit circle and the spectrum would repeat for additional cycles anyway[2].

5.3.1 Magnitude- and Phase Response

The complex valued result of the frequency response function can be decomposed into two real valued functions [Smi16] the systems *magnitude response*[3] $|H(\omega)|$ and its *phase response* $\angle H(\omega)$, where $\angle$ denotes the complex argument.

The magnitude response of a filter shows how frequencies are attenuated or amplified and the phase response specifies the phase-shift experienced by each frequency. In general the frequency response is of more interest because it is better

[2] see aliasing section 3.1.1

[3] Sometimes improperly called *amplitude* response because amplitudes can be negative.

suited for characterizing a filter but the phase response should not be completely ignored.

Chamberlin states that "Poor phase response in a filter also means poor transient response" and this effect will become worse with increasing filter order [Cha85, p. 392], i.e. sharp changes in a waveform (transients) will be smoothed by a filter with poor phase response. However, even a poor filter phase response is quite good compared to the phase error introduced by the audio speaker while transforming the signal from an electrical to an acoustical one [Cha85, p. 392].

5.4 Filter Classification

Filters of a musical synthesizer are classified by the magnitude curve of their frequency response function $|H(\omega)|$ which is the filter's characteristic *frequency response* curve. Exemplary frequency response curves for lowpass, highpass, bandpass and notch (sometimes called bandreject or bandstop) filters are shown in fig. 5.1.

Low- and highpass filters cut all frequencies below, respectively above of the *cutoff frequency* f_c, while bandpass and notch filters let frequencies in a certain range (*frequency band*) pass through or rejecting them. The width of the pass- or stopband is an additional property of those last two filter types and the difference between their high and low cutoff frequencies is called *bandwidth*. Correspondingly, the center of the pass- or stopband—the point of maximum or minimum amplitude in this band—is the fiter's *center frequency*.

A filters cutoff frequency is commonly specified for the half-power point [Rab+72, p. 8] where the filter reduces the signals energy to $1/\sqrt{2} \approx 0.707$ or in terms of

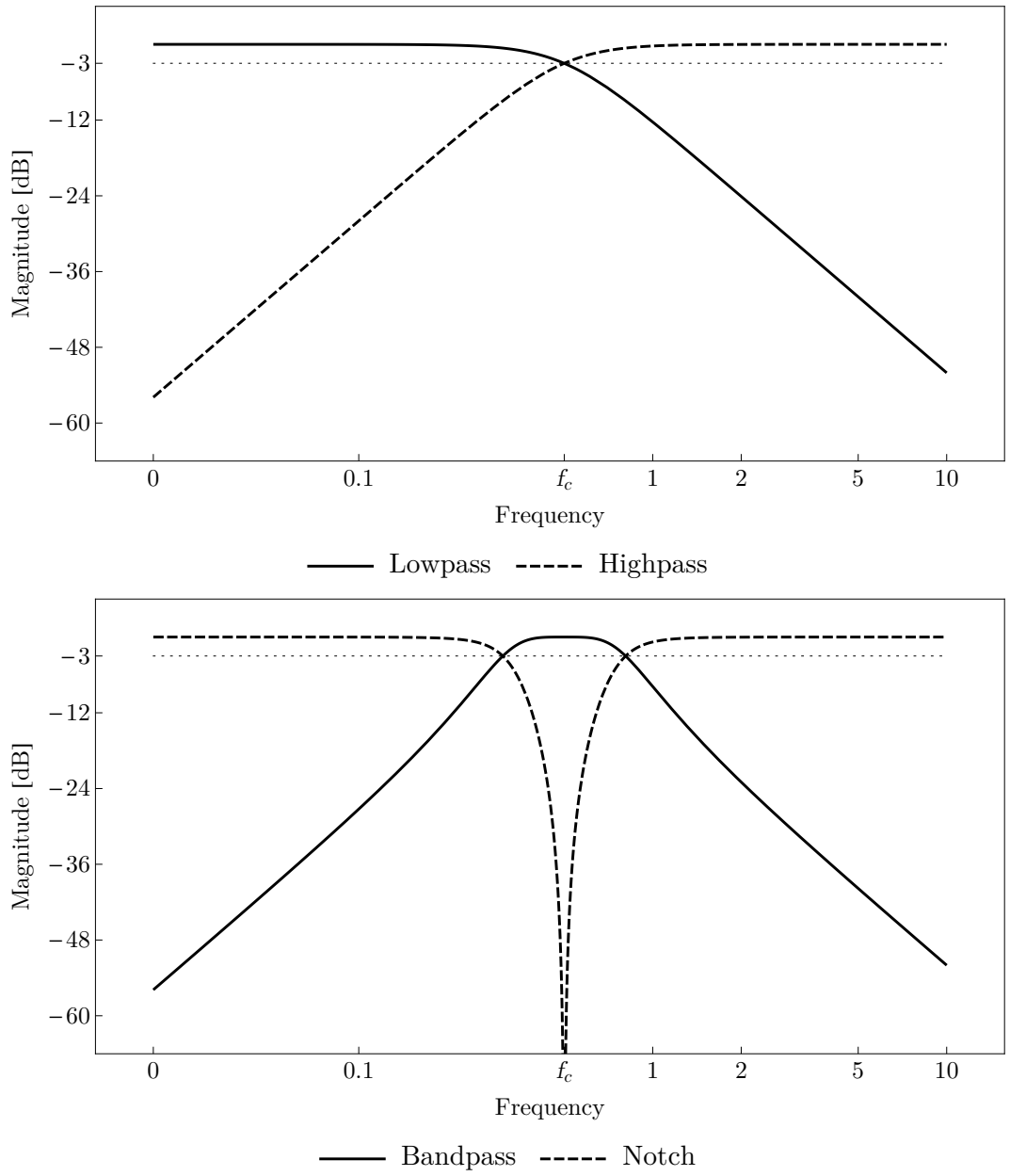

Figure 5.1: Log-log plots for exemplary frequency response curves of four elementary filter types with 12 dB/octave roll-off where f_c denotes the cutoff frequency at the half-power point (−3 dB) shown as a dotted line.

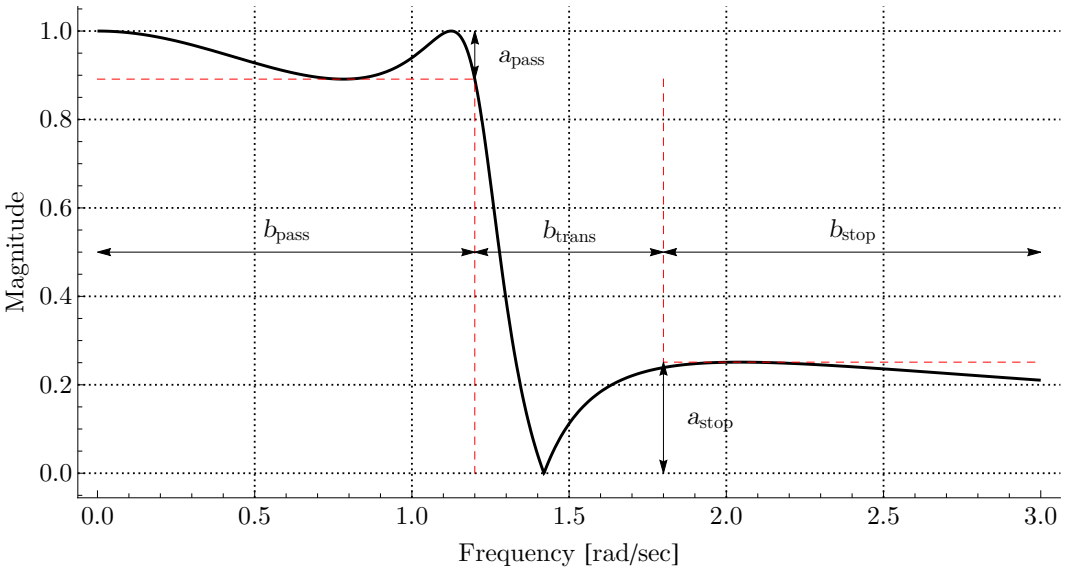

Figure 5.2: Terminology for describing the frequency response of a low-pass filter.

signal power $(1/\sqrt{2})^2 = 1/2$ or -3 dB (see eq. (3.11)).

An ideal filter would have a sharp cut between the pass- and stopband that looks like a rectangle in the frequency response but such a filter is not realizable because it would be of infinite order. Hence, there is a transition between the pass- and stopband called *transition band*. Figure 5.2 shows those bands and their respective bandwidths b_{pass}, b_{trans}, b_{stop} as well as other constraints like pass- and stopband a_{pass} and a_{stop} that must be specified and taken into account when designing a filter.

The steepness of the frequency response curve in the transition band is measured in dB/octave where a larger value implies an increased steepness of the curve, e.g. a *roll-off* of 12 dB/octave for a lowpass filter means that the amplitude is reduced by 12 dB for each doubling in frequency above f_c.

Depending on the curve of the phase reponse a filter is said to have *zero phase*

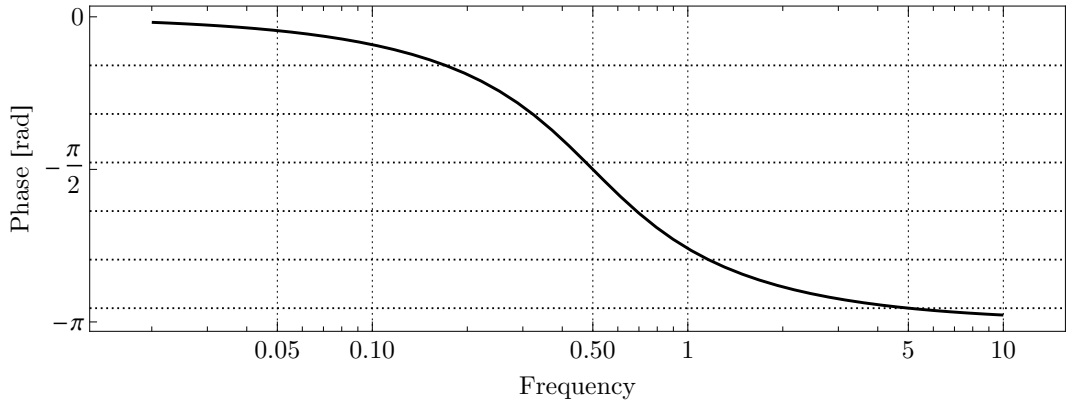

Figure 5.3: Non-linear phase response ($\angle H(\omega)$) of a second-order Butterworth lowpass filter.

if the phase-shift is constant over all frequencies, *linear phase* if there is a linear relationship between phase-shift and frequency and *non-linear phase* otherwise. The phase response for the lowpass filter used in fig. 5.1 is shown in fig. 5.3

5.5 FIR Filters

A FIR filter's response to an impulse will die away after a finite period of time [Roa96, p. 406], hence the name *finite* impulse response filter. The filter's structure is simply the sum of delayed and weighted samples where the weights for each delay are the coefficients a_j. As described in section 5.1.2, the filters order is equal to the order of its transfer function polynomial, i.e. the total number of unit-sample delays it uses. There are various methods for FIR filter design, e.g. window design method, frequency sampling or equiripple method [PM07, p.664-690] and constraint-based linear programming algorithms like METEOR [Ste+92].

The general equation for a finite impulse response filter is equal to eq. (5.3) when

all recursive coefficients a_k are zero:

$$y[n] = \sum_{k=0}^{M} b_k x[n-i]. \tag{5.10}$$

Thus, the denominator of a FIR filter's transfer function is one, hence its polynomial

$$H(z) = \sum_{k=0}^{M} b_k z^{-k} \tag{5.11}$$

has only zeroes but no poles. This implies that FIR filters are *always stable*, i.e. a *bounded* (finite) input always results in a bounded output. The transfer function eq. (5.11) may be written in factored form where each complex zero q of the polynomial can be directly seen:

$$H(z) = (1 - q_1\ z^{-1})(1 - q_2\ z^{-1}) \cdot \ldots \cdot (1 - q_M\ z^{-1}) \tag{5.12}$$

There can be less than M factors if some of them cancel out.

If z has the value of one of these factors q then the transfer function evaluates to zero. The positions of the zeroes in the complex z-plane of a 12-th order lowpass FIR filter are shown in fig. 5.4.

By the complex zero's angle from the z-plane's origin is determined which frequencies are effected from it and its distance to the unit circle, on which the frequency response is evaluated, determines how large the attenuation is. Frequencies are mapped on the unit circle counterclockwise starting from 0 at $(1, 0)$ and going to π, which translates to a frequency limit like the Nyquist frequency, at $(-1, 0)$

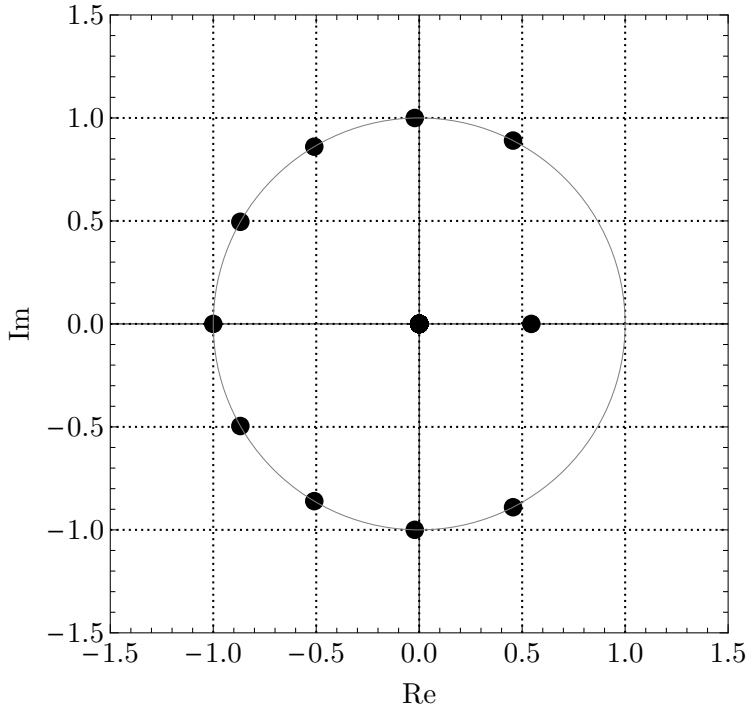

Figure 5.4: Zeroes of 12-th order lowpass FIR filter with cutoff frequency $f_c = \pi/4$.

where positive frequencies are carried out in the upper half of the unit circle and negative frequencies in the lower half. It can also be seen that complex zeroes $q = a + i\,b$ come in pairs; if they are not laying on the some point; where one of them is conjugate transposed $q^* = a - i\,b$, so that the imaginary parts cancel each other out to obtain a real valued filter response.

5.6 IIR Filters

Infinite Impulse Response filters additionally use previous output values to calculate the recent result. Therefore, an IIR's rational transfer function contains feedback coefficients a_k eq. (5.4) and can be viewed as $H(z) = H_1(z)H_2(z)$ [PM07,

p. 583], where $H_1(z)$ consists of the zeroes of $H(z)$ and

$$H_2(z) = \frac{1}{1 + \sum_{k=0}^{N} a_k z^{-k}} \tag{5.13}$$

consists of the poles of $H(z)$ [PM07, p. 583]. In contrast to FIR filters an impulse fed into an IIR filter will cause an infinite response—numerical quantization error ignored—because the transfer function will never truly reach zero due to the use feedback values, therefore the name *Infinite Impulse Response* filter.

IIR filters can become *unstable* because they can have poles outside of the complex plane's origin, e.g. if there is a pole on the unit circle at frequency ω_p then $H(z)$ would become ∞ when evaluated on the unit circle ($z = e^{i\omega}$) for ω_p.

An advantage of the use of feedback values is that IIR filters can achieve much steeper transition band slopes than FIR filters of the same order, thus they are more efficient in the number of arithmetic operations and memory required.

The effect of transfer function poles in the z-plane is the opposite of zeros, i.e. the distance to the unit circle determines how much a frequency is amplified by the pole. Musical synthesizers often allow the user to specify a *resonance* parameter for the filter cutoff frequency, where a low resonance results in smooth transition transition from the pass- to the transition band whereas a high value creates a peak at the cutoff frequency as shown for different resonance values in the magnitude response of a IIR second-order lowpass filter in fig. 5.5. Such a resonance parameter is easy to realize for IIR filters.

IIR filters are very sensitive to numerical rounding error, where the sensitivity depends on their order and implementation structure section 5.7. Therefore, second-order systems, so called *biquads*, are used as building blocks for higher-order filters

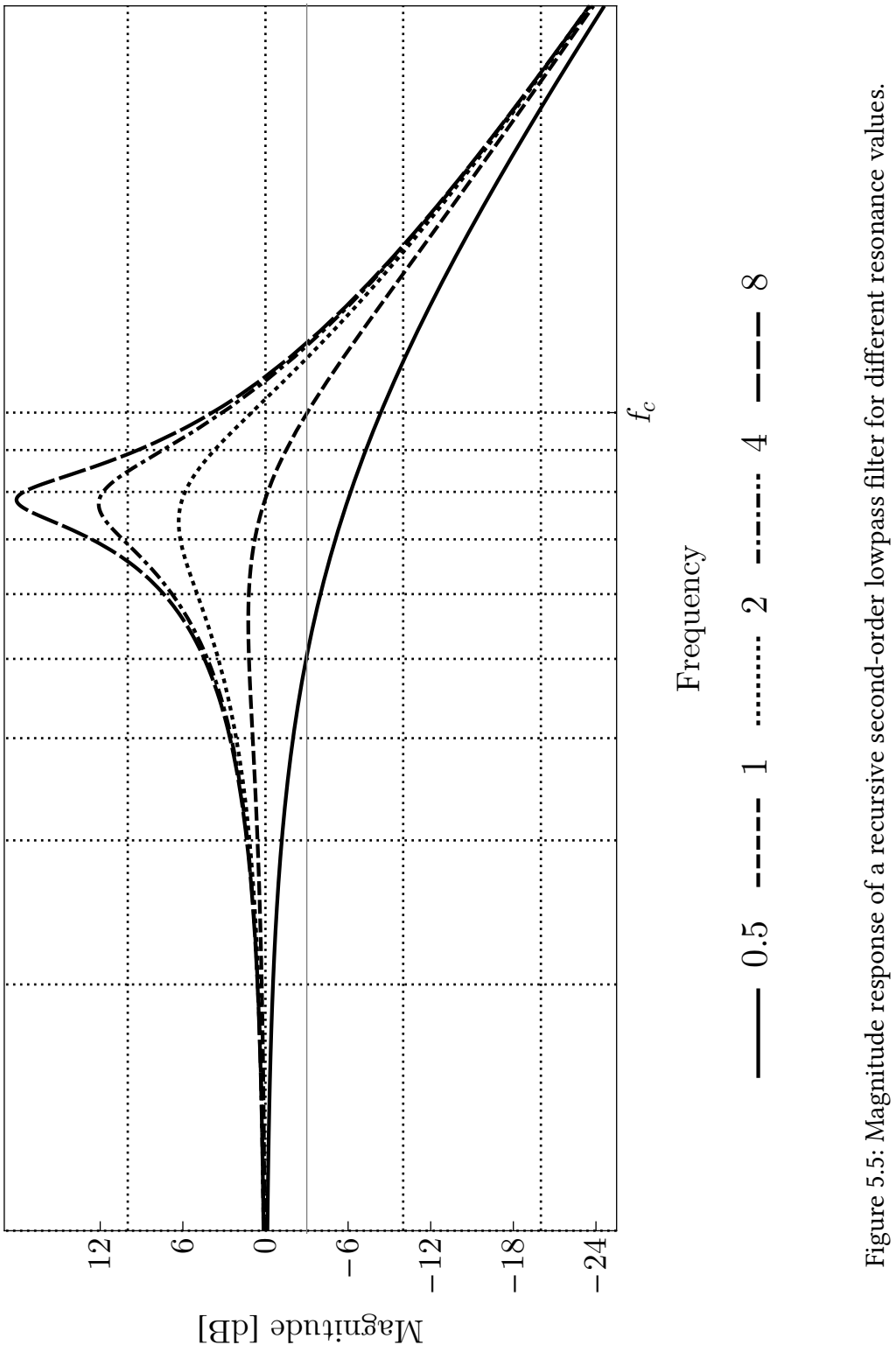

Figure 5.5: Magnitude response of a recursive second-order lowpass filter for different resonance values.

(order $>= 4$) because the filters sensitiviy to quantization increases with its order [PM07, p. 132,589]. Filters of odd-order a constructed as separate biquad and a single-order system whereas even-order filters are constructed solely from biquads.

Analog IIR filter design has a long history and is therefore a well researched and understood topic. There are a number of commonly used analog filters with different characteristics [PM07, p. 717pp]:

- *Butterworth* all-pole filters with monotonic frequency magnitude response in both pass- and stopband.
- *Chebyshev Type I* all-pole filters with equiripple behavior in the passband and monotonic characteristic in the stopband.
- *Chebyshev Type II* filters are like *Type I* except that they have monotonic passband characteristic and equiripple stopband behavior.
- *Elliptic Filters* with equiripple behavior in both pass- and stopband.

The filter design technique used in this thesis is to take an analog prototype filter and transform it into a discrete-time filter by using the *Bilinear transform*. Other design techniques like *approximation of derivates* or *design by impulse variance* have the limitation that they are only valid for a limited set of filter classes [PM07, p. 712].

5.6.1 Bilinear Transform

The Bilinear transform, defined by eq. (5.14) for sampling interval T, is used to convert a transfer function of a continious-time LTI filter transfer function $H_a(s)$ into a transfer function of a discrete-time LTI filter $H(z)$, where $H_a(s)$ defined in

the s-domain with $s = \sigma + i\Omega$.

$$s = \frac{2}{T}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right) \tag{5.14}$$

A continuous time filter defined in the s-plane is stable if all of its poles are located in the left-semi plane which is mapped by the bilinear transform into the unit circle, hence the transformed discrete-time filter is stable if all of its poles are located inside the unit circle of the z-domain.

The frequency relationship of the bilinear transform is non-linear, thus continuous time frequencies $\Omega \in [-\infty, \infty]$ are mapped from the $i\Omega$ axis of the s-plane into digital frequencies $\omega \in [-\pi, \pi)$ on the unit-circle by the following transformation (also called *frequency warping*)

$$\omega = \frac{2}{T} \arctan\left(\frac{\Omega T}{2}\right), \tag{5.15}$$

and the inverse transformation is given by

$$\Omega = \frac{2}{T} \tan\left(\omega \frac{T}{2}\right). \tag{5.16}$$

5.6.2 Bilinear Transform Example

The following steps illustrate the general procedure of designing a filter using bilinear transform at the example of a second-order Butterworth lowpass filter with cutoff frequency $f_c = 4000\,\text{Hz}$ for a sampling rate of $f_s = 48\,\text{kHz}$, hence sampling interval $T = 1/f_s$:

- Pre-warp the critical frequencies, in this case the filter's cutoff frequency $\omega_c = 2\pi f_c \text{rad}\,\text{s}^{-1}$:

$$\Omega_c = \frac{2}{T}\tan\left(\omega_c \frac{T}{2}\right) \approx \frac{2}{T} 0.268 \text{rad}\,\text{s}^{-1}$$

- Set the critical frequency ω_c and apply the bilinear transformation eq. (5.14) to obtain $H(z)$ from the analog transfer function $H_a(s)$:

$$H_a(s) = \frac{\Omega_c^2}{s^2 + s\sqrt{2}\Omega_c + \Omega_c^2} \tag{5.17}$$

$$H(z) = \frac{\left(\frac{2}{T}0.268\right)^2}{\left(\frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}\right)^2 + \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}\sqrt{2}\frac{2}{T}0.268 + \left(\frac{2}{T}0.268\right)^2} \tag{5.18}$$

$$= \frac{\left(\frac{2}{T}\right)^2 0.268^2}{\left(\frac{2}{T}\right)^2\left(\frac{1-z^{-1}}{1+z^{-1}}\right)^2 + \left(\frac{2}{T}\right)^2\frac{1-z^{-1}}{1+z^{-1}}\sqrt{2}\cdot 0.268 + \left(\frac{2}{T}\right)^2 0.268^2} \tag{5.19}$$

$$= \frac{0.268^2}{\left(\frac{1-z^{-1}}{1+z^{-1}}\right)^2 + \frac{1-z^{-1}}{1+z^{-1}}\cdot 0.379 + 0.268^2} \tag{5.20}$$

$$= \frac{0.0495(1+z)^2}{0.4775 - 1.2795z + z^2}\cdot\frac{z^{-2}}{z^{-2}} \tag{5.21}$$

$$= 0.0495\cdot\frac{1+2z^{-1}+z^{-2}}{1-1.2795z^{-1}+0.4775z^{-2}} \tag{5.22}$$

- $H(z)$ is evaluated on the unit circle to check the magnitude frequency response fig. 5.6 which shows the that the cutoff frequency lays exactly on the half-power point

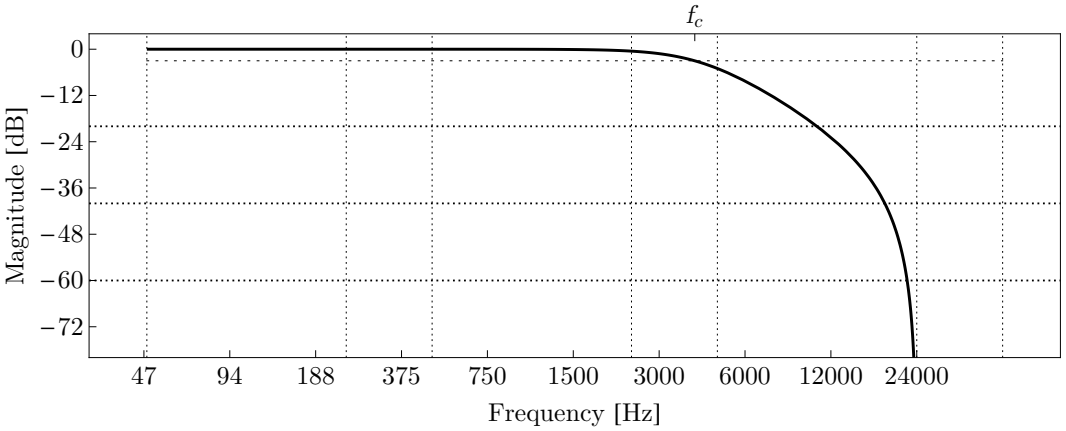

Figure 5.6: Magnitude frequency response of $H(z)$ eq. (5.22) showing cutoff frequency f_c at half-power point (dotted line).

- The pole-zero diagram fig. 5.7 of $H(z)$ shows that the pair of poles is located inside the unit circle, hence the filter is real-valued and stable. Also, a pair of zeros is located at the maximum frequency point which gives the lowpass characteristic.

- Lastly, the filters difference equation eq. (5.3) can be derived directly from $H(z)$: $y[n] = 0.0495x[n] + 0.099x[n-1] + 0.0495x[n-2] + 1.2795y[n-1] - 0.4775y[n-2]$.

5.7 Implementation Structures for IIR Filters

An recursive LTI filter given as difference equation may be implemented as one of four *direct-form* (DF) filter implementations. The direct-form is another way of representing a filter, besides impulse response and difference equation, with the benefit of directly representing its implementation structure.

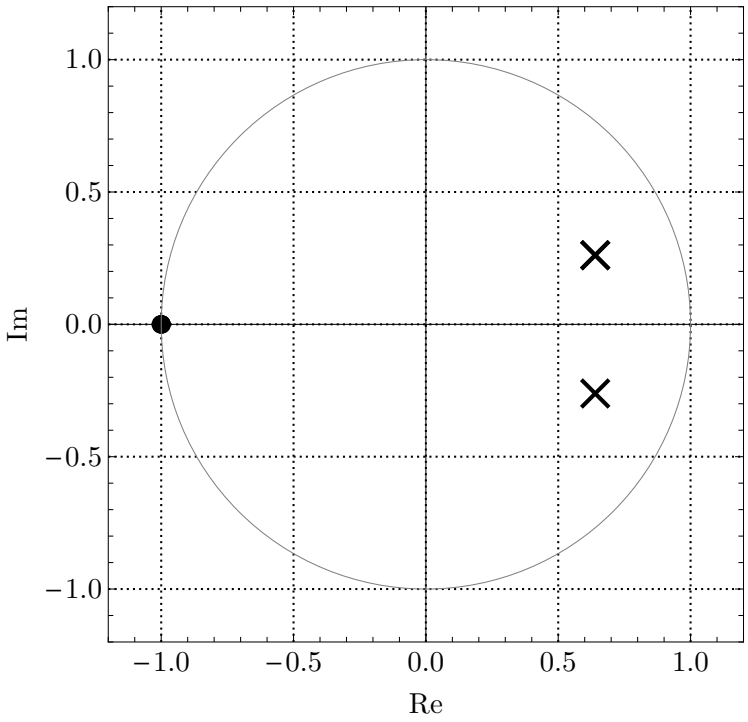

Figure 5.7: Pole-zero diagram of $H(z)$ eq. (5.22) with pole and zero positions marked as $\times$, respectively $\bullet$.

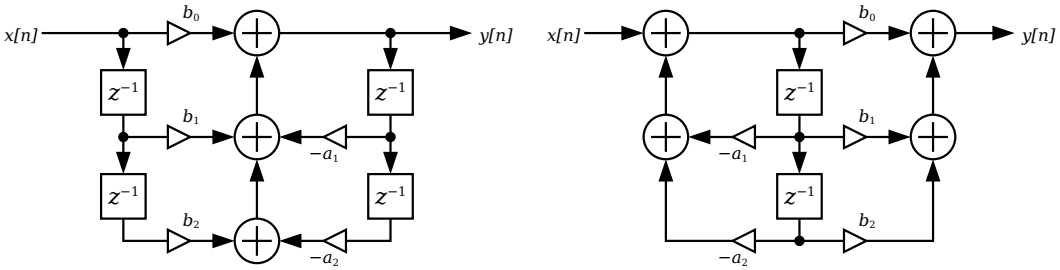

Figure 5.8: Direct-Form I and II implementation of a second-order filter with the normalized (divided by a_0) difference equation $y[n] = b_0x[n]+b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2]$ [Fou16].

The four direct-form structures are DF-I and DF-II, shown in fig. 5.8 where z^{-1} denotes a unit-sample delay, and their *transposed* counterparts. Transposed forms can be obtained from their DF-I or DF-II forms by reversing the signal path directions, replacing sums with branch-points and vice versa. This operations does not effect the filters transfer function.

The technical properties of all four forms are different, despite that they represent the same transfer function. A DF-I implementation saves addition operations at the cost of requiring twice as many delays (memory) as necessary while a DF-II implementation saves memory by sharing delays at the cost of possible fixed-point arithmetic overflow [Smi16]. Transposed forms, TDF-I and TDF-II, have enhanced numerical robustness while obeying the same advantages and disadvantages of their fundamental structure. An additional advantage of TDF-II structures is that they perform well in applications where the filter parameters change in audio rate, in other words as implementation for time-varying filters [Wis14].

Table 5.1: Comparison of FIR against IIR filters.

	FIR	IIR
Impulse Response	finite	infinite
Magnitude response	arbitrary responses are easy to design, e.g. *frequency sampling technique* [PM07, p. 671]	often based on analog prototypes, arbitrary responses are hard to achieve
Stability	always stable	feedback coeffecients can cause instability
Efficiency	more memory and operations	less memory and operations
Linear phase	always possible	no technique available

5.8 Comparison FIR against IIR

A comparison between FIR and IIR filter design techniques is given by table 5.1. For the synthesizer's filter an IIR design was chosen because of two requirements, first high computational efficiency to achieve short latencies, and an easy to implement parameter for controlling filter resonance. Additionally, a Butterworth characteristic was picked because the phase response of Elliptic filters is more nonlinear and the equiripple behavior in pass- and/or stopband of Elliptic and Chebyshev characteristics is unfavorable. It is not of great disadvantage for a musical synthesizers filter that a Butterworth filter rolls of more slowly at the cutoff frequency, because very sharp transition band steepness is not required.

6 Oscillators and Waveform Synthesis

An oscillator is one of a synthesizer's fundamental building blocks because it is the source signal from which the desired sound is modeled from. Typically, one or more oscillators are used as a signal source. Subtractive synthesizers require spectrally rich source signals. Therefore, oscillators must be able to generate a variety of waveforms other than pure sine, but at least the trivial ones listed in section 3.2. Requirements for an oscillator's waveform synthesis algorithm are to generate periodic bandlimited signals, to avoid aliasing and to be computationally efficient. The latter requirement originates from the number of times the oscillator is called, this is at least once for each sample instant. Depending on the amount of polyphony, i.e. the maximum number of parallel voices playable, they can even be called multiple times for each sample.

Välimäki and Huovilainen (and others [Pek07, p. 28], [Ota15]) divide digital oscillator algorithms into three classes in regard to the amount aliasing left[VH06]:

1. Ideal-bandlimited methods without harmonics above the Nyquist frequency, e.g. additive or wavetable synthesis
2. Quasi-bandlimited methods with low aliasing, e.g. *BLIT* and *BLEP* methods

[Bra01]

3. Alias-supressing methods, e.g. oversampling and filtering trivial waveforms

A fourth class, so called ad-hoc methods, that uses non-linear processing techniques is mentioned in more recent publications ([Ota15], [Pek07]). They are not of interest for this research because this class of methods is developed for very specific applications [Ota15, p. 26].

The aim of this chapter is to introduce the generic structure of a digital oscillator and to evaluate quasi-bandlimited methods against (fully) bandlimited wavetable synthesis.

6.1 Generic Oscillator Structure

A common structure that is used for a wide variety of oscillators is shown in fig. 6.1. It consists of a *phase accumulator* which adds a *phase increment* φ to itself each time a clock signal t arrives. The phase increment is given by the *fundamental frequency* f_0 as $\phi = 2\pi f_0 / f_s$. Subsequently, an initial *phase offset* $\phi_0 \in [0, 2\pi]$ is added to the accumulators output. The *phasor signal*

$$\begin{aligned} \phi[t] &= \phi t \mod 2\pi \\ &= (\phi[t-1] + \phi) \mod 2\pi \end{aligned} \tag{6.1}$$

for a discrete time variable t wraps around on a full cycle. The second form shows the recurrence relation for the phasor signal where $\phi[t] = 0, \forall t \leq 0$. It is convenient to normalize $\phi[t]$ to a fraction of the waveform's period

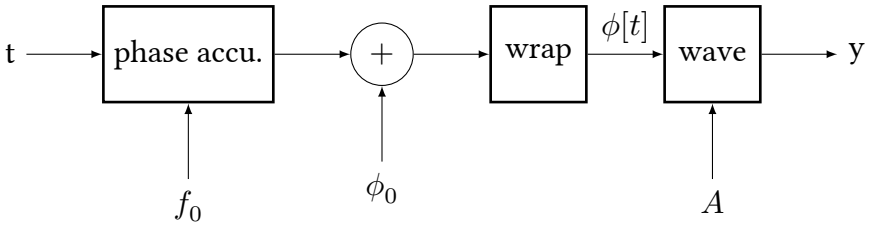

Figure 6.1: Block diagram of a generic oscillator.

$$\varphi[t] = \frac{\phi}{2\pi}t \tag{6.2}$$

with $\varphi \in [0, 1]$, e.g. to map the phasor to a wavetable lookup index. The *wave* function maps the *phasor signal* to the desired wave shape and multiplies the output signal with a given amplitude A.

6.2 Trivial Waveform Generation

The trivial way for generating geometric waveforms is to sample them without bandlimiting. A sawtooth wave can be expressed by a *bipolar modular counter*

$$saw(t) = 2\varphi(t) - 1 \tag{6.3}$$

where

$$\varphi(t) = f_0 t \mod 1 \tag{6.4}$$

is a phasor signal (modular counter) for a continuous time variable t in seconds [Pek14, p. 5]. An inverted sawtooth wave with a ramp that decreases from 1 to -1

can be obtained by

$$saw_{\text{invert}}(t) = 1 - saw(t). \tag{6.5}$$

Both remaining waveforms, square and triangle, can be derived from sawtooth waves. Rectangular waveforms can be produced by subtracting two sawtooth waveforms with a proper phase shift [VH06, p. 22]

$$rect(t) = saw(t) - saw(t - \frac{p}{f_0}) \tag{6.6}$$

where $p \in (0, 1)$ is the duty cycle. Square waves are simply the symmetric case of rectangular pulses with 50% pulse width. Another trivial way of generating a rectangular pulse is by comparing the output x of bipolar modular counter with the pulse width p as in the following closed form expression:

$$rect(x) = \begin{cases} 1 & x < p \\ 0 & x = p \\ 1 & x > p \end{cases} \tag{6.7}$$

By taking the absolute value of a sawtooth wave one gets a inverted triangle in the range of zero to one, doubling and subtracting from one results in bipolar triangle wave as shown in the first form of eq. (6.8). The second form shows that integrating a square wave over time t also results in a triangle wave which then needs to be scaled to a normalized range from -1 to 1 [Pek14, p. 7], hence a (scaled) square wave is the time derivate of a triangle waveform.

$$\begin{aligned} tri(t) &= 1 - 2|saw(t)| \\ &= 4f_0 \int_{-\infty}^{t} sqr(\tau) d\tau \end{aligned} \tag{6.8}$$

A straightforward digital implementation of those trivial waveforms is constructed by replacing the continuous-time phasor time signal with its discrete-time counterpart [Pek14, p. 8]. Unfortunately, those naive digital implementations suffer from severe aliasing distortion because the continous-time source signal of those geometric waveforms is not bandlimited, hence it contains an infinte number of harmonics as can be seen in their fourier-series representation (see eq. (3.7), eq. (3.8) and eq. (3.9)). The spectral tilt, i.e. the attenuation of harmonic partials with increasing frequency, is about 6 dB per octave for pulse and sawtooth waveforms and 12 dB per octave for triangle waveforms [Pek14, p. 11]. The steeper spectral tilt for triangle waveforms can be explained by their construction from pulse waves via integration which corresponds to the application of a first-order lowpass filter. Thus, a trivial triangle oscillator implementation can be sufficient if implemented with two or more times oversampling depending on the amount of tolerable aliasing, especially for devices with very limited processing resources.

6.3 Quasi-Bandlimited Waveform Synthesis

Quasi-bandlimited oscillator algorithms allow a certain degree of aliasing to be produced while making use of *psychoacoustic* effects like *masking*. Auditory masking means how sensitivity for one sound is affected by the presence of another sound which is largely dependend on the intensity and spectrum of the sound that

causes the masking [Gel10, p. 187], i.e. the human ear cannot differentiate between two sounds with roughly the same frequency spectrum if the intensity difference is large enough. Accordingly, the harmonics of a waveform can mask the aliasing components in their spectral vicinity. The intensity of aliasing must be particularly reduced in the range of 1 kHz to 5 kHz, because this is where human ears are most sensitive [Gel10, fig. 11.1].

6.3.1 BLITs

In 1996 Stilson and Smith presented in their paper "Alias-free digital synthesis of classic analog waveforms" [SS96] a method for synthesizing alias free geometric waveforms by integrating a *bandlimited impulse train* (BLIT). Sawtooth, pulse and, of course, triangle waveforms can be derived from a pulse train by integration which is inherently a bandlimited operation. Hence, it is sufficient to show how bandlimited impulse trains can be constructed by this method. The naive way of discretizing an impulse train is by approximating each impulse with a unit-sample pulse. The impulse trains period $p = f_s/f$ is rarely an integer, thus the locations of the unit-sample pulses must be approximated to the nearest sample instant. Figure 6.2 (b) clearly shows the irregular intervals between unit-sample pulses the *pitch-period jitter* which adds noise to the signal [SS96, p. 2].

The naive discretization approach suffers from aliasing just as the trivial waveform generation method (see section 6.2) because it is also not-bandlimited. Hence, a more sophisticated method is needed. The idea is to apply an ideal anti-aliasing filter before sampling the impulse train. Figure 6.3 shows the frequency response of an ideal anti-aliasing filter is a rectangle function in the frequency interval $(-f_s/2, f_s/2)$ and a continuous-time impulse response that is a sinc function:

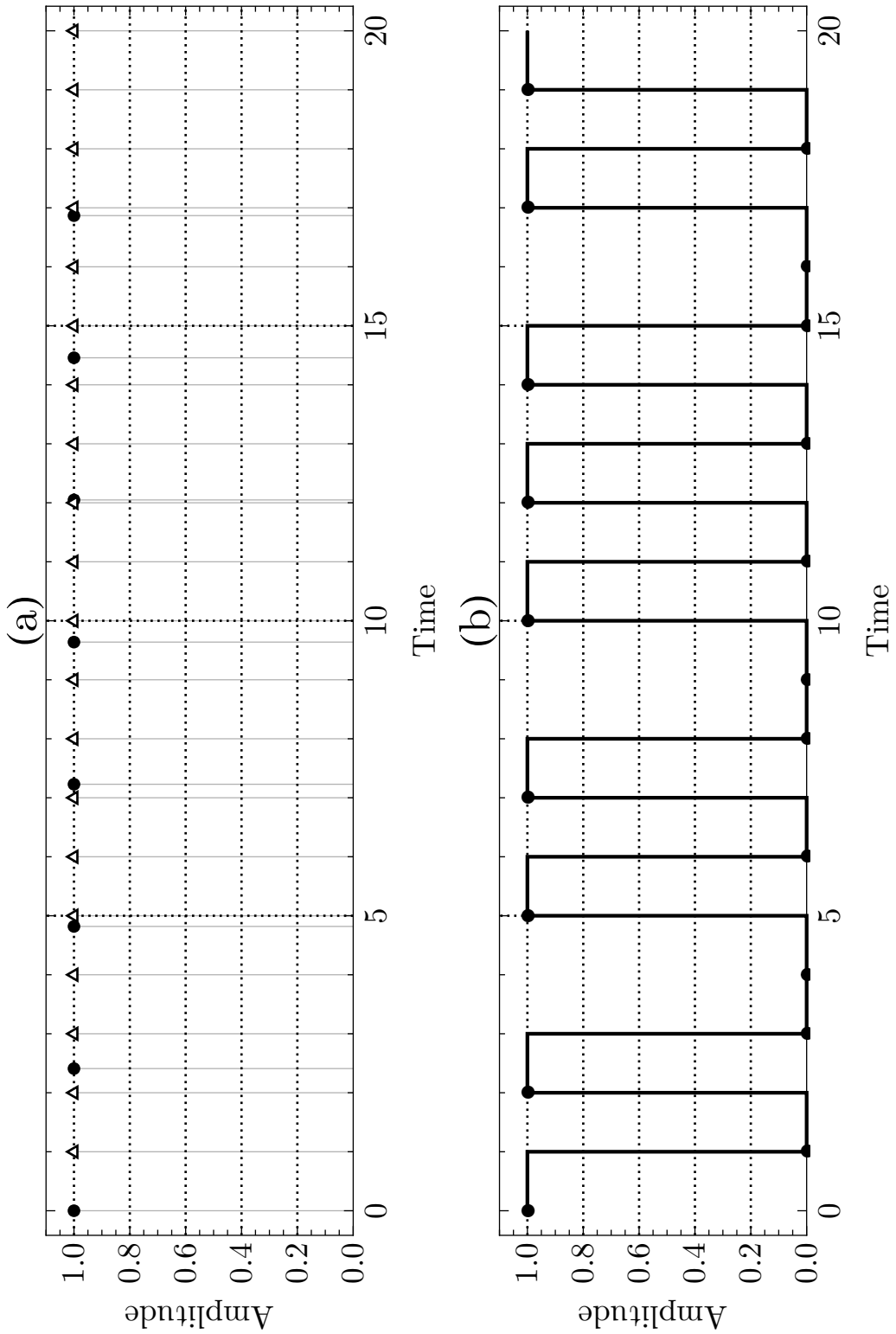

Figure 6.2: (a) Impulse train • with frequency $f = 8.3\,\text{Hz}$ and sample positions $\triangle$. (b) The approximated unit-sample pulse train with sample positions $p = f_s / f$ rounded to the nearest integer.

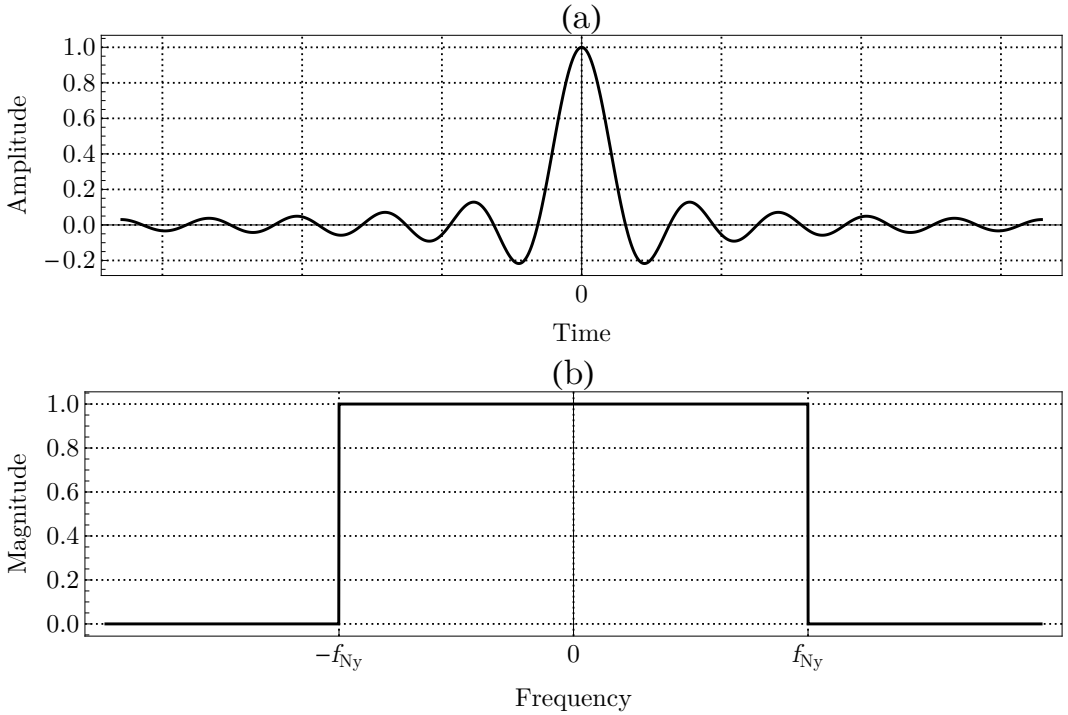

Figure 6.3: (a) Impulse response and (b) frequency response of an ideal anti-aliasing (lowpass) filter where $f_{Ny} = f_s/2$ is the frequency of the Nyquist limit.

$$h(t) = sinc(f_s t) = \frac{\sin(\pi f_s t)}{\pi f_s t} \tag{6.9}$$

Applying the ideal filter $h(t)$ to the unit-amplitude impulse train of period T_1

$$x(t) = \sum_{k=-\infty}^{\infty} \delta(t = kT_1) \tag{6.10}$$

by means of convolution gives a bandlimited signal $x_f(t) = (x * h)(t)$. Thus, x_f can now be sampled without aliasing which gives

$$y[n] = x_f(nT_s) = \sum_{k=-\infty}^{\infty} \text{sinc}(n + kp) \tag{6.11}$$

where $T_s = 1/f_s$ is the sample period and $p = f_s/f$ as defined above. The bandlimited discrete-time signal $y[n]$ can be interpreted as time-aliased sinc functions [SS96, p. 5], i.e. every impulse is replaced with sinc response of the ideal filter. Furthermore, Stilson and Smith provided a closed-form expression for the *sampled bandlimited impulse train*:

$$y[n] = \frac{M}{P} \text{sinc}_M \left(\frac{M}{P} n \right) \tag{6.12}$$

where

$$\text{sinc}_M(x) = \frac{\sin(\pi x)}{M \sin(\pi x / M)} \tag{6.13}$$

and M is the number harmonics. It is convenient to relate the number of harmonics M to the period in samples p as

$$M = 2\lfloor P/2 \rfloor + 1 \tag{6.14}$$

that is the largest odd integer smaller than the period [SS96, p. 6]. However, the BLIT method cannot be implemented as is because the sinc function is infinitely long.

6.3.2 BLIT-SWS

A realizable approximation that was proposed in the original paper by Stilson and Smith is called *Sum of Windowed Sincs* (SWS) and will be discussed in more detail. Since then, a lot more methods were developed but it is not in the scope of this thesis to give an overview about all of them. For a very thorough analysis of alternative methods refer to [Ota15] or [Pek14].

The difference between this realizable approach and the theoretical BLIT method in the previous section is that a window is applied to the ideal filter's impulse response to make it finite. Hence, eq. (6.11) becomes

$$y_w[n] = \sum_{k=-\infty}^{\infty} w(n)\,\mathrm{sinc}(n + kp) \tag{6.15}$$

where $w(n)$ is a window function. The choice of the window function determines the attenuation of harmonics in the spectrum. This is advantageous for frequency sweeps in contrast to exactly bandlimited methods where harmonics pop in and out which can cause unwanted transients. Aliasing is reduced by increasing the window length, in fact, a doubling in length approximately halves the transition band where most of the aliasing occurs. Stilson and Smith used a blackman window that spanned 32 zero crossings[1] of the sinc function [SS96, fig. 10] which attenuated aliasing to about -90 dB for 80% of the spectrum. The paper also proposed to use some oversampling to get a guard band in which the transition band can be moved by using a appropriate window length, e.g. for a sampling rate of 48 kHz and a window length of 64 sinc zero crossings the transition band would span 10% of the spectrum which gives a nearly alias-free frequency range up to

[1] The sinc function has zeros at $n * \pi$, $n \in \mathbb{Z}$

$0.9 \cdot 24\,\text{kHz} = 21.6\,\text{kHz}$.

A disadvantage of BLIT-SWS method is that the CPU consumption is proportional to the frequency because for each period a impulse must be inserted and replaced by the impulse response of the windowed filter (sinc). Furthermore, the sinc is centered on the impulse and must be mixed in several samples before the actual impulse arrives, thus lookahead is required [Huo10, p. 20]. By controlling the window length a tradeoff between CPU usage and quality can be achieved. Another feasible optimization is to tabulate the windowed sinc function and to retrieve its values by the use of an interpolated table-lookup.

6.3.3 BLEPs

Brandt proposed in 2001 a method to synthesize hard synced oscillators without aliasing called *bandlimited step* (BLEP) [Bra01], an extension of BLIT. Hard sync is the phase synchronisation of two oscillators with frequencies f_1, f_2 where a slave oscillator's phase is reset with each period of f_1. Clearly, this adds discontinuities to the synthesized waveform which cause aliasing if the waveform is sampled without bandlimiting. The method is not limited to hard-sync instead it can synthesize bandlimited versions of arbitray waveforms with discontinuities, like the geometric waveforms, if the derivates of the waveform are continuous across the point of discontinuity [Bra01, p. 3].

The method improves BLIT in two ways, first it almost removes the lookahead to the center of the impulse by using a minimum-phase impulse, and second, it removes the integration at run-time by pre-integrating the bandlimited step. The minimum-phase impulse is considered as a minimum phase FIR filter, i.e. all zeroes are located inside ($|z| < 1$) the unit circle, with the impulse response of a

windowed sinc. Integrating the minimum-phase impulse results in a minimum-phase bandlimited step (MinBLEP).

Waveforms are synthesized by producing their naive non-bandlimited shape and mixing in a MinBLEP each time a discontinuity in the waveform occurs.

Advantages of MinBLEP in contrast to BLIT-SWS are that former method removes numerical error and computational costs by avoiding the numerical integration of the impulse train at run-time and the lookahead is reduced to several samples of the bandlimited step. Also, the pre-integration step reduces aliasing by another 6dB over BLIT-SWS. However, the CPU usage is still proportional to the oscillators frequency.

6.4 Ideal Bandlimited Waveform Synthesis

Ideal bandlimited methods generate waveforms with a finite number of harmonics. The number of harmonics is limited by the highest harmonic frequency less than the Nyquist limit, hence, the waveforms are completly alias-free. A multitude of ideal bandlimited methods were developed, e.g. additive synthesis (section 4.1), feedback delay loops (FDL) and discrete summation formulae (DSF). DSF's are solely using properties of trigonometric functions to synthesize the waveform and are not considered for the synthesizer of this thesis because evaluating trigonometric functions is quite CPU intensive. Feedback Delay Loop's are a relatively new method (first published in 2009) and, unfortunately, are not taken into account since I have discovered this method at a very late time of editing and the implementation effort is unclear[2]. Pure additive synthesis is the most CPU intensive of

[2]The interested reader can refer to [Moo76] for description of DSF and [Nam+09] for FDL.

those three because it requires to sum a sinusoidal oscillator for every harmonic to be synthesized, therefore a wavetable based approach is evaluated which uses lookup-tables containing bandlimited cycles of the waveform.

6.4.1 Wavetables

Wavetable oscillators can generate *arbitrary* static harmonic spectra. Nonetheless, dynamic spectra can be generated by crossfading the output of two detuned or different wavetable oscillators [Fre10, p. 41] or by applying a time-varying filter to the output [Huo10, p. 37]. The basic idea behind wavetable oscillator's is simple, a single cycle of an arbitrary waveform is sampled and stored into an array of memory locations and looped at different speeds to simulate playback at a different pitch. The fundamental frequency f_0 of a wavetable oscillator's output signal is given by

$$f_0 = \frac{\varphi \, f_s}{N} \tag{6.16}$$

where f_s is the sample rate, N the table length and φ the phase or table increment, with a special case for $\varphi = 1$ called *natural fundamental* or f_{nat} [Fre10, p. 41]. Figure 6.4 shows the wavetable of a sine wave sampled at 24 equidistand points with a natural fundamental $f_{nat} = 48\,\text{kHz}/24 = 2\,\text{kHz}$ for a sample rate $f_s = 48\,\text{kHz}$.

The oscillators output signal loses clarity if the wavetable is played back with a frequency that is lower than f_{nat}, thus the wavetable should not be too short. The oscillators table increment

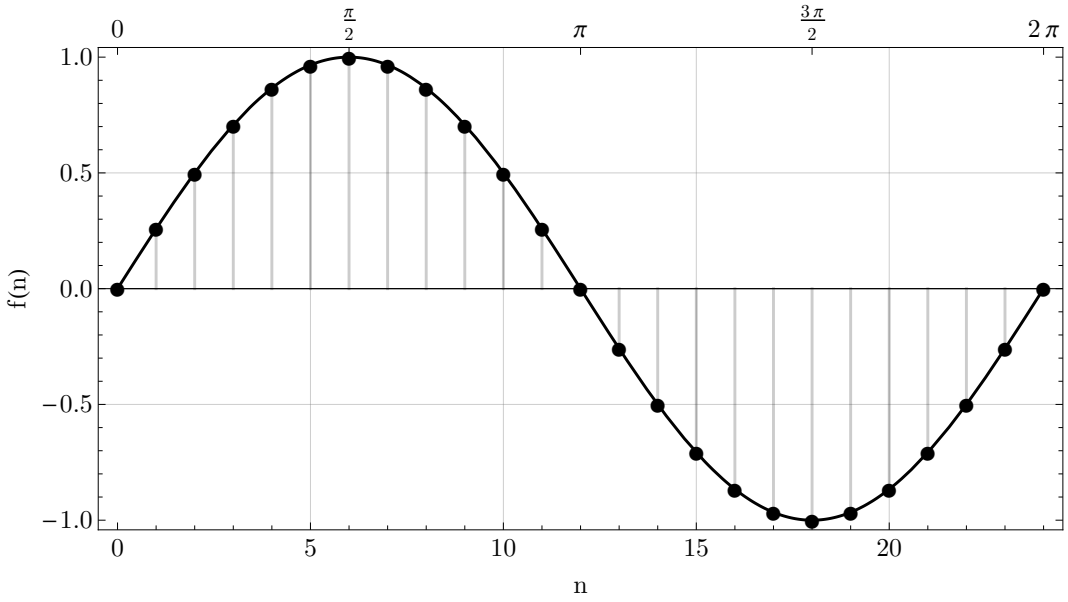

Figure 6.4: Wavetable for a single sine wave cycle sampled at 24 equidistant points. The bottom x-axis shows the index of the sample in the wavetable and the top shows angle in radians.

$$\varphi = \frac{N\ f_0}{N} \tag{6.17}$$

is usually not an integer, thus the lookup index

$$n = t\ \varphi \mod N \tag{6.18}$$

also has a fractional part. Therefore, some kind of interpolation is needed to map the lookup index n onto a integer valued table index. The accuracy of the table lookup depends on two factors, the interpolation scheme that is used and the stored waveform cycle's length. Interpolation schemes may be evaluated by comparing the error of the interpolated output that is the difference between the *ideal* waveform function's value at some point and the interpolated value at the same point. It is common practice to use a sinusoid as the ideal waveform because other waveforms can be interpreted as superpositions of those. The error for zero- and first-degree interpolation of a sine wavetable with length $N = 64$ is shown in fig. 6.5.

Polynominal interpolation schemes, which are the only ones considered here, are determined by their order n, i.e. a n-th order polynomial passes through the $n + 1$ nearest points in the wavetable. A zero-degree (or nearest-neighbour) interpolation simply rounds the index to the next integer. A first-order (or linear) interpolation uses a linear function to approximate the table output

$$y_{lin}(n) = y[n_0] + (y[n_0 + 1] - y[n_0])(n - n_0) \tag{6.19}$$

where $n_0 = \lfloor n \rfloor$. Higher order interpolation schemes can be considered for sys-

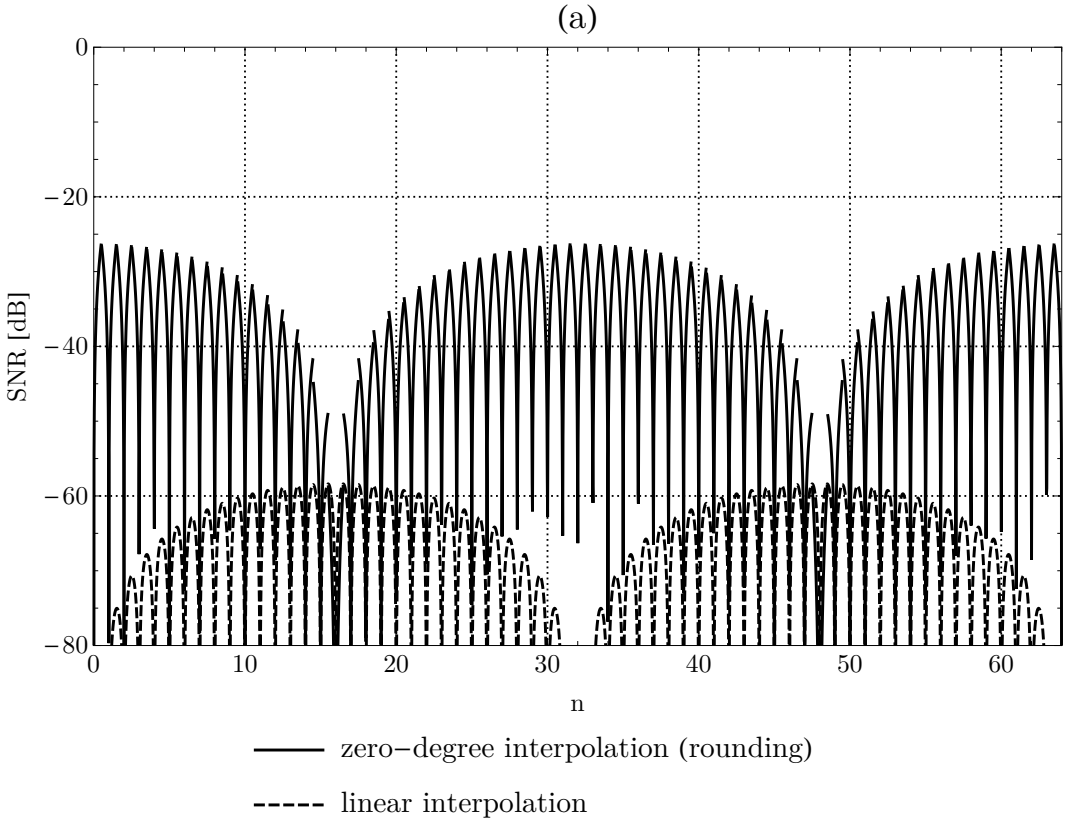

Figure 6.5: Error of zero-degree (nearest neighbour) and first-degree (linear) interpolation for a wavetable of 64 samples.

tems where memory is scarce, otherwise linear interpolation performs well for $N \geq 64$ samples. In fact, the root-mean square (RMS) lookup error decreases by 12 dB each time the table length doubles [Puc06, p. 44], hence the RMS error of a linear interpolated wavetable with 64 samples length is ≈ -64 dB (fig. 6.5 shows the non-RMS error which is ≈ -58.3 dB).

For sine, where the sole harmonic is also the fundamental, a single table is enough but for the geometric waveforms more tables are required. Each of those tables spans only frequencies in the vicinity of f_{nat} because pitch shifting the table by a large factor ($> 2f_{nat}$) can introduce severe aliasing whereas shifting it to a low frequency $< 0.75f_{nat}$ may result in a dull sound because too many harmonics are missing. Hence, a table for each octave of the desired frequency range should be used which is then selected a run time depending on the note to be played. A table switch causes a sudden drop in spectral energy, as can be seen in fig. 7.7, which can be perceived for slow frequency sweeps. This effect can be alleviated by increasing the number of tables at the cost of memory.

It is convenient to design the waveforms spectra directly in the frequency domain and convert it back into the time domain by using the inverse discrete-time fourier transform that is eq. (6.21) from the transform pair given below where $\omega = 2\pi f$ is the angular frequency as usual.

$$X[f] = \sum_{n=0}^{N-1} x[n] e^{-i\omega n} \tag{6.20}$$

$$x[n] = \frac{1}{N} \sum_{f=0}^{N-1} X[f] e^{i\omega n} \tag{6.21}$$

To get a real valued signal it must be ensured that $X[k] = X^*[N-k]$, i.e. the spectrum is conjugate symmetric (mirrored), where N is the table size. This process is illustrated by fig. 6.6 and fig. 6.7 for wavetables of length $N = 128$ where each iteration doubles the number of harmonics.

6.5 Conclusion

A wavetable based approach was selected for the synthesizer's oscillators because of its simple implementation, high audio quality and great flexibility that allows to easily support any arbitrary waveform. Nonetheless, a mixed approach that uses BLEP based oscillators for geometric waveforms and wavetables for arbitray or user defined spectras would be a desirable optimization.

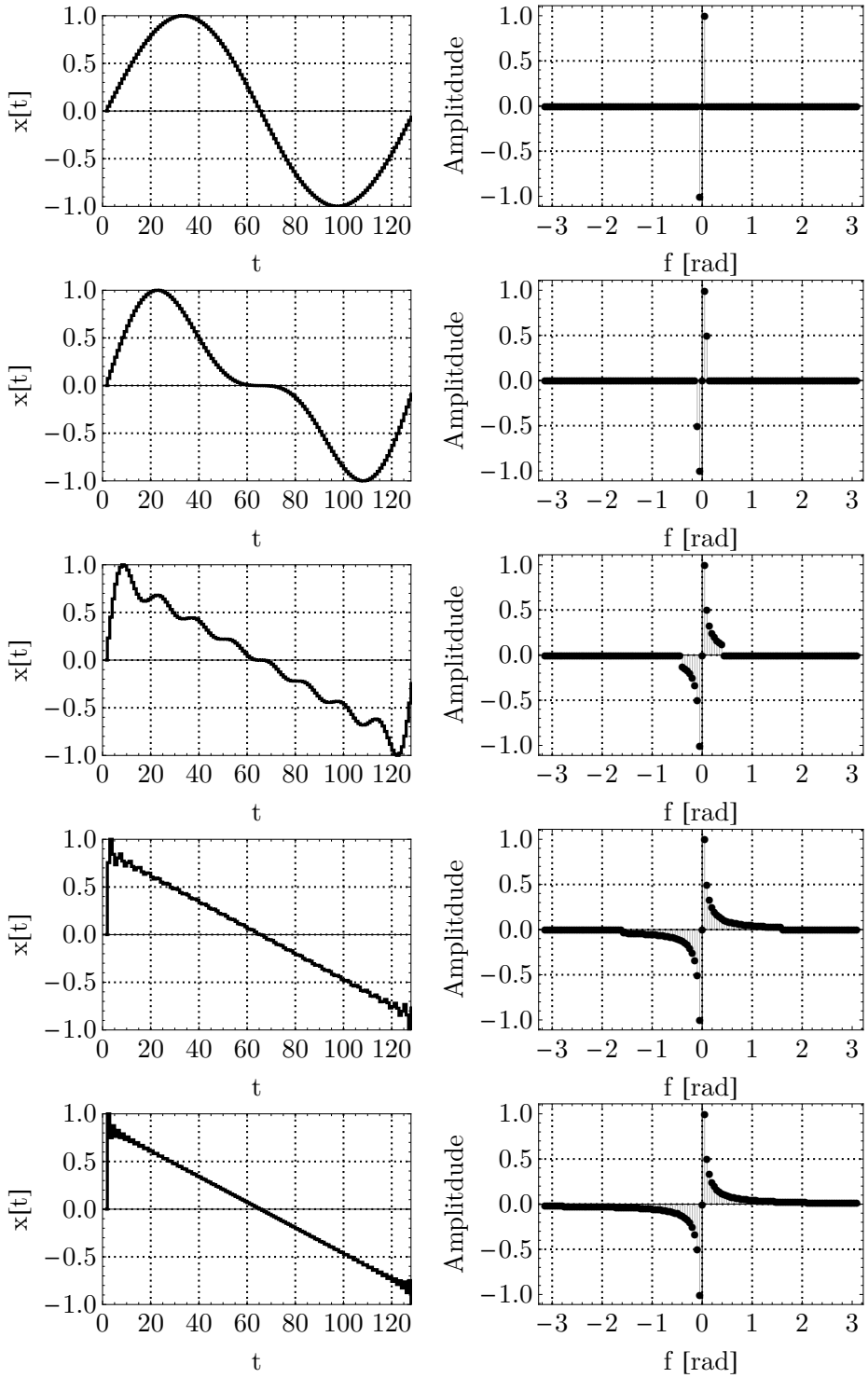

Figure 6.6: Harmonic spectra and time domain representation for a sawtooth waveform.

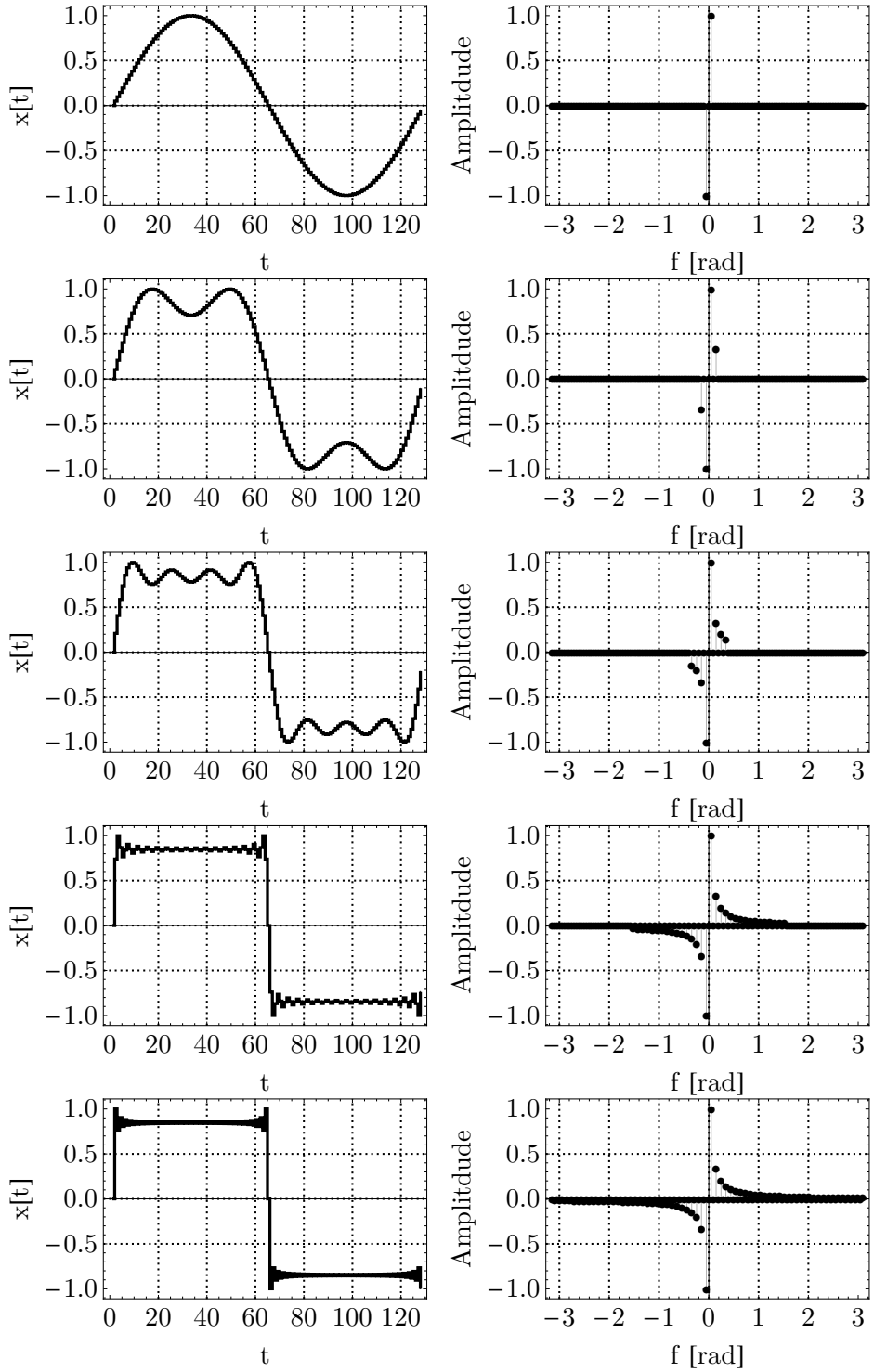

Figure 6.7: Harmonic spectra and time domain representation for a square waveform.

7 Implementation Details and Evaluation

The following chapter explains implementation details of the software synthesizer that was developed in this thesis. It also evaluates the implementation with respect to latency in section 7.3.1, aliasing behaviour of the wavetable oscillators in section 7.6 and the time-varying behaviour of the multi-mode filter in section 7.7. The decision to chose Rust as the implementation language is discussed in the first section and the important terms *real-time* and *latency* are defined in Section 7.2 and section 7.3, followed by an overview of the synthesizer's structure in section 7.4. Thereafter, all components of the structure are described in section 7.5 to section 7.9.

The full source code of the synthesizer and its libraries is published as open source software available in [Lin16e], [Lin16b], [Lin16d] and [Lin16c].

7.1 Why Rust?

Real time audio applications must finish their signal processing in tight time bounds to ensure that the sound card can output a continuous audio stream. Missing such a

time limit will result in unpleasant sound glitches, and—in the worst case—renders a whole song recording useless. Additionally, the time bounds must be as tight as possible to reduce the latency (see section 7.3) between user input (playing a note) and output of the calculated signal from the speaker. Therefore, memory-managed languages like Java, Go or C# that use a garbage collector, which can produce non deterministic program stops while examining the state of variable references, have not received any further consideration. Interpreted languages like Python or Ruby face the same problems as compiled managed languages by also having increased runtime costs. Audio application development in those languages is still possible but commonly requires to write the signal processing as C modules and interface with them through the language's foreign function interface (FFI) (e.g. pyo [Bél16] uses this approach). Unmanaged languages like C and C++ offer the control over memory that is needed to make reliable claims about the runtime behavior while avoiding the overhead of an additional runtime environment. This comes with a downside in memory safety and introduces a whole new class of possible bugs compared to managed languages. Those bugs are very likely to cause *undefined behavior* or to crash the program. Unfortunately, they are also very hard to debug.

One of the major selling points of Rust is guaranteed memory safety *without* garbage collection and *data race freedom* while providing the same level of control over memory as C/C++. These goals are achieved through a variety of concepts, like ownership, lifetimes and borrowing, to know at compile when memory can be safely freed, and to enforce that there is only a single mutable access to any variable at any given time in the run of the program. Explaining these concepts is outside the scope of this thesis and the official Rust Book [K+16] does a great job doing this in detail, hence, this is left as an exercise for the interested reader.

The Max Planck Institute for Software Systems has started the RustBelt [Sof15]

research project in 2015 to develop formal foundations for the Rust programming language. One of the main goals of the research group is to formally investigate if the claims made about data-race freedom, memory and type safety actually hold[1].

7.2 Real-Time

There are different forms of real-time computing in computer science but common to all of them is that computations have to be finished in a predefined amount of time and the shorter this time span is the faster those computations have to be. The definition of real-time used in the context of this thesis is of Rabiner et al. [Rab+72, p.2]:

> *A real-time process is one for which, on the average, the computing associated with each sampling interval can be completed in a time less than or equal to the sampling interval.*

Based on this definition, the time to compute a single sample should not exceed $1/f_s = 20.833\,\mu s$ for a common sampling rate of $f_s = 48\,kHz$. Nonetheless, sample calculation time should be much shorter than this limit to compensate for delays introduced by the operating system, e.g. context switches caused by the process scheduler. Multimedia applications can greatly benefit from a custom process scheduler like Kolivas's *Brainfuck Scheduler* or the more recent implementation called *Multiple Queue Skiplist Schedule* [Kol16] because those schedulers are optimized for process responsiveness on symmetric multiprocessing platforms like desktops or laptops unlike the default *Completely Fair Scheduler* [Mol07] which aims to maximizes for overall CPU utilization and must fit for a wide variety of

[1] As of the time of writing there is no evidence that the claims are untrue, therefore it's assumed that they actually hold.

use cases.

7.3 Latency

The *responsiveness* of an electronic musical instrument is mainly determined by its latency, i.e. the time delay between two causally connected events.

An instrument is the more responsive, the less latency between an input event and the corresponding sound output it has. The latency is imperceptible for the user if the delay between the input event and audio output stimuli is less than 24 ms [Ade+03]. Furthermore, there is an even stronger latency limit of $\approx$ 2 ms, that is the *temporal resolution* of the human hearing, as shown by [FZ07, p. 294] using psycho acoustic measurements. However, it is not realistic to use this as an upper limit for the synthesizer's system latency, considering that the sound propagation delay from a speaker to a listener at a speed of sound $v = 343.2\,\mathrm{m\,s^{-1}}$ at normal temperature[2] and a common listening distance of $d = 2\,\mathrm{m}$ is about 3 times larger than the temporal resolution $d/v = 2\,\mathrm{m}/343.2\,\mathrm{m\,s^{-1}} = 5.82\,\mathrm{ms}$. Therefore, achieving a system latency of less than 24 ms is favorable.

The sum of two delayed audio signals can act like a comb filter [LB07, p. 1], such that even very short latencies can cause spectral artifacts in applications that monitor input signals. Fortunately, there is no input audio signal for this synthesizer, thus only temporal issues must be taken into account.

[2] The *normal temperature* is defined by the National Institute of Standards and Technology (NIST) as $20°C$ at 1 atm absolute pressure.

7.3.1 System Latency

The overall latency of the synthesizer is the sum of different latency portions made up a variety of components in the audio output chain. At first there is the control input latency that is either that of the MIDI or OSC connection, where the OSC latency greatly depends on the network connection's physical layer, e.g. Bluetooth, Wi-Fi or USB. Usually a Wi-Fi connection is used to connect an OSC client, like a tablet running liine's lemur or some other OSC capable controller application. A USB connection is recommended to get reliable network latencies between an OSC client and the synthesizer but an ad-hoc Wi-Fi connection that is exlusively used for transporting OSC messages can also work well. Another latency portion is introduced through the synthesizers and the audio backends output buffer. The backend's audio buffer size can be configured in most cases except for some backends like PulseAudio which is not recommended for real-time use anyway. Lastly, there is a propagation delay which can be neglected for usual listening distances or is near zero if headphones are used.

Measuring the system latency is hard because an experimental setup is needed that triggers the input device and starts a sound recording at the same time. Such a setup is error prone because the time for triggering a piano key of the MIDI keyboard and the input buffer of the sound card must be compensated. Summing the input latency and delays introduced by the internal audio buffers gives a good approximation without the errors of an experimental setup. The overall latency l can be approximate by evaluating

$$l = l_{in} + \frac{b_{syn} + b_{out}}{f_s} \tag{7.1}$$

where l_{in} is the input latency, f_s the sample rate and b_{syn}, b_{out} are the synthesizer's and sound card's buffer sizes in samples. Given a sample rate of 48 kHz, an input latency of 1.3 ms and buffer sizes of $b_{syn} = 512, b_{out} = 256$ inserted in eq. (7.1) results in a system latency $l = 1.3\,\text{ms} + (512 + 256) \cdot (48\,\text{kHz})^{-1} = 17.3\,\text{ms}$ that is about ¾ of the 24 ms limit and thus sufficiently small to be unperceivable. Both buffer sizes were used without buffer underruns on the development machine, a HP elitebook 8460p equipped with an intel® i5-2520M running Arch Linux kernel 4.8.15-2-ck-sandybridge and JACK2 [Dev16] as audio backend. The input latency was measured by taking the average round-trip times of 100 ICMP echo requests (ping) between an OSC client, an android tablet running liine's Lemur OSC app, connected through an ad-hoc Wi-Fi connection and the computer running the synthesizer application.

7.4 Structure

The synthesizer's structure consists of mainly three parts. In the first part, incoming user input, as MIDI or OSC messages, is processed and transformed into user events which are then send to the appropriate component addressed by the input message. Signal generation takes place in the second part which also handles user events, e.g. by triggering voices on a note input or modifying parameters of a component like changing the filter's cutoff frequency. Then the generated signal is fed into a ring buffer (see section 7.8). Lastly, in the third part, the ring buffer is read-out by a callback from an audio backend that writes the data into the sound card's audio buffer. The overall structure of the synthesizer is depicted in fig. 7.1.

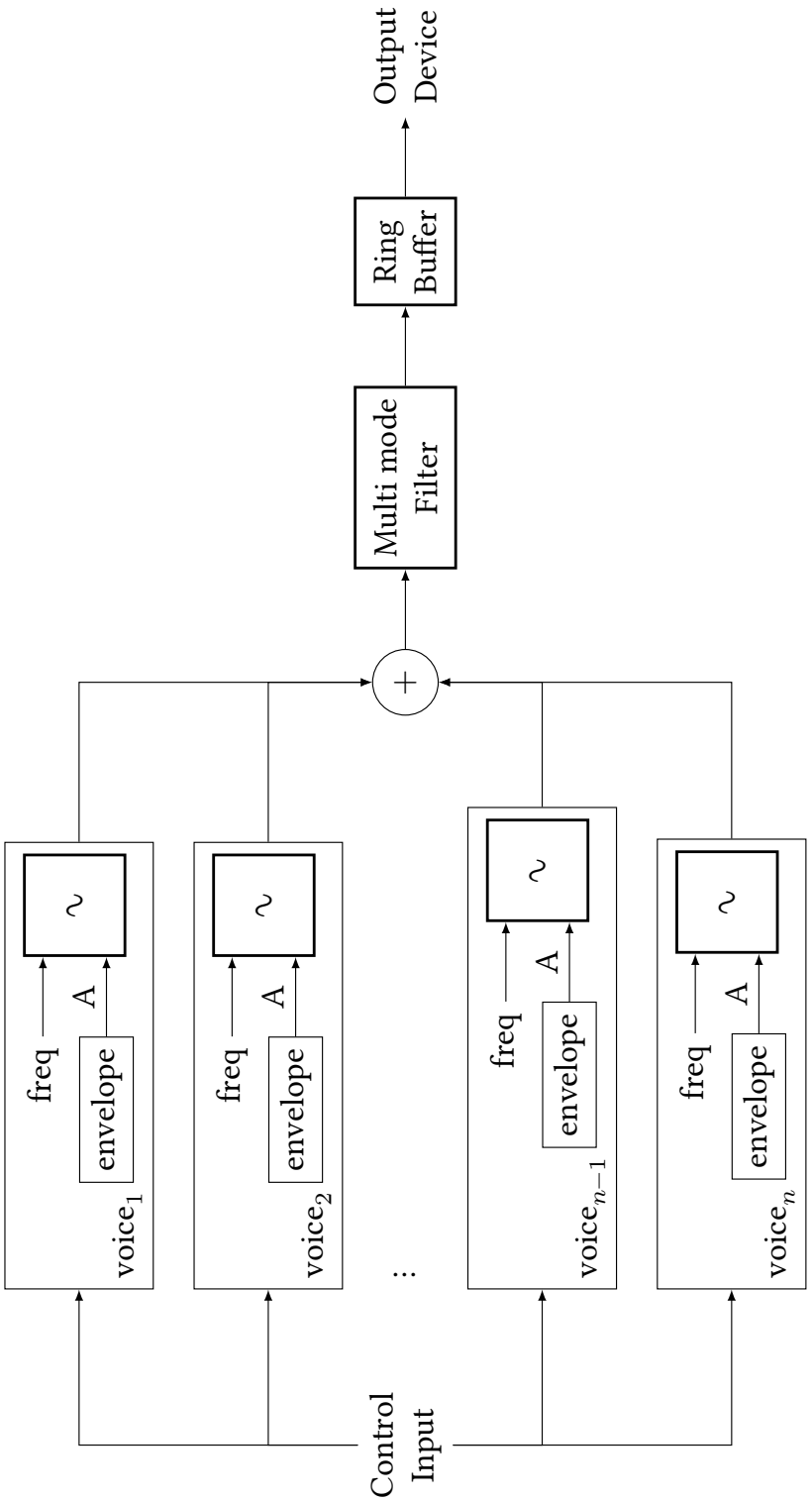

Figure 7.1: Structure of a basic polyphonic subtractive synthesizer.

7.5 Control Input

MIDI and OSC (see section 2.1 and section 2.2) protocol support is provided by two libraries, *portmidi-rs* [Phi16] and *rosc* [Lin16c]. The first library, *portmidi-rs*, is a safe Rust wrapper around the cross-platform real-time MIDI C library *portmidi* [16a]. The word *safe* in this context means that memory-safety is guaranteed by the fact that there are no unsafe operations for the use of *portmidi-rs*' API necessary, like e.g. pointer dereferencing. Furthermore, *portmidi-rs* was completly rewritten [Lin16a] in order to follow Rust's language idioms more strictly, i.e. a library user does not have to drop unused objects explicitly, instead this is done implicitly when the object goes out of scope.

At the time of writing, there was no Rust OSC library that fully supported the OSC 1.0 specification and was also compatible with Rust 1.0 and later versions, thus a new library had to be implemented, named *rosc*. The library achieves full compatibility with the OSC 1.0 specification and features encoding and decoding of OSC messages while being transport and platform indepent. OSC messages are decoded from their byte array representation, e.g. received as payload of an UDP packet, by pattern matching [Lin16c, 'src/decoder.rs', lines 129-161] against the corresponding variant of the algebraic Rust data type shown in listing 1.

7.5.1 Lemur

A multitouch capable graphical user interface (GUI) for the synthesizer is provided by the implemenation of a custom controller patch for liine's Lemur application [Lii16]. Lemur resembles the control surface of Jazzmutant's Lemur, a multi touch capable OSC control surface hardware device, but runs on Android or iOS devices

```
pub enum OscType {
    Int(i32),
    Float(f32),
    String(String),
    Blob(Vec<u8>),
    Time(u32, u32),
    Long(i64),
    Double(f64),
    Char(char),
    Color(OscColor),
    Midi(OscMidiMessage),
    Bool(bool),
    Nil,
    Inf,
}
```

Listing 1: Definition of OSC data types in [Lin16c, ‘src/types.rs‘].

instead. Control information is send as OSC Bundles which contain one or more OSC Messages containing the actual control signal. The custom control interface shown in fig. 7.2 to fig. 7.6 was implemented using Lemur’s control surface editor and makes use of the integrated scripting language.

Figure 7.2 shows the piano grid of the control surface with 8 key rows of 12 keys each. A row represents a single octave of a musical keyboard with black and white keys where the lowest key is colored blue. The root key of the grid can be shifted in a range from -3 to +3 octaves with the slider shown on the right. In contrast to classical MIDI keyboards it is not possible to play notes with different velocities because, Android and iOS devices lack pressure sensitivity sensors. Multi touch support allows to play the synthesizer polyphonically by pressing multiple keys at once.

There are four identical oscillator control sections with separate controls for each oscillator’s envelope, phase, detune and transpose parameters and a list of wave-

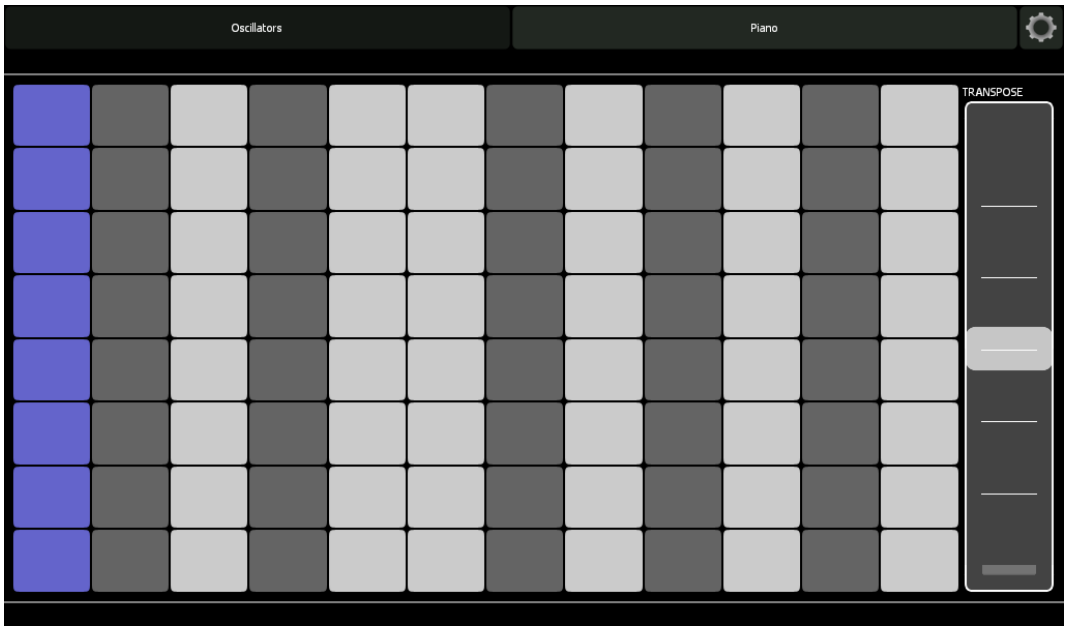

Figure 7.2: View of the piano panel

forms to choose from. Each oscillator can be transposed in a ± 3 octave range and detuned in a by ± one seminote at a resolution of 1 cent [Lin16e, ‘src/dsp/wavetable.rs‘, lines 357-372] which is a 100th of an equal tempered semitone. Also, the phase offset can be controlled in a range of 0° to 180°.

Figure 7.4 shows the FM control section where each oscillator section contains a group of sliders to control the amount of frequency modulation applied to its fundamental frequency, including feedback modulation, i.e. an oscillator is modulating its own frequency. In the screenshot the frequency of the first oscillator is modulated by the second which is itself modulated by the third oscillator.

The oscillator mixing panel shown in fig. 7.5 contains a slider for amplitude control and a bipolar (zero at center position) slider to set the stereo panning for each oscillator.

Figure 7.3: View of the oscillator control panel showing the first of four control panels

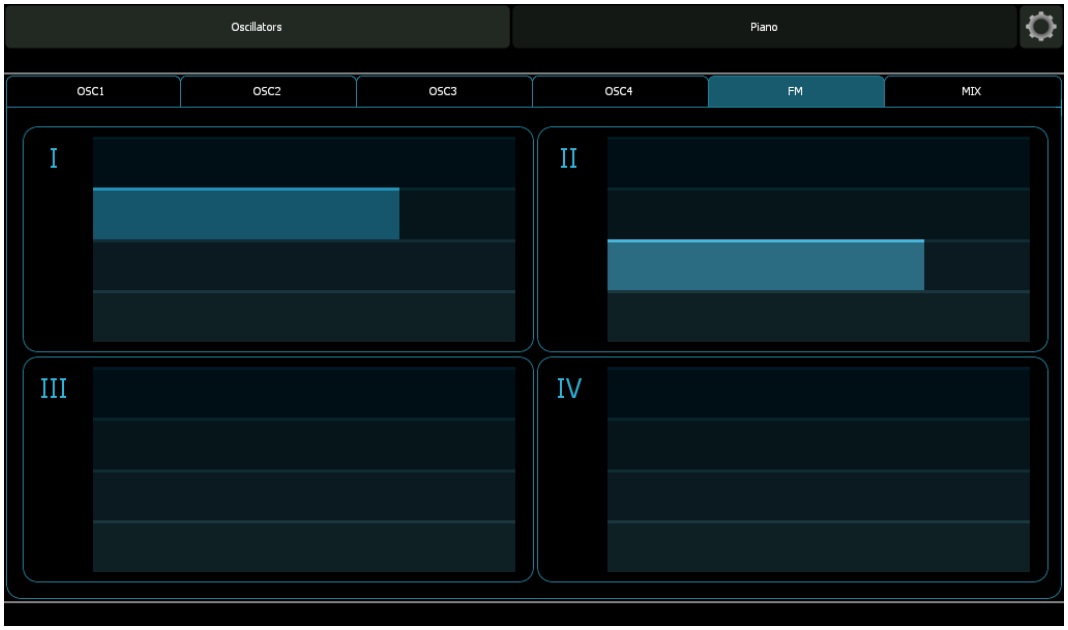

Figure 7.4: View of the FM control panel

Figure 7.5: View of the mixer panel

Parameters of the multi-mode filter can be set in the filter control panel. It contains a drop-down list for selecting the filter mode and a two dimensional control field to set the filter's cutoff frequency on its x-axis and the resonance control on its y-axis.

7.6 Wavetable Oscillator

The wavetable oscillator's implementation follows closely the description that was given in section 6.4.1, i.e. there are separate wavetables for each octave of the desired frequency range, i.e. ongoing from a lowest frequency f_L there is a new wavetable that covers frequencies in

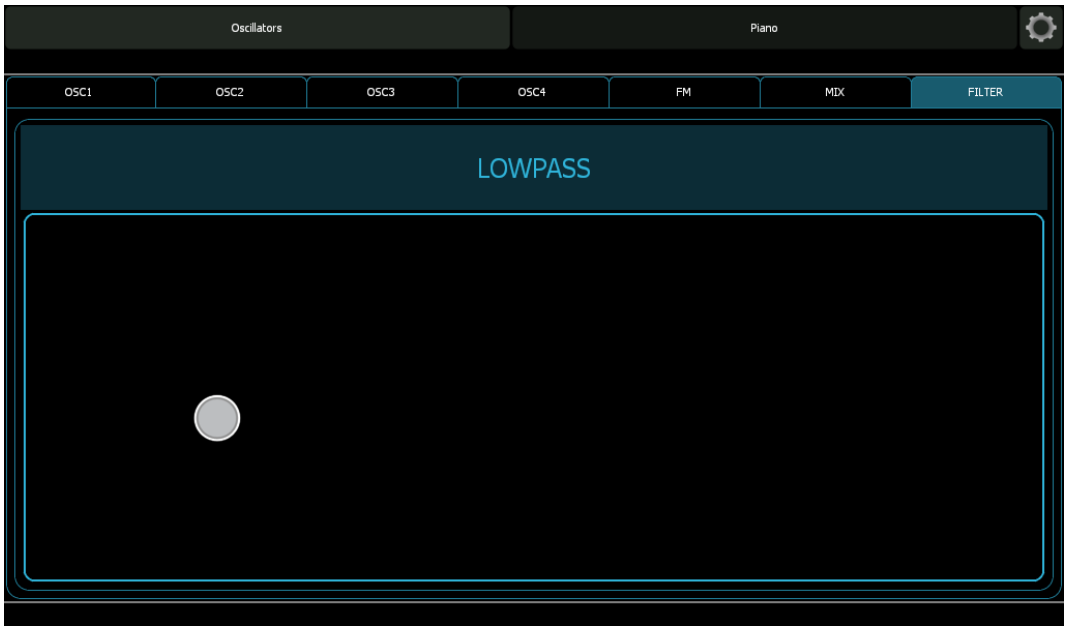

Figure 7.6: View of the mixer panel

$$[n\,f_L, (n+1)\,f_L]\,, \quad n \in \mathbb{N} \wedge n\,f_L < \lfloor f_s/2 \rfloor - 1$$

where n is the octave. A tables length and the number of harmonics contained in its waveform gets halved each time n increases until the waveform consists solely of its fundamental in the highest wavetable.

The waveforms are generated by means of Fourier synthesis, i.e. an inverse Fourier transform (IFT) is applied to the spectrum that was predefined for the desired waveform. Section 7.6 shows how the spectrum of a inverse sawtooth wave is defined where line 8 asserts that the spectrum is mirrored in order to obtain a real valued signal from the IFT. By using a length $l = 2^n,\ n \in \mathbb{N}$ for the initial spectrum table assures that each following table also has a length which is a power of two because halving the table's length corresponds to decrementing n by 1. Those

table lenghts are optimal for the inverse mixed-radix fast Fourier transform (FFT) [Wel16] used in the implementation because it can make use of all symmetries in the FFT algorithm. All wavetables are serialized to disk so they can be read from a file [Lin16e, ‘src/dsp/wavetable.rs‘, lines 89-100] in subsequent application starts to reduce its initialization time.

```
Waveform::Saw => {
    for i in 1..harmonics {
        let magnitude = (i as Float).recip();
        spectrum[i] = Complex {
            re: 1.0,
            im: -1.0 * magnitude,
        };
        spectrum[table_size - i] = -spectrum[i];
    }
}
```

Listing 2: Calculation of Fourier coefficients for sawtooth waveform definition in the frequency domain [Lin16e, ‘src/dsp/waveform.rs‘, lines 163-172].

Both spectrograms shown in fig. 7.7 are calculated from frequency sweeps to which a Hann window was applied and that were generated from the wavetable oscillator as part of the unit test set. It can be seen that the aliasing amount does not exceed −80 dB which makes it inaudible. Additionally, the frequency sweep visualizes the use of separate tables and their decreasing number of harmonic content. The loss of spectral energy at each table switch can be perceived in the frequency sweep but it is not of great concern because the oscillators frequency is rather constant when the synthesizer is played. However, this should be considered for effects with pulsating pitch changes like a vibrato.

A spectrogram of a frequency sweep for a sinusoidal carrier and modulator with increasing modulation amount is shown in section 7.6. The increasing amount of harmonic content is clearly visible as well as the distortion that starts approx-

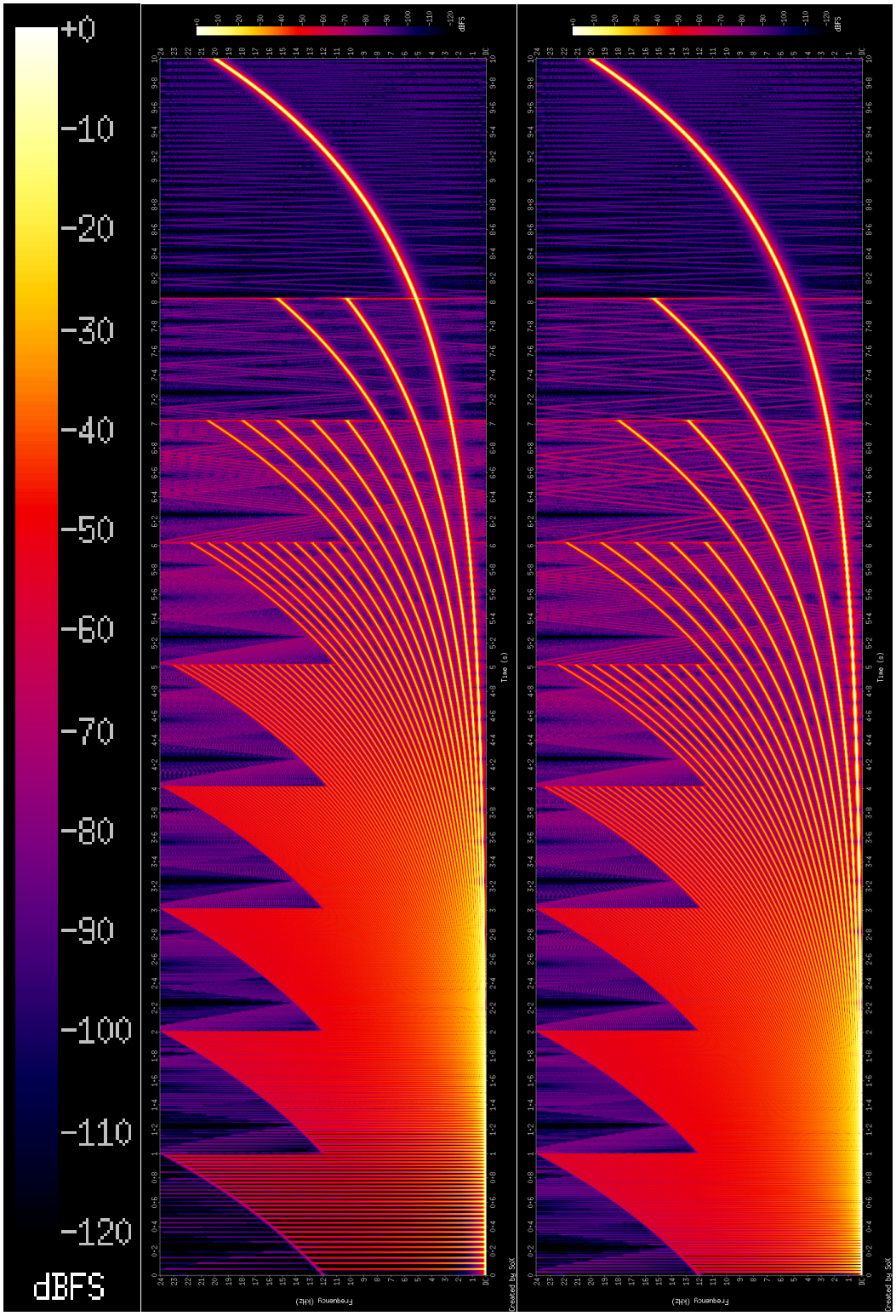

Figure 7.7: Spectrogram of a frequency sweep from 20Hz to 20kHz in the sawtooth (upper) and square (lower) wavetable. The x-axis denotes time in seconds whereas the y-axis represents frequencies in a range from 0 to 24 kHz.

imately in the last quarter of the spectrogram and which is almost unavoidable because it is hard to give a constraint for the modulation amount so that no distortion will occur.

7.7 Filter

The multi-mode biquad filter [Lin16e, ‘src/dsp/filter.rs‘] is a direct implementation of the filter design from Bristow-Johnson’s EQ Cookbook [Bri16]. Bristow-Johnson’s filter is derived from an analog prototype using bilinear transform (see section 5.6 and section 5.6.1) and is a stable and popular design which is also used in the biquad implementation of the WebAudio API [16b] which got adopted by chromium [16c] and firefox [16d] browsers.

Calculation of coefficients for the biquad filter is shown in listing 3 where `w` is the cutoff frequency expressed as angular frequency[3], `q` is the resonance parameter and `filter_type` is an `enum` variant that is matched against the supported filter types (line 11 in listing 3) to select the its zero coefficients corresponding to the desired mode.

A spectrogram of a white noise filtered by the biquad implementation in lowpass and bandpass mode with increasing filter cutoff and constant neutral filter resonance is shown in fig. 7.9. The parameter sweep of the spectrograms is generated by the filter’s unit test to check the stability for time-varying parameters by sweeping the cutoff parameter at audio rate with the result that there are no sound artifacts visible and perceivable in the rendered audio, hence the implementation is suitable as a time-varying filter.

[3] As usual, $w = (2\pi f_c)/f_s$ where f_c is the cutoff frequency and f_s is the sample rate in Hertz.

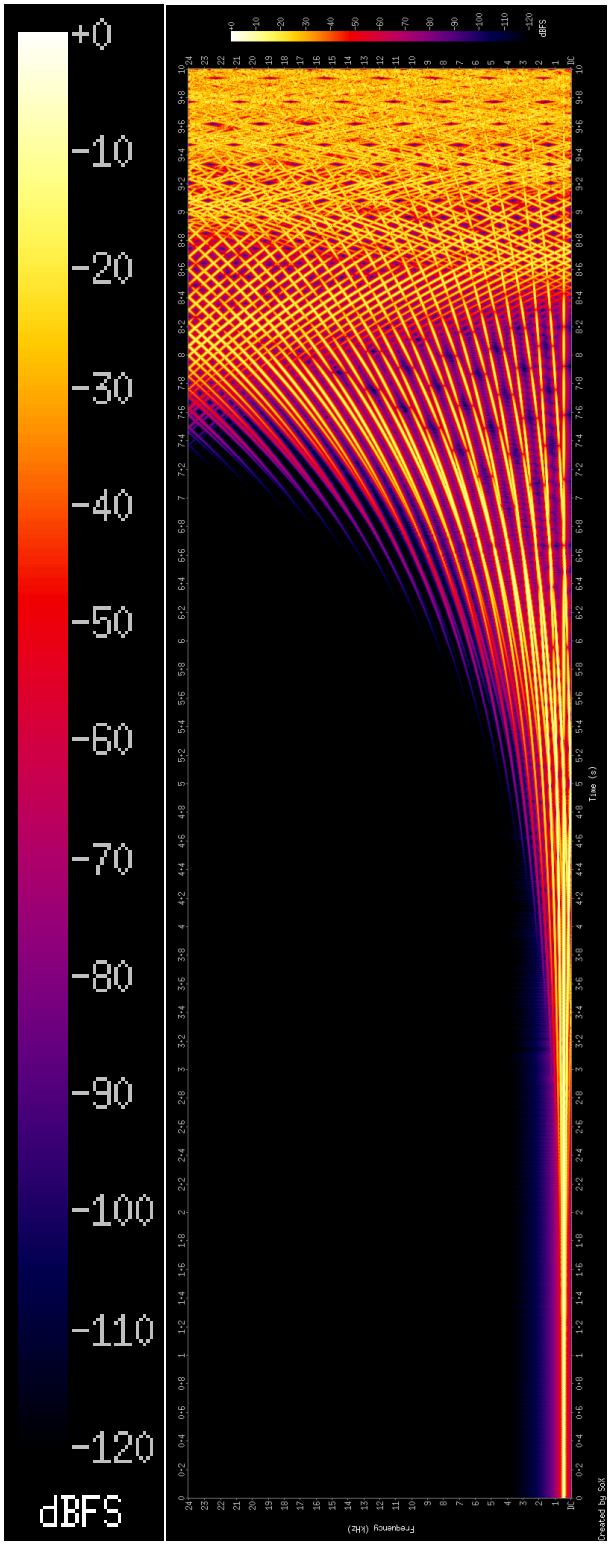

Figure 7.8: Spectrogram of a frequency sweep for sine modulator and carrier with increasing modulation amount.

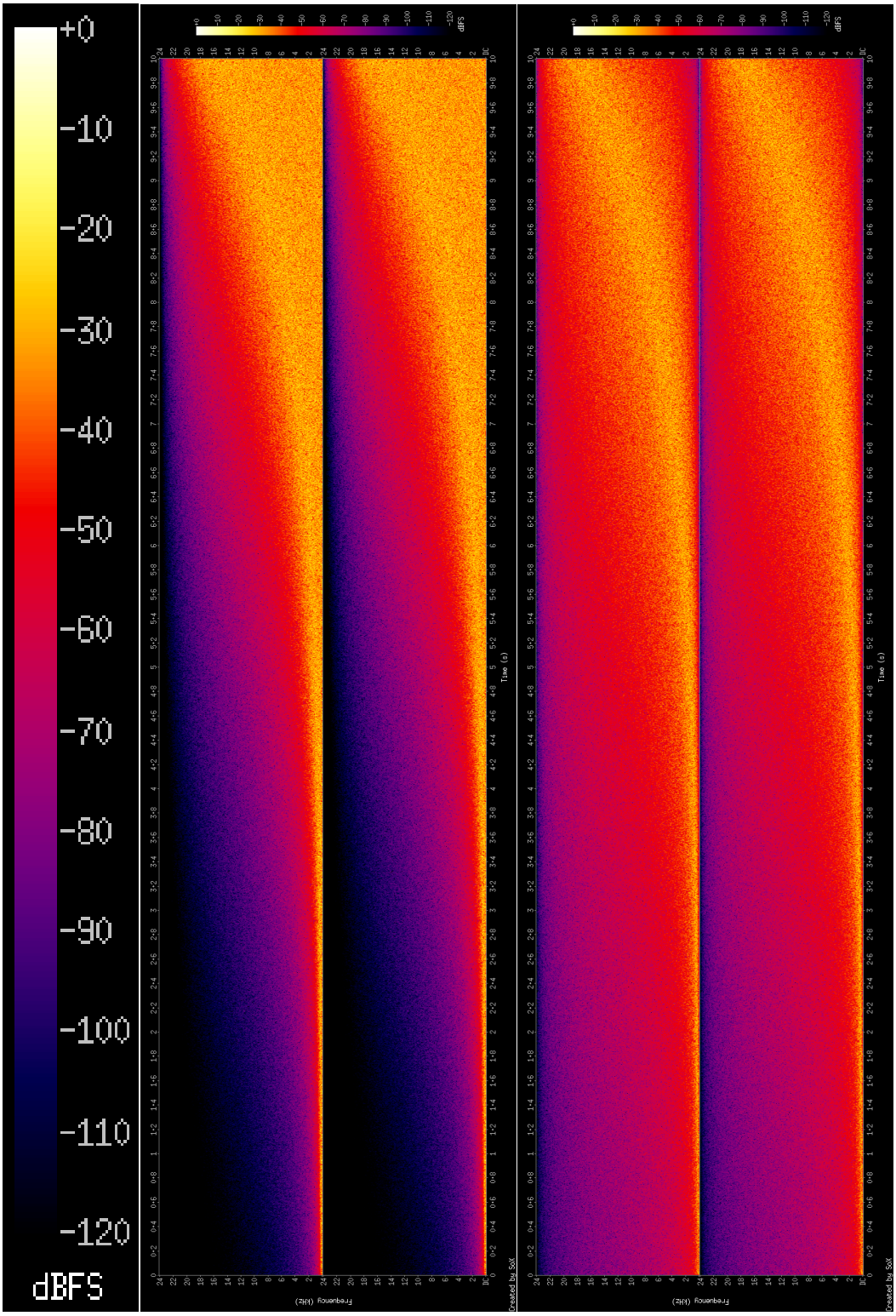

Figure 7.9: Spectrogram of a white noise filtered by a lowpass (upper) and bandpass (lower) biquad with increasing filter cutoff and constant neutral filter resonance.

7.8 Ring Buffer

The ring buffer is used as a synchronization element between the signal generated from the DSP section (the synthesizer's oscillators and filter) and the audio backend. A ring buffer is a fixed size FIFO queue with a write and read index. In contrast to a normal queue the index wraps around when it reaches the end of the queue as if both ends of the queue were connected. Implementing such a buffer in Rust proved to be more complicated than in traditional system languages like C because Rust's compiler assures exclusive write or read access to any data structure at any time [K+16, 4.9 References and Borrowing] in order to achieve its memory safety guarantees. Mutual exclusive access to the buffer by either a consumer (reader) or producer (writer) view was ensured by wrapping the buffers underlying array in a mutex [Lin16b, 'src/lib.rs', line 148].

The ring buffer's interface provides blocking and non-blocking read and write access and, in contrast to common ring buffer implementations, protects for over- and underflow by assuring that none of the read or write indexes is overtaking the other. A buffer underflow means that the read pointer overtakes the write pointer, e.g. if the producer is to slow, and an overflow is the exact opposite, i.e. the reader cannot consume the data fast enough. The under- and overflow protection is achieved by keeping track of the number free and filled slots (line 4 and 5 in section 7.8). The synthesizer implementation uses the blocking interface which allows the DSP thread to sleep if the whole buffer is filled. The implementation of the blocking interface does not use a naive busy-wait loop, instead, it uses conditional variables (`Condvar` in section 7.8) to signal either a consumer or producer view when there are slots free or data is available to be read.

```
pub struct SpscRb<T> {
    buf: Arc<Mutex<Vec<T>>>,
    inspector: Arc<Inspector>,
    slots_free: Arc<Condvar>,
    data_available: Arc<Condvar>,
}
```

Listing 4: [Lin16b, ‘src/lib.rs‘, lines 147-152]

It showed, by benchmarking the throughput of the ring buffer implementation, that the use of synchronization primitives did not introduce a significant amount of runtime overhead. In the benchmark 2 880 000 samples were pushed through the buffer [Lin16b, ‘benches/bench.rs‘], which equates to one minute of audio data at 48 kHz sampling rate, in 13 915 402 ns $\approx$ 14 ms[4] which is by several orders of magnitude faster than the required throughput-rate of the synthesizer that is equal to its sample rate. Hence, the ring buffer implementation is suitable for use in real-time audio applications.

7.9 Audio Output

There is a variety of different audio backend APIs, e.g. CoreAudio for Apple’s macOS, WASAPI is one of the audio APIs for Microsoft Windows and Linux has JACK, ALSA, PulseAudio and others. Implementing a separate binding for each of those C APIs is error prone and makes the portability of the code more difficult. Therefore, it is convenient to use a wrapper that provides a cross-platform abstraction for those audio backend APIs.

A popular choise which provides Windows, Mac and Linux support is *portaudio*

[4]The same machine was used for the benchmark as for the latency test in section 7.3.1.

[Ben+16] and, with rust-portaudio, a Rust binding for this library was also available at the time of writing. Nonetheless, with *rsoundio* [Lin16d], a Rust binding for *libsoundio* [Kel16a], another cross-platform library, was implemented. The decision for using *libsoundio* was made because it provides support for PulseAudio, has better error handling than *portaudio* (a more thorough comparison is provided by [Kel16b]).

Sound data is send to the sound card by a callback function which had to be registered in libsoundio's API. Libsoundio's API required to register a callback function before a sound stream could be opened, i.e. the callback was called each time the sound card requested more data. The same rules that apply for signal handlers in systems software apply to the callback function, the code should avoid memory allocations, complex operations and every form of blocking IO to prevent underruns of the sound card buffer. In general, the callback function should only copy data from the synthesizer's internal ring buffer into the sound card's buffer while casting each samples into another float or integer type if necessary.

Registering a callback funtion in *rsoundio* translates to storing a pointer to a Rust function into a C struct of *libsoundio* [Lin16d, 'src/stream.rs', lines 173-185]. Unfortunately, the execution of the callback function left the scope of the output stream struct [Lin16d, 'src/stream.rs'] that stored a pointer to the callback and also caused the structs memory to be freed automatically. This in turn caused a segmentation fault when the callback function returned. Those types of segmentation faults at runtime are hard to debug and preventing the output stream struct to be dropped at all will likely cause a memory leak. A simple solution was to set a boolean marker that prevented the struct from being freed in a callback context [Lin16d, commit #1fcdde5] but otherwise allowed the struct to be dropped.

Implementing a safe Rust wrapper for *libsoundio* has shown to be more complicated than expected but the final implementation could be used successfully as an audio backend for the synthesizer.

```rust
fn coeffs(w: Float, q: Float, filter_type: FilterType)
    -> ([Float; 2], [Float; 3]) {
    let (sinw, cosw) = (Float::sin(w), Float::cos(w));
    let (mut As, mut Bs) = ([0.; 2], [0.; 3]);
    let alpha = sinw / (2.0 * q);

    let a0 = 1. + alpha;
    As[0] = -2. * cosw;
    As[1] = 1. - alpha;

    match filter_type {
        FilterType::LP => {
            Bs[0] = (1. - cosw) / 2.;
            Bs[1] = 1. - cosw;
            Bs[2] = (1. - cosw) / 2.;
        }
        FilterType::HP => {
            Bs[0] = (1. + cosw) / 2.;
            Bs[1] = -1. - cosw;
            Bs[2] = (1. + cosw) / 2.;
        }
        FilterType::BP => {
            Bs[0] = alpha;
            Bs[1] = 0.;
            Bs[2] = -alpha;
        }
        FilterType::Notch => {
            Bs[0] = 1.;
            Bs[1] = -2. * cosw;
            Bs[2] = 1.;
        }
    }
    // normalize by dividing through a0
    for x in Bs.iter_mut().chain(As.iter_mut()) {
        *x /= a0;
    }
    (As, Bs)
}
```

Listing 3: Calculation of the biquad filter's coefficients [Lin16e, 'src/dsp/filter.rs', lines 47-78].

8 Summary

This work presented an overview of synthesis techniques and algorithms suitable for the implementation of a polyphonic real-time audio synthesizer. The evaluation of the synthesizer prototype showed that the chosen techniques achieved the required amount of audio quality and are efficient enough so that the instrument can be played with very low latency through MIDI and OSC controller hard- or software. Also, Rust proved to be an excellent choice for the implementation of real-time audio software, even though the language's ecosystem still lacks mature signal processing libraries.

8.1 Optimizations

It should be possible to compile the source code on platforms other than Linux, but those were not tested, hence a cross platform build setup could be developed. Moreover, commercial synthesizers usually provide a vast amount of modulation options, e.g. nearly every parameter of the instrument can be controlled from low frequency oscillators, envelope generators or integrated step sequencers, the possibilities are nearly endless. Adding an envelope generator to control the filter's cutoff frequency would greatly expand the range of sounds that can be created

with the synthesizer. A combination of different oscillator algorithms for specific waveforms could improve the performance of the application, especially the sine wave could be generated without the use of wavetables. Also, of great value would be the addition of a variety of sound effects to the synthesizer's audio chain, e.g. a reverb effect, a delay or one of its special forms like phaser and flanger, bit reduction or a distortion effect. The set of possible extensions and improvements is as large as the variety of sonic themes that can be produced, so the given ideas can be seen as a starting point.

8.2 Conclusion

I underestimated the amount of work that comes with such a project, especially for one with no previous experience in the development of signal processing software.

The closing sentence is a quote from my presentation of the master's thesis:

> *Implementing an audio synthesizer is serious work, if done right.*

Appendix

List of Figures

List of Tables

List of Listings

References

[13] *Piano Key C_4*. 2013. URL: https://www.freesound.org/people/Goup_1/sounds/176449/ (visited on 09/16/2016).

[16a] *portmidi*. 2016. URL: http://portmedia.sourceforge.net/portmidi/ (visited on 12/23/2016).

[16b] *WebAudio API*. 2016. URL: https://github.com/webaudio/web-audio-api/commit/a6842f2f733911a8ac6b330a405eac19878adc15 (visited on 12/22/2016).

[16c] *WebAudio biquad filter equation usage in Chromium*. 2016. URL: https://groups.google.com/a/chromium.org/forum/#!topic/chromium-os-checkins/nchPbrqfC2w (visited on 12/22/2016).

[16d] *WebAudio biquad filter equation usage in Firefox*. 2016. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1265395 (visited on 12/22/2016).

[Ade+03] Bernard D. Adelstein et al. "Sensitivity to Haptic-audio Asynchrony". In: *Proceedings of the 5th International Conference on Multimodal Interfaces*. ICMI '03. Vancouver, British Columbia, Canada: ACM, 2003, pp. 73–76. ISBN: 1-58113-621-8. DOI: 10.1145/958432.958448. URL: http://doi.acm.org/10.1145/958432.958448.

[Ass14] MIDI Manufacturers Association. "The Complete MIDI 1.0 Detailed Specification". In: Third Edition (2014). Document Version 4.2. The specification and all related practices are described as of 1996., p. 334.

[Ass16] The MIDI Association. *Midi.org*. 2016. URL: https://www.midi.org/ (visited on 07/22/2016).

[Bél16] Olivier Bélanger. *pyo*. 2016. URL: https://github.com/belangeo (visited on 06/20/2016).

[Ben+16] Ross Benecia et al. *portaudio*. 2016. URL: http://www.portaudio.com/ (visited on 12/24/2016).

[Ben08] David J. Benson. "Music: A mathematical offering". In: *The Mathematical Intelligencer* 30.1 (2008), pp. 76–77. ISSN: 0343-6993. DOI: 10.1007/BF02985765.

[Bra01] Eli Brandt. *Hard Sync witwith Aliasing*. 2001.

[Bri16] Robert Bristow-Johnson. *Cookbook formulae for audio equalizer biquad filter coefficient*. 2016. URL: http://shepazu.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html (visited on 10/09/2016).

[Cha85] Hal Chamberlin. *Musical applications of microprocessors*. 2nd ed. 1985, p. 802. ISBN: 0810457687.

[Cho73] John M. Chowning. *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation*. 1973.

[Com75] ISO/TC 43 Acoustics Comittee. *Acoustics – Standard Tuning Frequency (Standard Musical Pitch)*. ISO. ISO, 1975. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=3601.

[Das78] James Dashow. "Three methods for the digital synthesis of Chordal structures with non-harmonic partials*". In: *Interface* 7.2-3 (1978), pp. 69–94. ISSN: 0303-3902. DOI: 10.1080/09298217808570251. URL: http://www.tandfonline.com/doi/abs/10.1080/09298217808570251.

[Dev16] JACK Developers. *JACK Audio Connection Kit*. 2016. URL: http://www.jackaudio.org/ (visited on 12/22/2016).

[Ehr13] Fabian Ehrentraud. *SynOSCopy*. 2013. URL: https://github.com/fabb/SynOSCopy/wiki (visited on 07/25/2016).

[Fou16] Wikimedia Foundation. *Digital Filters*. 2016. URL: https://en.wikipedia.org/wiki/Digital_filter (visited on 01/10/2016).

[Fre10] Beat Frei. *Digital Sound Generation*. 2010, p. 85. URL: http://www.icst.net.

[FS09] Adrian Freed and Andy Schmeder. "Features and Future of Open Sound Control version 1.1 for NIME". In: *NIME*. Apr. 2009. URL: http://cnmat.berkeley.edu/node/7002.

[FZ07] H. Fastl and E. Zwicker. *Psychoacoustics: Facts and Models*. Springer series in information sciences. Springer Berlin Heidelberg, 2007. ISBN: 9783540688884. URL: https://books.google.de/books?id=eGcfn9ddRhcC.

[Gel10] Stanley A. Gelfand. *Hearing: An Introduction to Psychological and Physiological Acoustics*. Fifth Edit. Colchester, UK: Informa Healthcare, 2010, p. 480. ISBN: 978-1-4200-8865-6.

[Huo10] Antti Huovilainen. "Design of a Scalable Polyphony-MIDI Synthesizer for a Low Cost DSP". In: *Science And Technology* (2010).

[Int96] International Organization for Standardization, ed. *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. 1996.

[K+16] Steve Klabnik, Charol Nichols, et al. *The Rust Programming Language*. 2016. URL: https://doc.rust-lang.org/book/ (visited on 06/20/2016).

[Kel16a] Andrew Kelley. *libsoundio*. 2016. URL: https://github.com/andrewrk/libsoundio (visited on 12/22/2016).

[Kel16b] Andrew Kelley. *libsoundio vs PortAudio*. 2016. URL: https://github.com/andrewrk/libsoundio/wiki/libsoundio-vs-PortAudio (visited on 12/24/2016).

[Kol16] Con Kolivas. *Multiple Queue Skiplist Scheduler Announcement*. 2016. URL: https://lkml.org/lkml/2016/10/29/4 (visited on 12/20/2016).

[KS83] K Karplus and A Strong. "Digital Synthesis of Plucked String and Drum Timbres". In: *Computer Music Journal* 7.2 (1983), pp. 43–55.

[LB07] Michael Lester and Jon Boley. "The Effects of Latency on Live Sound Monitoring". In: *Audio Engineering Society Convention 123*. Oct. 2007. URL: http://www.aes.org/e-lib/browse.cfm?elib=14256.

[Lii16] Liine. *Liine Lemur Controller*. 2016. URL: https://liine.net/en/products/lemur/ (visited on 07/21/2016).

[Lin16a] Andreas Linz. *portmidi-rs rewrite*. 2016. URL: https://github.com/musitdev/portmidi-rs/pull/24/files (visited on 12/23/2016).

[Lin16b] Andreas Linz. *rb*. 2016. URL: https://github.com/klingtnet/rb (visited on 12/22/2016).

[Lin16c] Andreas Linz. *rosc*. 2016. URL: https://github.com/klingtnet/rosc (visited on 12/22/2016).

[Lin16d] Andreas Linz. *rsoundio*. 2016. URL: https://github.com/klingtnet/rsoundio (visited on 12/22/2016).

[Lin16e] Andreas Linz. *ytterbium*. 2016. URL: https://github.com/klingtnet/ytterbium (visited on 12/22/2016).

[Loy11] Gareth Loy. *Musimathics, Volume 2: The Mathematical Foundations of Music*. 2011.

[Mol07] Ingo Molinar. *Completely Fair Scheduler Announcement*. 2007. URL: https://lwn.net/Articles/230501/ (visited on 12/22/2016).

[Moo76] J. A. Moorer. "The Synthesis of Complex Audio Spectra by Means of Discrete Summation Formulae". In: *Journal of the Audio Engineering Society* 24 (1976), p. 717. URL: https://ccrma.stanford.edu/files/papers/stanm5.pdf.

[Nam+09] Juhan Nam et al. "Alias-free Virtual Analog Oscillators Using a Feedback Delay Loop". In: *Proceedings of the 13th Int. Conference on Digital Audio Effects (DAFx-10)* (2009), pp. 1–6.

[Ota15] Francisco J. Valencia Otalvar. "Aliasing and Harmonic Decay Control of the Bandlimited Step Method Using Psychoacoustic Models and the Genetic Algorithm". PhD thesis. University of Miami, 2015, p. 256. URL: http://mue.music.miami.edu/wp-content/uploads/2012/11/Fransisco-Thesis.pdf.

[Pav13] David Svoboda Pavel Karas. "Algorithms for Efficient Computation of Convolution". In: *Design and Architectures for Digital Signal Processing*. 2013, pp. 179–208. ISBN: 978-953-51-0874-0. DOI: 10.5772/3456. URL: http://www.intechopen.com/books/design-and-architectures-for-digital-signal-processing.

[Pek07] Jussi Pekonen. "Computationally Efficient Music Synthesis – Methods and Sound Design". PhD thesis. Electrical and Communications Engineering, 2007, pp. 1–92. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.8753%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[Pek14] Jussi Pekonen. "Filter-Based Oscillator Algorithms for Virtual Analog Synthesis". PhD thesis. Aalto University, 2014, p. 72. ISBN: 9789526055886. URL: https://aaltodoc.aalto.fi/handle/123456789/12835.

[Phi16] Andreas Linz Philippe Delrieu. *portmidi-rs*. 2016. URL: https://github.com/musitdev/portmidi-rs (visited on 07/22/2016).

[PM07] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing*. Pearson Prentice Hall, 2007, p. 1084. ISBN: 0133737624.

[Puc06] Miller Puckette. "The Theory and Technique of Electronic Music". In: *Puckette, M. (2006). The Theory and Technique of Electronic Music. World, 11, 1–337. doi:10.1186/1471-2105-11-50World* 11 (2006), pp. 1–337. ISSN: 14712105. DOI: 10.1186/1471-2105-11-50. URL: http://www.amazon.com/Theory-Technique-Electronic-Music/dp/9812700773.

[Rab+72] Lawrence R. Rabiner et al. "Terminology in Digital Signal Processing". In: *IEEE Transactions on Audio and Electroacoustics* 20.5 (1972), pp. 322–337. ISSN: 00189278. DOI: 10.1109/TAU.1972.1162405.

[Roa96] Curtis Roads. *The Computer Music Tutorial*. Cambridge, MA, USA: MIT Press, 1996. ISBN: 0262680823.

[Smi16] Julius Orion Smith III. *Introduction to Digital Filters with Audio Applications*. 2016. URL: https://ccrma.stanford.edu/~jos/filters/filters.html (visited on 01/10/2016).

[Smi85] Julius Orion Smith III. *Introduction to Digital Filter Theory*. 1985. URL: https://ccrma.stanford.edu/papers/introduction-digital-filter-theory.

[Sof15] Max Planck Institute for Software Systems (MPI-SWS). *Rust Belt*. 2015. URL: http://plv.mpi-sws.org/rustbelt/ (visited on 06/20/2016).

[SS96] Tim Stilson and Julius Smith. "Alias-free digital synthesis of classic analog waveforms". In: *Proc. International Computer Music ...* (1996), pp. 1–12. URL: https://ccrma.stanford.edu/files/papers/stanm99.pdf%7B%5C#%7Dpage=48.

[Ste+92] K Steiglitz et al. "Meteor". In: 40.8 (1992), pp. 1901–1909.

[VH06] Vesa Välimäki and Antti Huovilainen. "Oscillator and Filter Algorithms for Virtual Analog Synthesis". In: *Computer Music Journal* 30.2 (2006), pp. 19–31. ISSN: 0148-9267. DOI: 10.1162/comj.2006.30.2.19.

[WDJ97] Alan West, Charles Dodge, and Thomas Jerse. *Computer Music, Synthesis, Composition, and Performance*. Vol. 2. 1997, p. 475. ISBN: 0028646827. DOI: 10.2307/3680102. URL: http://www.jstor.org/stable/3680102?origin=crossref.

[Wel16] Allen Welkie. *RustFFT*. 2016. URL: https://github.com/awelkie/RustFFT (visited on 12/22/2016).

[Wis14] Aaron Wishnick. "Time-Varying Filters for Musical Applications". In: *DAFx* 1 (2014), pp. 1–8.

[Wri02] Matthew Wright. "Open Sound Control 1.0 Specification". In: (2002). URL: http://opensoundcontrol.org/spec-1_0.

Erklärung (Statement of Originality)

I hereby declare, that this master's thesis is original work performed by me, and that all resources and references to other works that I used are properly and duly cited. I am aware, that any kind of infringement can, even retroactively, entail the revocation of my degree.

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses füh-ren kann.

_____ Leipzig, 8. Januar 2017

Andreas Linz