Thanh Do

Overview of Digital Signal Processing in Audio Application Development

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

15 May 2020

Author Title Number of Pages Date	Thanh Do Overview of Digital Signal Processing in Audio Application Development 62 pages 15 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Antti Piironen, Principal Lecturer

Digital signal processing has been playing an indispensable role in the domain of ICT. The objective of this Bachelor's thesis is to provide an overview of the field and its applications. To attain a thorough insight into the topic, this document consists of a theoretical summary of digital signal processing. It also describes the development of an audio application, in order to illustrate the practical aspect.

The thesis discusses the field of digital signal processing from multiples perspectives, including the historical context, theoretical knowledge, and practical applications. To provide a foundational understanding of the field, fundamental theories are summarized briefly before discussing a development project. The project focused on building modules for VCV Rack, an open-source digital synthesizer written in C++. The study was conducted as individual work and is largely based on textbooks and open-source materials.

The outcome of the project is a set of modules for a digital modular synthesizer with which elemental signal-based sounds can be created and modified extensively. The set of modules is able to demonstrate signal processing and its application in the field of digital audio programming. In addition, the project offers potential for further development where additional extensions and customizations can be made for the modules. Thus, this study serves as a stepping stone for deeper learning in the immense field of signal processing, and also as a trigger for enthusiasm for the field.

Keywords	Digital signal processing, audio programming

Contents

List of Abbreviations

DSP	Digital signal processing
ICT	Information and communication technology
LTI	Linear time-invariant, usually refers to linear time-invariant systems
VCO	Voltage-controlled oscillator
VCA	Voltage-controlled amplifier, or voltage-controlled attenuator
VCF	Voltage-controlled filter
PCM	Pulse-code modulation
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
LFO	Low frequency oscillator
MIDI	Musical instrument digital interface

1 Introduction

It is a fact that an immense part of modern life, and its increasingly advanced technologies, are dependent on information. The possession of information and communication ability have become one of the most crucial assets in the 20th and 21st centuries. Thus, the advancement of Information Communication Technology (ICT) has been inevitable, and its significance is great and will grow greater in the foreseeable future.

Digital signal processing is undoubtedly the most important field of study in the area of ICT, and it has started to gain popularity since the second half of the 20th century. It does not merely provide humans the power of telecommunication, but it also gives people the capability to thoroughly utilize the gigantic amount of information that is possible to gather. The applications of digital signal processing, such as data compression or image processing, are countless, and their impacts are undeniable. [1, pp. 1-10.] Without digital signal processing, or DSP, there is no solid foundation for the current modern era, an era when everything revolves around the acquisition of information and how to make use of it.

The field of DSP has been expanding both vertically and horizontally. Nowadays, signal-processing applications can be found, among others, in the medical field, military operations and space exploration. Such extensive usage is the result of increasing achievements in DSP techniques and research. The emergence of the digital computer has opened up new DSP techniques, and it also makes many others applicable. [1, pp. 1-10.] The limitations of analog signal processing have been overcome using digital approaches, and the depth of the field has now become bottomless.

The aim of the project discussed in this thesis is to provide a fundamental understanding of digital signal processing and its application. The thesis can only scratch the surface of this immense field, but such elemental information will lay the foundation for deeper study of signal processing and related fields. The development of an audio application, a simulation of an audio synthesizer and compatible processing modules in particular, was included in this project. An objective of the project was not only to give a thorough overview of the field but to also demonstrate the practical and compelling aspects of DSP.

2 Theoretical background

Digital signal processing, or signal processing in general, has gradually become a major force in the catalogue of emerging technologies of the 20th century. Signal processing, with its potential of handling information, offers a great diversity of practical applications in the modern era. It started out with analog circuits and heavily lied in the domain of electrical engineering. However, with the appearance of digital computers around the mid-1900s, the interest arose significantly with the option of digital approaches. Digital signal processors are more easily produced, flexible, and tolerant than their analog component counterparts; hence, DSP became a rival of the traditional way. [2, pp. 37-38.]

Like many other fields, the development of digital signal processing began with an experimental phase. Its early usages were mostly to replicate previous analog methods. In the 1960s, the maturity of DSP made it a standalone field. [2, pp. 37-38.] Since then, it has been advancing, both theoretically and practically, at a rapid speed, with many innovative techniques, mathematics, and algorithms being developed and applied, in order to constantly optimize the performance and push the limits.

Consequently, the evolution of signal processing drastically shifted the landscape of specific domains. One of such heavily influenced domains is music and audio applications. With the advancement of software and DSP-based hardware, the post-processing of audio and music mixing were facilitated. Furthermore, it was crucial that sounds and audio became programmable on both a hardware and a software platform. Pioneering musicians in the 60s and 70s introduced electronic music using synthesizers and computer-generated sounds, which based on various means of DSP handling. For instance, it could be sampling generation, filtering, or a range of different synthesis techniques. [3.] Additionally, many programming languages were specifically developed for audio programming, such as ChucK or Pure Data. Thus, nowadays, DSP and audio programming continue to flourish, and play a key role in audio applications and the general music scene.

The goal of this thesis is to present the fundamentals of DSP and its practical usage in audio applications. This chapter specifically aims to provide a general perspective of the field, and a brief history. Additionally, audio programming and audio applications are dealt with. Thus, the thesis provides a sufficient theoretical background for the discussed fields in general, and also for the project development included in later chapters.

2.1 Introduction to digital signal processing

Digital signal processing, as its name implies, is the science of manipulating signals. Mitra notes that “signal processing is concerned with the mathematical presentation of the signal and the algorithmic operation carried out on it to extract the information present” [2, p. 1] The signals need to be sampled before processing, which can be in either digital or analog format. Depending on the format of the sampled data, pre-processing or post-processing conversion methods are needed, which commonly means using an analog-to-digital and a digital-to-analog converter. [2, pp. 1-3.]

The domain of DSP is different from other subfields of computer science by tackling a unique and abundant information unit – signal. A signal is a phenomenon evolving over time or space. Therefore, it can have different abstract representation, or mathematical forms, and the chosen abstraction for the signal directly affects the information that is extracted from it. [4, p. 1.]. Signals are an indispensable part of the world today, and the signals that are commonly encountered are usually analog signals created by natural means. To render digital approaches applicable, analog signals are digitized. [1, pp. 1-10, 4, pp. 1-3.]

This study will investigate the elemental understandings of digital signal processing in later chapters. The current section, however, is dedicated to providing a general introduction of the field, its history of development and related applications.

2.1.1 A brief history and overview of studies of DSP

Many DSP techniques can be traced back to centuries ago, but only in the mid-1900s they began to attract attention. Over the decades, the field has gone through an active history, with many phases. This report discusses only certain timeframes that are considered significant to the development of the field.

2.1.1.1 1948 and 1949

1948 was one of the most significant years in the history of DSP. In this year, Claude Shannon published the revolutionary article “A mathematical theory of communication”, which was heavily influenced by Harry Nyquist’s 1928 paper “Certain topics in telegraph transmission theory”. In the article, he interprets communication as the process of transmitting messages between a source and a receiver via a channel. It studies many aspects of information, such as quantification and communication. This is considered the birth of the information theory and DSP. [5, pp. 2-5.]

Also, in 1948, Shannon co-authored the publication “The philosophy of PCM”. The study focuses on transmission using on-off pulses, which later became significantly important during the expansion of optical fiber usage and computers [6]. Additionally, the year 1948 marked the foundation of IEEE Signal Processing Society, which has been a major force of signal processing community since. [5, pp. 2-5.]

Following his ground-breaking papers in 1948, Shannon published “Communication in the Presence of Noise” in 1949. This paper laid the foundation for the development of geometric communication systems, and also introduced the Shannon-Nyquist sampling theorem for conditioning proper sampling. The paper defines the fundamental limits of signal processing and establishes a link between continuous and discrete signals, which are the groundwork for all DSP studies. [6, 7.]

2.1.1.2 1960s and the birth of modern DS

The 1960s witnessed the turning of DSP into a fully mature field. The field broke out of its experimental phase, as researchers came to the realization that signal manipulation in the digital domain closely resembles operations in the analog domain. Digital processing finally started to show its true potentials. [5, p. 15.]

In addition to this, in 1965, James Cooley and James Tukey published the discovery of the Fast Fourier Transform algorithm, which became one of the most extensively used techniques in DSP. By transforming and computing the signal between frequency and time domains, the algorithm can achieve the same results as many common operations, but hundreds of times faster. As Alan V. Oppenheim noted, the paper “marked the beginning of the field of DSP in its modern form” [8, p. 36].

2.1.1.3 1970s and early studies of DSP

The 1970s, with the advancement of technologies during the Third Industrial Revolution, such as microprocessors, impelled DSP to grow exponentially. In this decade, the first real-time DSP computer was introduced, followed by innovations in a myriad of areas. Echo cancellation and sub-band coding are a few of many advances in speech processing, in addition to new digital filter designs, namely multirate filters. [5, pp. 26-27.]

On the other hand, circuit technology also kept evolving. Algorithms which had been restricted by analog hardware were rendered possible in practice. One of the landmarks was the exemplification of sampled-data-processing by the paper on CCD and switched-capacitor circuits, which eliminates the requirement of analog-to-digital conversion. [5, pp. 27-28, 9.]

2.1.1.4 From 1990s

In the 1990s, the direction towards interconnection, with the development of the telephone network, further emphasized the significance of DSP. The exposure of DSP technology to the general public resulted in the growth of the consumer electronics market. A few of the iconic products of the time are pagers, camcorders, and digital cameras. This demanded the rapid growth of DSP innovations, whilst the advancement of integrated circuit technology with chips components thrived as Moore’s Law stated. Thus, a cycle of booming for both DSP and its market was created. [5, pp. 44-46.]

2.1.2 Applications of DSP in ICT

In this section, the paper lists a few technologies that apply DSP techniques. These serve as representative examples for the practical use cases of the domain.

2.1.2.1 Radars

‘Radar’ stands for Radio Detection and Ranging, a detection system using radio waves to identify objects’ distance and location relative to the location of the system. A radar detects an object by propagating directional radio waves, which are generated using a radio transmitter. A receiving antenna placed near the transmission site will gather reflected signals from any object that the initial wave possibly strikes and measure the distance to that object based on the delay time of the echo. [10.] Figure 1 illustrates a standard radar system and its key components.

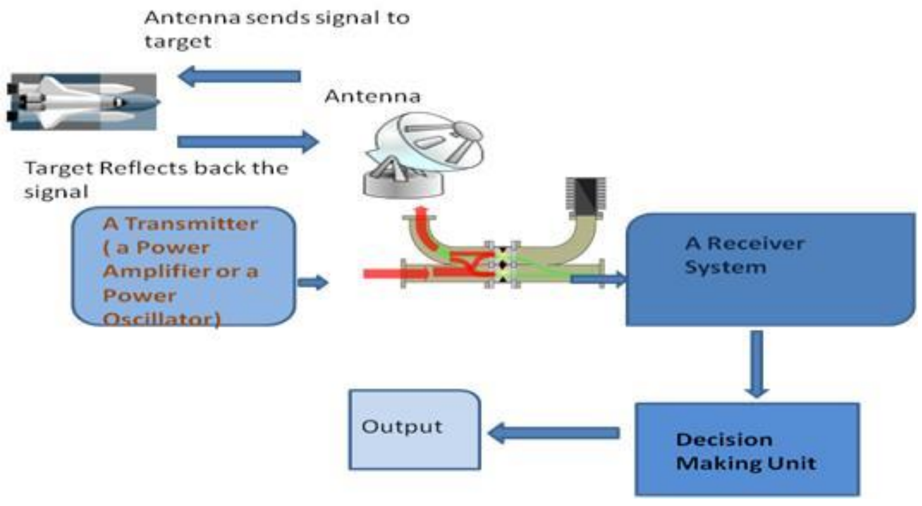
Antenna sends signal to target
Antenna
Target Reflects back the signal
A Transmitter (a Power Amplifier or a Power Oscillator)
A Receiver System
Output
Decision Making Unit

Figure 1. Example of a radar system. Copied from [10].

A major aspect of radar design is the tradeoff between the reachable range of the initial wave and noise level in the echo signal [1, p. 7]. Radio waves created from a longer pulse can travel further, but the precision for elapsed time measurement will be reduced.

Using DSP, pulses can be compressed to improve reachable distance, and noise in a reflected signal can be filtered out. Both conflicting factors are upgraded to avoid compromising when designing the system. Furthermore, a range of pulse length can be selected rapidly to suit specific problems. [1, p. 7.]

2.1.2.2 Image processing

Another popular application of DSP is image processing. Unlike the usual representation of signals which is a time-varying parameter, image signals are the measure of a parameter over a surface. The amount of information that is encoded in an image is quite large, and it is sampled to unit of pixel. [1, pp. 9-10, 373-376.] Image processing applications using DSP vary from domain-specific areas, such as the medical field, to more common uses, such as multi-media graphic adjustments.

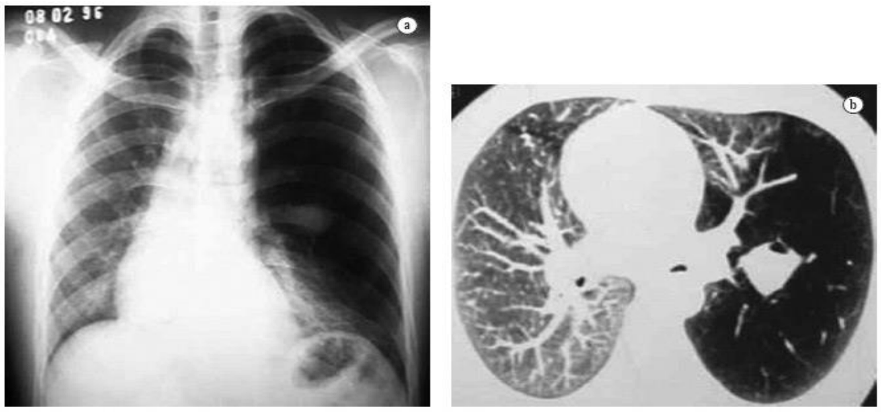

Figure 2. From left to right. X-ray and CT scan from the same case of bronchial atresia with mucocele. Modified from Irion (2020) [11].

A classic example of DSP's medical usage exploiting image processing is the computed tomography (CT) scanner, a now common equipment for diagnosis. It revolutionized the traditional way of inspecting the human body of a living person by forming images from passed X-rays with a digital approach. The signals from detected X-rays are converted and handled digitally on a computer; thus, a considerably higher amount of details can be seen in multiple dimensions in the obtained images. [1, pp. 9-10.] Figure 2 shows the difference between X-ray and CT scanned images.

2.1.2.3 Compression

Telecommunication is an essential field of ICT. The key concerns of all telecommunication applications are the transfer of information between locations, the load of data conveyance, quality of data, and cost. DSP offers a variety of solutions such as multiplexing and echo control. [1, pp. 4-5.] These techniques do not only reduce the costs by using digital counterparts for the conventional analog way, but they also improve the quality of transferred data, which is commonly audio, over a long distance.

Compression is a universal method for increasing the amount of data transferred using the same bandwidth. Raw sampled signals contain a lot of redundant information, and compressing data is to remove such redundancy for delivery. An audio compressor consists of an amplifier that diminishes signals with the amplitude exceeding a certain threshold according to a certain ratio. [1, pp. 4-5, 2, pp. 22-23.] Figure 3 illustrates how this works. To restore the compressed signal to its initial profile, compression algorithms are complemented with corresponding decompression methods.

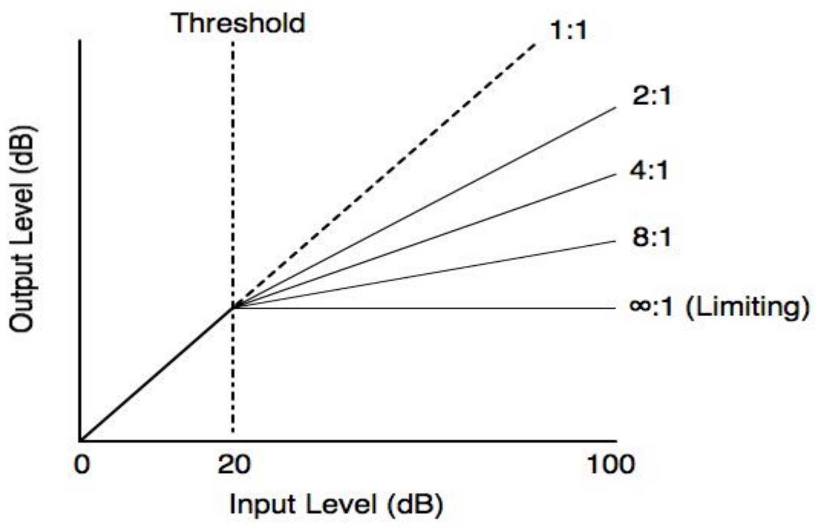

Figure 3. How threshold level and compression ratio typify an audio compressor. Copied from [12].

2.2 History and approaches of audio programming

Audio programming first flourished with audio systems were built on analog hardware in order to achieve a level of flexibility in creating sound. In the 1960s, with the beginning of electronic music and analog synthesizers, audio programming attracted more attention, especially in music. The synthesizers at the time supported interconnecting and interchangeable sound processing modules, and programming sounds followed a block-based paradigm. [15.] Audio programming, then, had this analog, modular approach as its backbone. Even in the present time, such designs are still playing an important role in modern audio.

Stepping into the digital era, the focus has partly shifted to the interaction with the computer when it comes to sound-making. The computing power of computers, with the development of operating systems in the 1970s and the 1980s, makes working on audio digitally appealing. Early sound-specific programming languages appeared, for instance, Common Lisp Music (CLM) and Csound. Additionally, libraries for sound processing, namely Cmix, were also developed. [15.]

Since then, audio programming has been further evolving. The processing power of modern computers and the need for online control has stimulated the growth of real-time systems and strongly-timed programming language. To match the creativity of sound-making, for example, in live performances of electronic music, languages have become more high-level. They are also packed with signal-processing supports. Figure 4 shows a program in Pure Data, a visual programming language for multimedia. [15.]

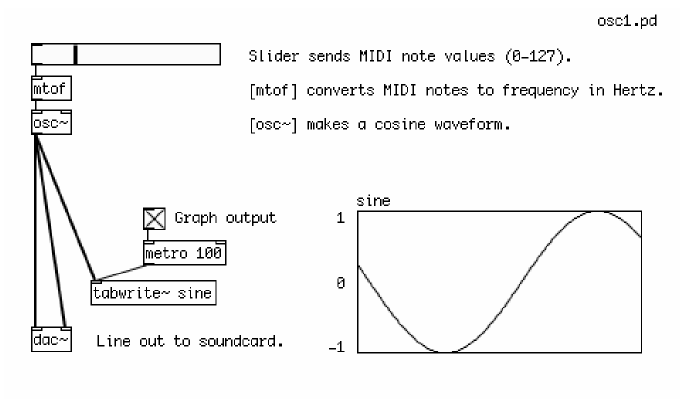
Slider sends MIDI note values (0-127).
[mtof] converts MIDI notes to frequency in Hertz.
[osc~] makes a cosine waveform.
Line out to soundcard.

Figure 4. A patch for making pure tone in Pure Data. Copied from [14].

3 Overview of fundamentals of DSP field

3.1 Understanding signal

It is noteworthy that signals are elemental units of digital signal processing. Thus, the understanding of signal is a prerequisite for any other knowledge about the world of signal processing.

In the previous section, we have learnt that the nature of signals is physical phenomena evolving over time. In mathematical terms, a signal is a mathematical function describing how dependent variables are related to independent variables, usually time. The values of dependent variables at a specific value of independent variables are called amplitudes. There can be single or multi-dimensional signals. For example, an image can be broken down to 2-D signals, and they can be represented using a function with two spatial dependent variables. [1, pp. 11-13, 2, p. 2.]

For most of the discussion in the next sections, the focus is placed on the single-dimension signal. It is the most fundamental abstraction of a signal, and it is also the representational form of audio, the subject of this thesis.

3.1.1 Classification of signals

Signals can be categorized using many standards, but mostly there are two types in DSP: continuous and discrete signals. This classification is based on the range of independent variables' value, and it directly affects the mathematics of signal processing.

Continuous signals

Continuous signals are commonly called analog signals. This type of a signal has a continuous range of value for independent variables. For example, with a continuous-time signal, this means there is an amplitude specified at every instance of time [2, p. 2]. Continuous signals exist mostly in nature.

Discrete signals

Accordingly, discrete signals, or also called digitized signals, are signals with a discrete range of value for independent variables. As a result, the signal is only defined at a certain value of an independent variable. Discrete signals are generally computer-generated or created by sampling continuous signals. DSP works mostly with this type of signals. [2, p. 2.]

Sampling

Sampling in signal processing means converting continuous to discrete signals. Since real world information exists mainly in analog form, sampling plays a crucial role in practical applications of DSP.

The significance of sampling is emphasized by the Nyquist-Shannon sampling theorem. The theory states that a signal is properly sampled only if the sampled signal does not contain frequency components above one-half of the sampling rate. In other words, to maintain the integrity of information and the underlying process in the original signal when sampling, the minimum sampling rate must be twice the bandwidth. The failure to attain this lower bound can lead to aliasing, or a phenomenon where the frequency of a component sinusoid is misrepresented. Figure 5 illustrates the differentiating results between proper and improper sampling.

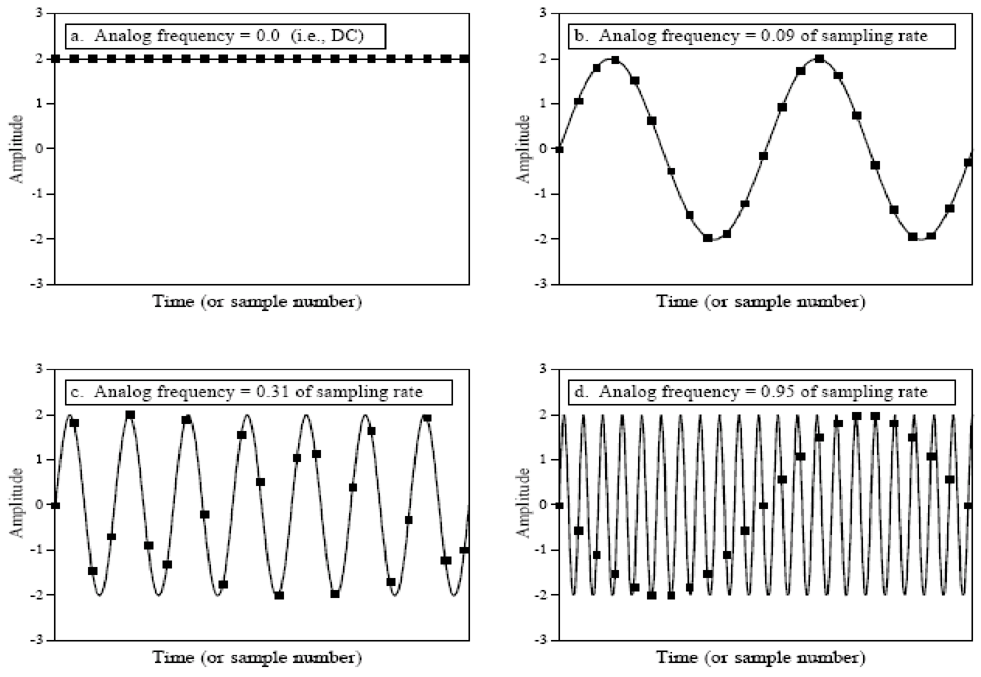

Figure 5. Signal sampling and aliasing. Modified from Smith (1999) [1, p. 41].

In Figure 5, from a to c, the signal is properly sampled. In d, however, aliasing happens. Sinusoid of frequency 0.95 is misinterpreted as sinusoid of lower frequency.

It is important to notice that for discrete signals, there can be sampled-data signals and digital signals. The former has continuous-valued amplitudes, whereas the latter has quantized amplitudes which means the values are rounded or truncated in a digital system.

3.1.2 Information representation and signal processing

Signals contain information, and the way this information is represented is essentially linked to the mathematical form of signals. Different forms emphasize different aspects of information, which directly affects how processing algorithms can be applied. Depending on the independent variable being used, signals can be described on separate domains.

3.1.2.1 Time domain

A signal is represented in a domain when it uses time as an independent variable. As a result, this method of representation describes how a signal changes through time. Discrete-time signals are usually obtained by periodically sampling continuous signals into finite sequences of samples, whereas a sample numerical value $x[n]$ is the signal amplitude at the time instant n with -∞ < n < ∞. It is a common practice for n to have an integer value and represent the nth sample in the sequence. [2, pp. 42-43.]

Time-domain representation is important for sampling signals. Additionally, many operations in signal processing are applied directly in the time domain, such as addition, modulation, and convolution.

3.1.2.2 Frequency domain

Another alternative domain for representing signals is the frequency domain. Time-domain signals can be mapped to the frequency domain without losing information using the Fourier transform. The next section will discuss this technique and how to obtain frequency-domain representation in detail, but in the simplest terms, it is possible to deconstruct any signal to a collection of individual sinusoidal signals with a unique integer frequency. To represent a signal in the frequency domain means specifying the amplitude for each component sinusoid of the original signal. Figure 6 demonstrates the physical interpretation of this.

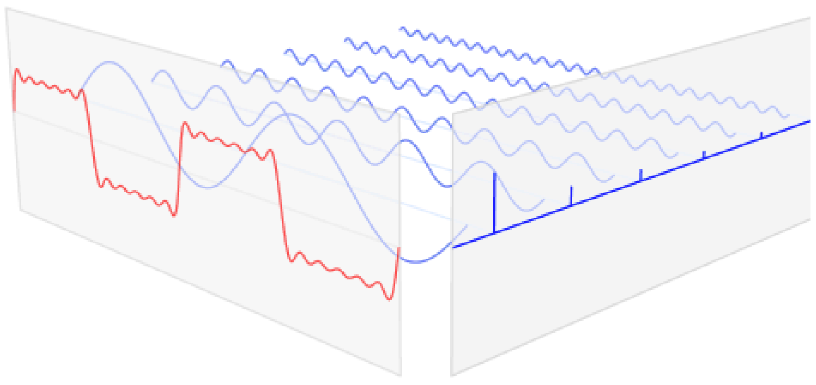

Figure 6. How Fourier transform maps time-domain signal (red) to frequency domain (blue). Modified from Barbosa (2020) [15.].

According to the sampling theory mentioned in the previous section, it is noted that error-free discrete signals can only contain sinusoidal components with maximum frequency of half the sampling rate. Thus, the value of n sample in the frequency-domain signal representation is the amplitude of the component sinusoid with the frequency n, where $0 < n \leq f_s$ and f_s is the highest frequency sampled. Using the Nyquist rate, this can be normalized to a proportion of a sampling rate in the range of (0, 0.5]. The frequency zero component is usually a constant, or a DC value, in the signal.

3.1.2.3 Fourier transform

The Fourier transform is the mathematical technique to map a time-domain signal to its frequency domain. In other words, the Fourier transform decomposes a signal to its collective sinusoids, which is illustrated in Figure 5. This is the basis for many techniques since it greatly optimizes computation in signal processing.

The reason for using sinusoids, instead of other waveforms, as elemental components of the Fourier transform is sinusoidal fidelity. A system taking a sinusoid as an input always produces a sinusoid as an output with the same frequency and wave shape. This property is tremendously useful for the Fourier transform, as it allows a variety of operations to be executable in the frequency domain. [1, pp. 141-142.]

Categories of Fourier analysis

The Fourier transform can be categorized into the following four types, depending on the characteristics of the signal.

- Fourier Transform
- Fourier Series
- Discrete Time Fourier Transform
- Discrete Fourier Transform

The differentiation of these categories is illustrated in Figure 6.

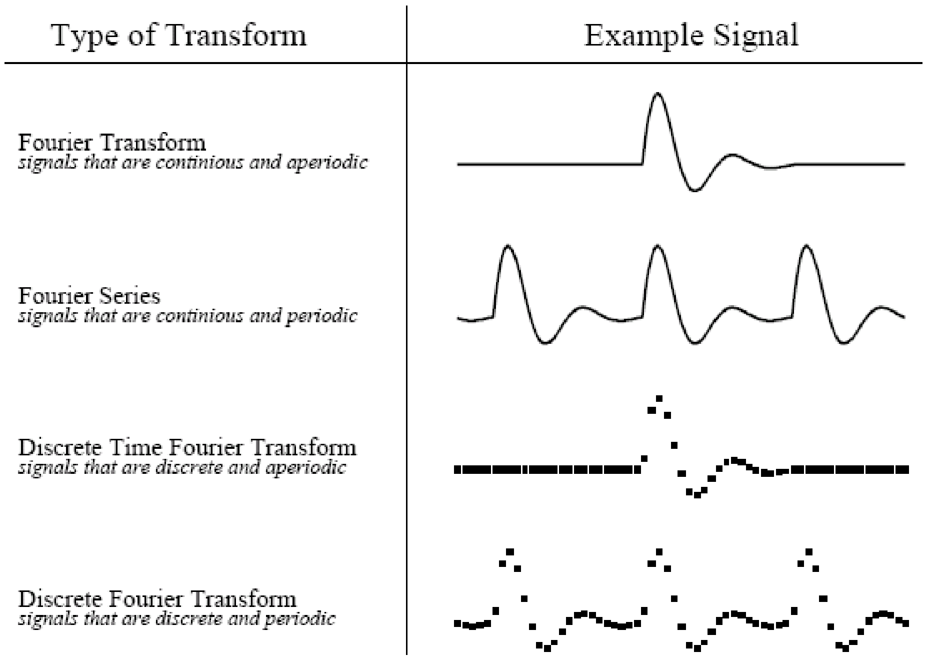
Type of Transform
Example Signal
Fourier Transform
signals that are continious and aperiodic
Fourier Series
signals that are continious and periodic
Discrete Time Fourier Transform
signals that are discrete and aperiodic
Discrete Fourier Transform
signals that are discrete and periodic

Figure 7. Family of Fourier analysis for signals with different continuity and periodicity. Modified from Smith (2020) [1, p. 145].

Nonetheless, not all of these techniques are applicable. It is essential to understand that the signal involved in the Fourier analysis extends to both negative and positive infinity. However, the limit of memory and computation in computer on the computer make it impossible to process an infinite amount of information. Thus, the computer can only work on a finite number of samples at a time. In other words, practical DSP can only use the Fourier transform for signals synthesized from a finite number of sinusoids. This eliminates the analyses involving aperiodic or continuous signals, since an infinite number of samples is needed for the transformation. Hence, for computer algorithms, only DFT, or Discrete Fourier Transform, is applied. Due to the limited length of this thesis, this is the only Fourier analysis technique being discussed. [1, pp. 143-145.]

DFT calculation

The basis function of DFT can be written in polar form using a complex number, as shown in Equation 1 below. In this formula, k is the component frequency, N is the number of samples being used for DFT calculation, and $x[n]$ is the time domain signal. Corresponding to this, the reconstruction of the original time-domain signal can be done based on using inverse DFT, which is shown in Equation 2 below. Using Euler's formula to further analyze these equations, the contribution of cosine and sine waves will be more apparent.

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]e^{-\frac{j2\pi kn}{N}} \tag{1}$$

$$x[n] = \sum_{k=0}^{N} X[k]e^{\frac{j2\pi kn}{N}} \tag{2}$$

It is noteworthy that k runs from 0 to $N-1$. However, the sampling theory only allows maximum frequency of N / 2. This leads to the complex DFT containing negative exponents from N / 2 to N, which mirrors the real part. Then, due to the periodic nature of the discrete signal being used in the DFT, this spectrum will repeat itself to both negative and positive infinity, as the DFT view time-domain signal as circular. [1, pp. 225-227, 570-575.]

In common application, DFT is mostly calculated using the Fast Fourier Transform (FFT). The FFT is an algorithm based on the complex DFT that can significantly reduce the computational time for the Fourier transform.

Properties of Fourier transform

Being the mathematical tool for conversion between the time and frequency domain, the Fourier transform consequently establishes the relationship of changes between them. Its properties bridge the calculation between two domains, which is extremely important in signal processing.

The Fourier transform possesses linearity, which is exhibited in its homogeneity and additivity. Changes in amplitude, or the addition of signals in one domain will result in similar changes and addition in the other. However, the Fourier transform is not shift invariant. The DFT is applied to periodic signals; hence, the domain signal is circular. Thus, shifting samples in the time domain will change the phase of sinusoids in the frequency domain. [1, pp. 185-188.]

Apart from linearity, convolution in one domain is equivalent to multiplication in the other. For example, the multiplication of time-domain signals can be done by convolving the same signals in the frequency domain. In fact, this is the basis for amplitude modulation, where two signals are merged into a third signal with frequency-related characteristics of both. Amplitude modulation was very common in the development of the audio application discussed in this thesis. [1, pp. 204-206.]

It is noteworthy that there is also a relationship between the sampling length in the time domain and the resolution of the normalized frequency spectrum. As the number of samples being used for the DFT reaches infinity, the resolution of the independent variable in the frequency domain increases. For an infinitely long signal, which makes it aperiodic, the frequency domain eventually becomes continuous. Expansion in one domain results in compression in the other. [1, pp. 200-203.]

3.2 DSP systems

Systems are processes that produce an output signal in response to their given input signal. Understanding systems and their properties is extremely important in the field of DSP. In fact, most practical DSP applications are about designing systems which determine how the wanted output is generated from the intake data.

3.2.1 Linear time-invariant systems

Generally, for the sake of simplicity, this report mainly focuses on linear time-invariant systems (LITs). They are mathematically straightforward, modular, and convenient for most DSP techniques and practical applications.

3.2.1.1 Properties of LTI systems

An LTI system is defined by its linearity and time-invariance. Much like linearity in the Fourier transform, a linear system must be both homogenous and additive. Additionally, linear systems usually possess shift invariance, which means the shifted input signal will result in the shifted output, though it is not a compulsory requirement. [1, p. 90.]

Homogeneity

Homogeneity means changes in amplitude are identical between the input and output of a system. In mathematical terms, given a system S that outputs $y[n]$ for input $x[n]$, system S is homogenous if it produces output $k \cdot y[n]$ for input $k \cdot x[n]$ correspondingly.

Additivity

In addition to homogeneity, a system is additive if, for each input signal, the addition of corresponding outputs is the same as the output of a signal that is the addition of the input signals. Figure 8 explains additivity more demonstratively.

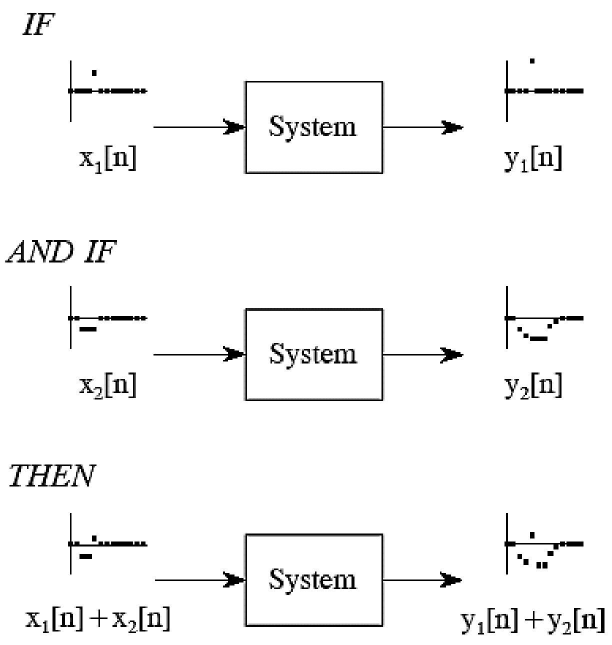

Figure 8. Additivity of a system. Modified from Smith (2020) [1, p. 90].

Time invariance

In most linear systems, it is assumed that the system also holds the property of shift invariance. Much of the nature of a signal processing system is about the relationship between input and output. Shift-invariance means the established relations between these two factors should be maintained for any arbitrary sequence of input and the corresponding output. In the case of discrete-time linear systems, where the sequence of samples is defined at a series of discrete instants of time, this property is also called time invariance. [2, p. 151.]

In more intuitive terms, shift invariance makes sure that any shift in input will result in an equivalent shift in the output. In signal analysis where the independent variable is a time instant, the characteristics of the system are retained regardless of the change in time. Thus, for most DSP techniques involving linear systems, shift invariance is presumably expected as a mandatory property.

3.2.1.2 The importance of linearity in DSP

Most real-life signals are extremely complicated. Thus, designing a practical system must take into account the equivalent amount of computation, and no straightforward approach is viable. However, the common properties of LTI systems, namely additivity, homogeneity and shift invariance, allow systems and signals to be analyzed modularly. The strategy for this analysis is called superposition. The signal is decomposed into simpler additive components, which can be processed independently before the corresponding outputs are synthesized to accomplish the final output. The power this strategy offers makes it the foundation of DSP techniques. With superposition, which springs from the properties of linearity, real-world signals can be understood in the sum of elemental components, and expectedly complex systems can be broken down to sets of modular, simple subsystems. [1, pp. 98-100.]

To facilitate the processing, signal decomposition should replace the complex original input with a set of simpler ones. Though there is a variety of methods for decomposing signals, the major options are impulse and Fourier decomposition. The former decomposes signals into a set of sequences, each containing only one impulse. Thus, it studies how the system reacts to impulses, the contribution of input samples in output samples, and vice versa. It is the basis for convolution calculation. The latter, as its name suggests, decomposes signals to sinusoids according to the Fourier analysis. In this approach, the system is investigated based on how it affects the amplitude and the phase of sinusoids constructing the original input. [1, pp. 100-104.]

3.2.2 System response

The response of a linear system represents its underlying process. Given a unit input, a system will produce a standard output which typifies the process. The most commonly known is the impulse response, which is a system’s response to a unit impulse. The impulse response fully characterizes a system, and designing a system is nothing more than designing its impulse response.

The unit impulse is also known as the delta function (δ), which maps functions to their values at zero. The Dirac delta function can be intuitively understood as an infinitesimally narrow spike at the origin, with the integral of the function amounting to 1, which makes the function (?) an impulse at the origin. In discrete terms, the Kronecker delta function is defined in the discrete domain, and it contains a sole impulse of value 1 at zero. Though the Dirac delta function is slightly too mathematical to be included here, Kronecker’s version can be simplified as seen in Equation 3 below.

$$\delta[n] = \begin{cases} 1, & \mathrm{n} = 0 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Using impulse decomposition, systems process signals by convolving the input signal with their impulse response. It is noteworthy that impulse decomposition and impulse response rely heavily on the concept of unit impulse. The convolution of the unit impulse with any signal will result in a signal which is identical to the output. In addition to this, every impulse can be represented as a scaled and shifted unit impulse. Thus, in a linear system, for any input impulse, which is a scaled and shifted unit impulse, an output impulse can directly be found by shifting and scaling the system’s impulse response in a similar way. [1, p. 107-108.] Figure 9 clarifies how this process behaves. In the figure, the system’s impulse response is a single unit impulse being right-shifted by 15 samples and scaled by factor -0.5. Consequently, the output signal is a sequence of input signals with each sample being inverted, shifted by 15 samples, and halved in amplitude.

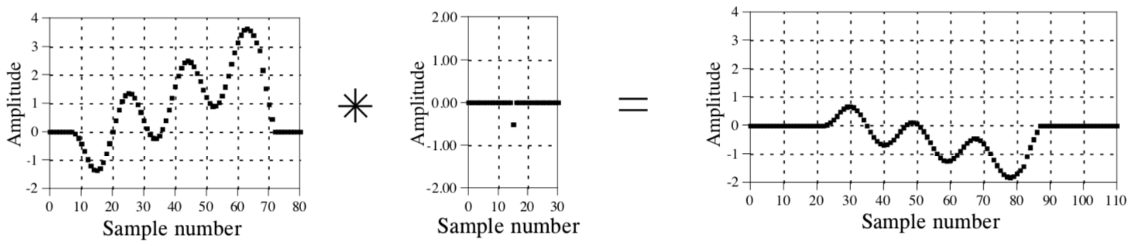

Figure 9. Convolution of input signal and system’s impulse response. Modified from Smith (2020) [1, p. 111].

The frequency response of the system can be obtained by mapping its impulse response to the frequency domain using the Fourier transform. The frequency response typifies how the system responds to the input signal in a function of frequency. It is commonly shown as spectra that measure the changes in amplitude and the phase of the frequency components in the input signal passing through the system. Figure 10 below gives an example of a frequency (magnitude) response from a low-pass filter.

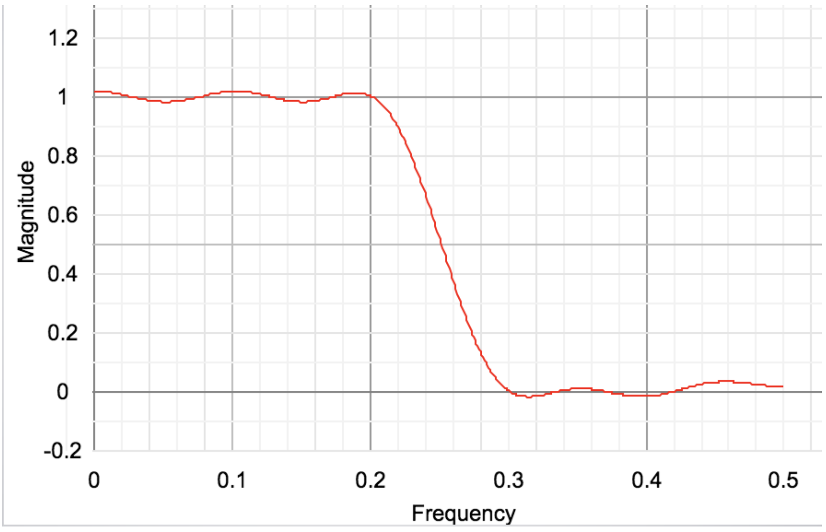

Figure 10. Frequency response of a low-pass filter.

Additionally, a system can also be characterized by its step response. Similar to the impulse response, the step response tells how a system responds to the unit step function and measures the rate of the output’s correspondence in time. This means the step response can be derived from the convolution of the unit step function and impulse response. Despite being seemingly dissimilar, the impulse response, the step response, and the frequency response are just different representations of a system's attributes and can be derived from each other.

3.3 Filter design

Digital filters are an extremely essential section in the field of DSP. They attain this role by demonstrating the extraordinary performance of DSP over analog counterparts and providing solutions for a myriad of problems. Digital filters are DSP systems with mainly two objectives, which are signal restoration and signal separation. This section examines the fundamentals of linear digital filter design, their common categories and techniques. [1, p. 261.]

3.3.1 Filter design and types of filters

As mentioned in the previous chapter, a system is fully characterized by its impulse response. Hence, in essence, implementing a digital filter means convolving the input in the time domain with the impulse response. Each output sample is generated by summing the weighed input samples using the impulse response. [1, p. 262-263.]

FIR filters

FIR stands for Finite Impulse Response. Filters of this type are implemented using the convolution approach, which was mentioned above. Practical FIR filters usually have short, simple impulse responses, and their employments are majorly due to the simplicity they offer. However, in the requirement of more complex filters, FIR implementation is reportedly slow, since convolution is generally considered a computation heavy operation.

IIR filter

This type of a filter solves the issue with long convolution by feeding back calculated output alongside with input samples to bypass a substantial amount of computation.

Making use of recursion, these filters have infinitely long impulse responses, hence the name IIR which stands for Infinite Impulse Response.

Instead of impulse response representation with infinite length, linear IIR filters are usually identified by their transfer function, which is the z-transform of the impulse response in an LTI system. In essence, z-transform maps a discrete time domain signal to the z domain, which will be explained in subsequent sections. Equation 4 below describes the operation of the filter, whereas $Y(z)$, $\mathrm{H}(\mathrm{z})$, $\mathrm{X}(\mathrm{z})$ are the z-transform of the output, impulse response, and input in the time domain. [2, p. 308.]

$$\mathrm{H}(\mathrm{z}) = \frac{Y(z)}{X(z)} = \frac{b_0+b_1z^{-1}+b_2z^{-2}+\cdots}{a_0+a_1z^{-1}+a_2z^{-2}+\cdots} \tag{4}$$

The roots of $Y(z)$ and $\mathrm{X}(\mathrm{z})$ are generally called zeros and poles. The zeros and poles of a transfer function determine the nature and stability of its filter. Designing filters means locating the zeros and poles that determine the desired systems. Since both $Y(z)$ and $\mathrm{X}(\mathrm{z})$ are polynomials in z^{-1}, the finding of zeros and poles is equivalent to identifying sets of coefficients a_k and b_k for those polynomials. [2, p. 308.]

Laplace transform in filter design

The z-transform can be interpreted as the discrete version of the Laplace transform which maps the time domain to the s-domain. Understanding the s-domain is essential to understanding the transfer function and linear filter design. The Laplace transform decomposes a time domain impulse response to sinusoids and exponentials. More specifically, in the s-domain, the impulse response is represented using exponential complex frequencies e^{st} as in equation 5 and 6, whereas ω is the natural frequency and σ is the distance along the real axis. For example, point $s = 3 + 4j$ constitutes a pair of 4 Hz sinusoids with the magnitude being multiplied by e^{-3t}. [1, pp. 581-586.]

$$\mathrm{s} = \sigma + \omega\mathrm{j} \tag{5}$$

$$\mathrm{e}^{\mathrm{st}} = \mathrm{e}^{(\sigma+\omega\mathrm{j})\mathrm{t}} = e^{\sigma\mathrm{t}}e^{\omega\mathrm{jt}} \tag{6}$$

Consequently, zeros are complex frequencies at which the system does not response. On the other hand, poles are complex frequencies at which the system responds at an infinite level. By using poles and zeros as pinpoints, the contour on the s-plane can be shaped. Therefore, the desired frequency response, which is the value of the transfer function along an imaginary axis in the s-domain, is constructed. [1, pp. 597-599.] Figure 11 describes the relationship between the frequency response and the zeros and poles on the s-plane. The red line represents the graph of the frequency response.

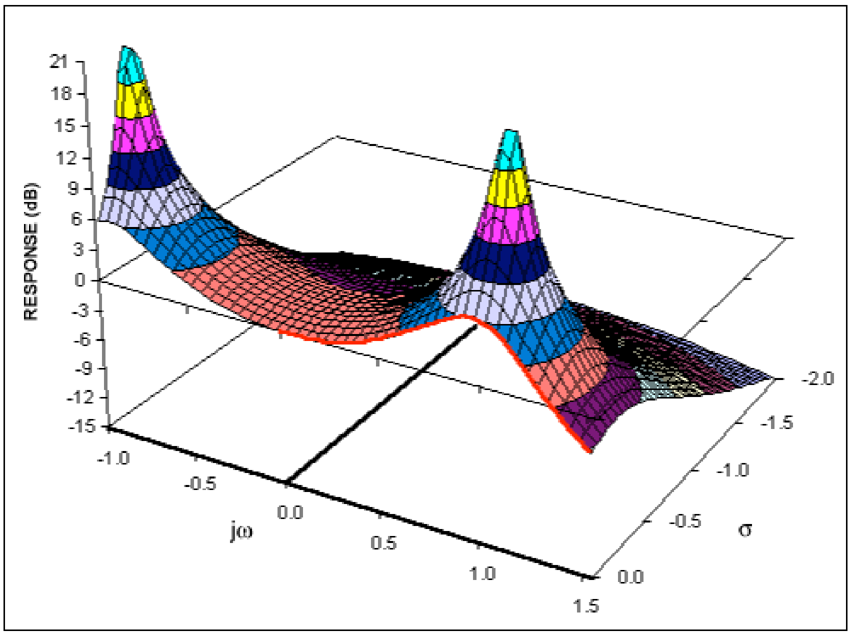

Figure 11. The contour on s-plane of a second-order filter. Copied from [16].

With $\sigma = 0$, there is no exponential. Thus, the Laplace transform is analogous to the Fourier transform in this case. Systems with all poles on the real axis are fundamentally oscillators. With $\sigma < 0$, the sinusoid gradually decays and reaches 0 at $+\infty$. However, with $\sigma > 0$, the filter alters the sinusoids to an increasing extent without bounds, and the system soon becomes saturated. Thus, to achieve a stable system, it is a necessity that the poles should be on the left-hand side of the s-plane.

3.3.2 Overview of common filter design techniques

Generally, the Laplace transform opens the possibility to directly design filters on the s-domain by pinpointing the number and locations of zeros and poles. On the z-plane, a stable system needs the poles to be within the unit circle, which encapsulates the left side of the s-domain. Figure 12 provides an example of a pole-zero plot on the z-plane and its corresponding frequency amplitude response.

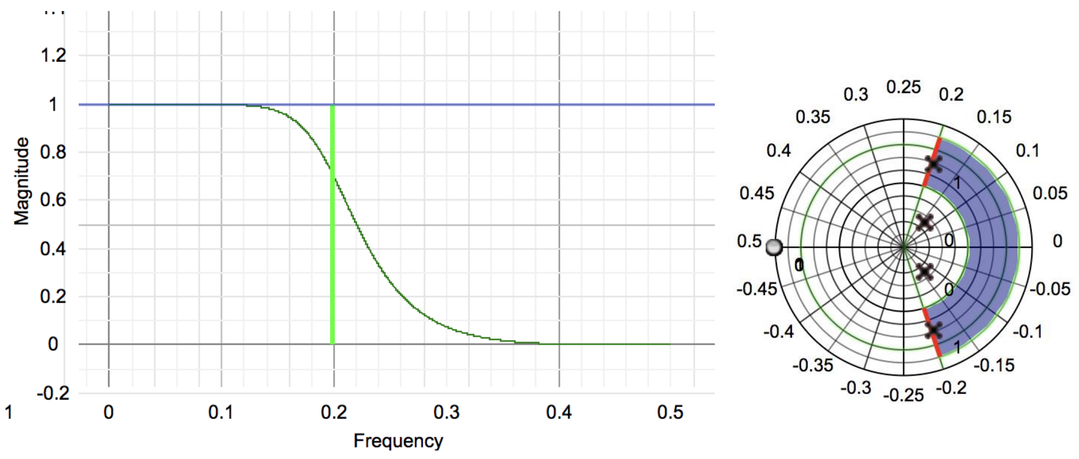

Figure 12. The frequency amplitude response and z-plane of a low-pass filter with a proportional cutoff frequency of 0.2 the sampling rate.

It is common that complex-conjugate pole pairs are used to achieve a flat pass-band while maintaining sharp roll-off. In systems with more than two poles and zeros, the usual solution is to implement the filter in successive stages of up to two poles and two zeros each. The filters implemented by this approach are often called biquad filters, due to the biquadratic numerator and denominator in the transfer function.

Common filter types

Using a mathematical approach, there is a variety of strategies for placing zeros and poles in the s-domain to produce filters with specific trademarks. For instance, these filter types include Butterworth, Chebyshev, and elliptic filters. Each of them prioritizes a certain aspect of the filter response, since there is no optimal filter on all grounds.

Butterworth filters are known to have a maximally flat passband, meaning the transition between passband and stopband is optimally sharp without peaking. The poles in Butterworth filters are evenly placed on the same circle in the s-domain. To achieve sharper transition, Chebyshev filters have poles positioned on an ellipse; hence, ripples may exist in the passband or the stopband. Elliptic filters push this further by allowing ripples in both the stopband and the passband to attain an even steeper transition band. The name of the elliptic filter is derived from the filters' poles being arranged based on elliptic functions and integrals. [1, pp. 600-603.] Figure 13 illustrates the distinction between these three filter types more visibly.

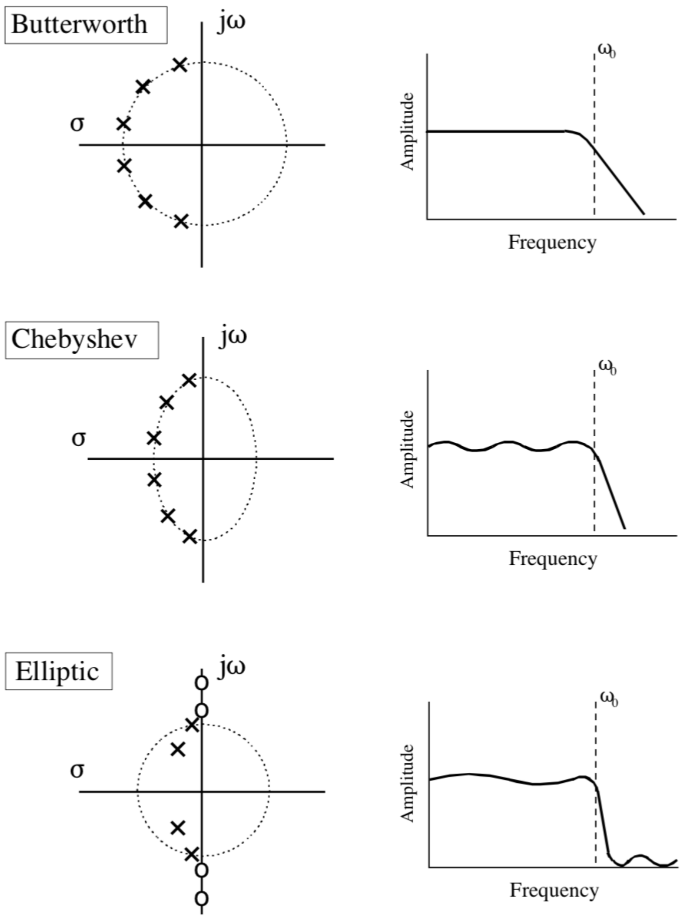

Figure 13. Example of pole-zero maps and frequency responses of the Butterworth, Chebyshev and elliptic filters. Modified from Smith (2020) [1, p. 603].

Spectral inversion and spectral reversal

Filter design generally starts with a low-pass filter. The filter response for a high-pass filter, if needed, can then be obtained by translating a proper low-pass filter response. Band-pass and band-reject filters can be effortlessly constructed by combining appropriate high-band and low-pass filters. For all this to work, converting between low-pass and high-pass is indispensable. Spectral inversion and spectral reversal are the common methods to do such conversion.

Spectral inversion, in short, means changing the signs of each sample in the impulse response, then adding 1 at the center sample. Intuitively, this is analogous to excluding the output of the initial filter from an all-pass filter. Consequently, using spectral inversion, a low-pass filter response is converted to its inverse. Nonetheless, this technique requires the filter kernel to have left-right symmetry. [1, pp. 271-273.]

Spectral reversal, on the other hand, reverses the frequency spectra by changing the sign of every other sample in the initial impulse response. This is similar to multiplying the filter kernel with a sinusoid with the frequency of 0.5 the sampling rate. According to the shift theorem of the Fourier transform, this shifts the frequency domain by 0.5, and reverses the frequency spectra left-for-right. [1, pp. 273-274.] Figure 14 clarifies how spectral reversal behaves.

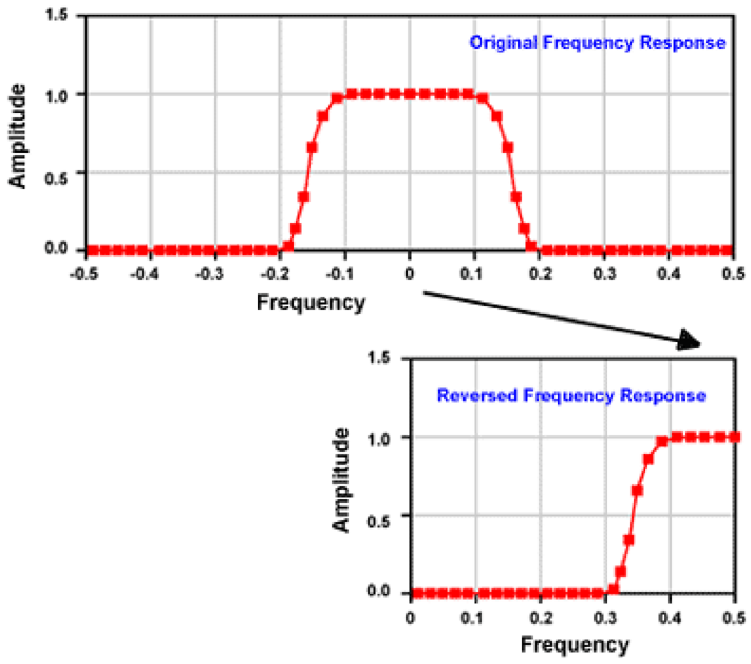

Figure 14. How frequency response is changed by spectral reversal. Copied from Quinnell [17].

The discussion regarding spectral inversion and spectral reversal mostly involves FIR filters. Such methods for low-pass to high-band conversion can also be applied to recursive filters, although not as effectively.

4 Project methods and materials

4.1 Case summary

The previous section discussed the most important aspects of the field of DSP. Thus, a proof of concept is provided in this chapter to complement the theoretical framework. The project documented here is based on VCV Rack, an open-source virtual modular synthesizer that largely replicates the original analog hardware. The project showcases the practical application of DSP by dealing with the foremost direct form of signal in everyday life – audio.

By the end of the project, a working set of modules for processing audio was completed as the result. The modules are largely based on principal analog archetypes, but the possibility for further extension and module development are certain. Also, the study investigates some insights about the VCV Rack engine, on which the modules operate, and provides a simple similar emulation as proof of concept.

4.2 Modular synthesis

Modular synthesizers are electronic musical instruments that integrate a group of individual components called modules to create electronic sounds. Modules can be connected with each other via input-output ports offered by the modules themselves through cables. This is how the audio signal is transferred from one module to another. Additionally, cable patching enables inter-module communication using voltage, which allows a module to adjust the parameters of another. This concept is called voltage control. The unique set of modules and cable connections that is combined to create a specific sound is called a patch.

Modules for modular synthesis can be separated into several categories. The first type is the audio sources, to provide the patch signals to work with. The most common modules in this type are oscillators, which generate waveforms, or noise generators. The second block of modular synthesis is audio modifiers, for example, filters and waveshapers. Finally, there are modules that work as modulation sources. These modules are mainly used to generate signals that regulate other modules. They are typical low frequency generators (LFO), enveloper generators, or random sources. Modulation is a distinctive element that adds singularity to the sound.

The reasons for choosing modular synthesis are twofold. The modularity of this type of a synthesizer allows the building of separate modules, one small unit at a time. This resembles designing and building systems and sub-systems in DSP applications. Another reason is the endless number of patching permutations that modular synthesis offers. Especially in a virtual environment, where one module can provide multiple instances, this will show how achievable it is with only a small set of basic components.

In this project, it was decided to build a minimal set of modules sufficient for making patches, which includes at least one module for each category described above. Also, separate modules for signal-to-audio conversion and MIDI were needed. However, due to time limitation, the project made use of open-source module packages to fill in that gap.

4.3 Technical information of the project

This section gives more details about the technical aspect of the project. It covers the technology and tools being used, and the structure of the project. Also, the VCV Rack engine for processing signals is examined, in order to have a better understanding of the system. Finally, module development and steps that need to be taken and in module development are discussed.

4.3.1 Programs and tools

The project aims to create a set of modules as a plugin for the VCV Rack system. A manual which is available on the official website of VCV Rack was used in plugin development and graphical user interface (GUI) design [18]. To finish the implementation, the following tools and programs have been used.

- C/C++ (for application and plugin source code)

- Python (to run provided setup scripts)
- Xcode and VS Code (as text editor)
- MacBook (as a running platform for VCV Rack and the plugins)
- Inkscape (a graphic editor to design the GUI and panels of the plugins)
- Octave (for plotting sampled signals from the engine simulation)
- Other public resources: open-source plugins and online tools

4.3.2 VCV Rack engine

Common audio software programs have a sampling system to process signals in buffers. On the other hand, VCV Rack handles one single sample at a time. The engine needs to allow an extensive number of modules and patching; thus, the latency that block-based processing introduces will become additive with more modules added to the rack. However, since the application involves real-time interaction, this can potentially cause overhead from the call chain. VCV Rack takes advantage of modern processors and the large amount of internal and fast cache they provide to make the sampled-based approach come to terms. In addition to this, the operation for handling processed signals in the engine was initially single-threaded, but it has recently been re-implemented using a multi-threading approach, hence supporting a larger number of modules in one patching.

Fundamentally, the VCV internal engine keeps a vector of modules added to the rack, and a vector of cables connected module-to-module. Other general setting parameters, including sampling rate and sampling time, are also handled internally and passed to each module during processing. A module basically needs to have outputs and/or inputs with a process function that computes the processed signal. It is noteworthy that it is flexible for modules to process per sample, or per buffer. Iterating the vectors of modules and cables, the engine repeatedly invokes the process function in each module and moves the output of one module to another module’s input, based on connected cables. The iteration takes place in the time frame for one single sample, which is calculated based on the given sampling rate.

The application has one main thread which handles the GUI call chain, a separated thread for audio hardware I/O, and a pool of worker threads to iterate through the modules and cables in the current rack. The engine workers simultaneously race for modules added to the rack and invoke their processing function until the module list is exhausted. A barrier is used for synchronizing the engine workers, assuring that all worker threads finish before finalizing synchronous execution, including cable iterating and parameter smoothing, and repeating the same procedure for the next sample. [19.]

4.3.3 Modules development

All of the modules developed for this project are elemental building blocks of modular synthesizers. The detailed steps for developing a module is included in the VCV official manual [18]. To summarize, the manual begins with instructions for designing the panel for the module using Inkscape. Then, based on the panel SVG file, a template for module source code is generated using the Python scripts provided with the Rack SDK. The template should include the expected inputs and outputs specified on the panel design [20]. It is possible to write the source code and define the inputs and outputs of the module from scratch, but it is easier to use the template. The actual computation for processing the signal is then implemented.

It was unexpectedly troublesome to test the working of the modules. There was little time, and no systematic way was used for testing. Mostly the implementation was tested manually by comparing it to similar published plugins and by plotting the time domain signals using another module named Scope which is part of the Fundamental package provided by VCV. By inspecting the waveform of the output signal, it was possible, although not optimal, to dissect the errors encountered.

5 Implementation

In this section, the report will go into details about the implementation of the project. Simulating the VCV engine is also described. The objective is to illustrate the essential working of the application with a simple proof of concept, instead of directly dealing with the complex source code of VCV. It is important to note that multi-threaded programming is not discussed, due to the needless complexity it demands. Subsequently, the development of each module is discussed.

5.1 Processing engine simulation

Firstly, a model for the engine is needed. The following code (see Listing 1) shows the properties and methods that the engine encapsulates in a C++ data structure. The data structure consists of two vectors to keep track of the modules and cables added, methods for adding them, sampling rate and sampling time of the system, and a method run to trigger the whole process.

```
struct Engine {
      std::vector<Module*> modules;
      std::vector<Cable*> cables;
      float sampleRate;
      float sampleTime;
      Engine();
      void addModule(Module* module);
      void addCable(Cable* cable);
      void run();
}
```

Listing 1. Model for VCV engine.

It is important to note that a module has three main properties: inputs, outputs, and parameters. All is stored in corresponding vectors. The modules also need an inheritable method to compute actual output samples. Engine settings, namely sampling rate and sampling time, are passed to this method. Specific modules are implemented by extending this base model. Listing 2 demonstrates the base model and necessary properties for all module.

```
struct ProcessArgs {
      float sampleRate;
      float sampleTime;
}
```

```
struct Module {
      std::vector<float> inputs;
      std::vector<float> outputs;
      std::vector<float> params;
      virtual void process(ProcessArgs args);
}
```

Listing 2. Base model for a module.

Subsequently, the model for the cable is also needed. A cable object should encapsulate pointers to the modules at the two ends of the cable and a method for copying the signal value from one end to the other. It is noteworthy that since a module can have multiple outputs and inputs, the cable also needs to keep the ID of the inputs and outputs it is connecting. The model for the cable that connects different modules is provided in Listing 3.

```
struct Cable {
      Module* inputModule;
      Module* outputModule;
      int inputId;
      int outputId;
      // copy output from inputModule to input in outputModule
      void step() {
            outputModule->inputs[outputId] = inputModule->outputs[inputId];
      };
}
```

Listing 3. Model for connecting a cable.

Since the engine processes one single sample at a time, the timeframe for one iteration is depended on the chosen sampling rate. In one iteration, the engine goes through all modules, invoking their processing function, then stepping through all the connected cables. As can be seen in Listing 4, an engine is initialized in the C++ main function, and the standard sampling rate for audio, 44100 Hz, is used.

```
const float SAMPLE_RATE = 44100.f;
const float SAMPLE_TIME = 1.f / SAMPLE_RATE;

int main(int argc, const char* argv[]) {
      Engine* engine = new Engine();
      engine->sampleRate = SAMPLE_RATE;
      engine->sampleTime = SAMPLE_TIME;
      …
}
```

Listing 4. Initializing the engine.

Since this is only a proof of concept, the modules have to be added to the rack manually by hardcoding for the engine to process. In this example, two example modules are added, including a VCO which generates a sine wave, and a VCA which reduces the signal amplitude by half. The output of the VCO is connected to the input of the VCA, and a cable is also manually added for that. Listing 5 illustrates the setup.

```
VCO* vco = new VCO();
VCA* vca = new VCA();
Cable* cable = new Cable();
cable->inputId = VCO::SINE;
cable->outputId = VCA::IN_1;
cable->inputModule = vco;
cable->outputModule = vca;

engine->addModule(vco);
engine->addModule(vca);
engine->addCable(cable);
```

Listing 5. Adding a sine wave VCO, followed by a VCA that halves the signal in the rack.

The described setup is the simplest patch for modular synthesis. The output for this patch comes from the output port of the VCA. The main loop of the engine's processing requires it to iterate through all the added modules and cables in the timeframe of the sample time, which is $\frac{1}{44100}$ seconds in this case. Most likely, the actual computation will take less time than the sample time; hence, it is needed to make the engine wait, until it can process the next sample in the next timeframe. Listing 6 explains the logic. The code is followed by a detailed explanation.

```
ProcessArgs args;
args.sampleRate = engine->sampleRate;
args.sampleTime = engine->sampleTime;

auto curTime = std::chrono::high_resolution_clock::now();
auto lastTime = curTime;
double overheadTime;
float output;

while (true) {
      for (Cable* c : engine->cables) {
            c->step();
      }
      for (Module* m : engine->modules) {
            m->process(args);
      }

      output = vca->outputs[VCA::OUT_1];

      curTime = std::chrono::high_resolution_clock::now();
      overheadTime = std::chrono::duration<double>(curTime - lastTime).count();
      lastTime = curTime;
```

```
        std::this_thread::sleep_for(std::chrono::duration<double>(SAMPLE_TIME -
overheadTime));
}
```

Listing 6. The main loop for the iteration of the engine.

In this processing loop, the iteration for cables and modules can be seen from line 11 to line 16. Subsequently, how the output of the patch is saved can be seen on line 18. There are two variables, curTime and lastTime, in order to keep track of the time from the system internal clock before and after the computation. The overhead time is calculated from the difference of the timestamps before and after the computation. Lines 20-22 illustrate how this is executed. The main thread of the application in which the engine is executed then waits until the sample time is over to repeat the main loop. This is handled by the code in line 23, the end of the engine's iteration.

The resulted output is an 8 Hz sine wave with magnitude 0.5 as in Figure 15. If necessary, the output can be written to a WAV file to play the actual audio although the sine wave should be of a higher frequency to really be audible.

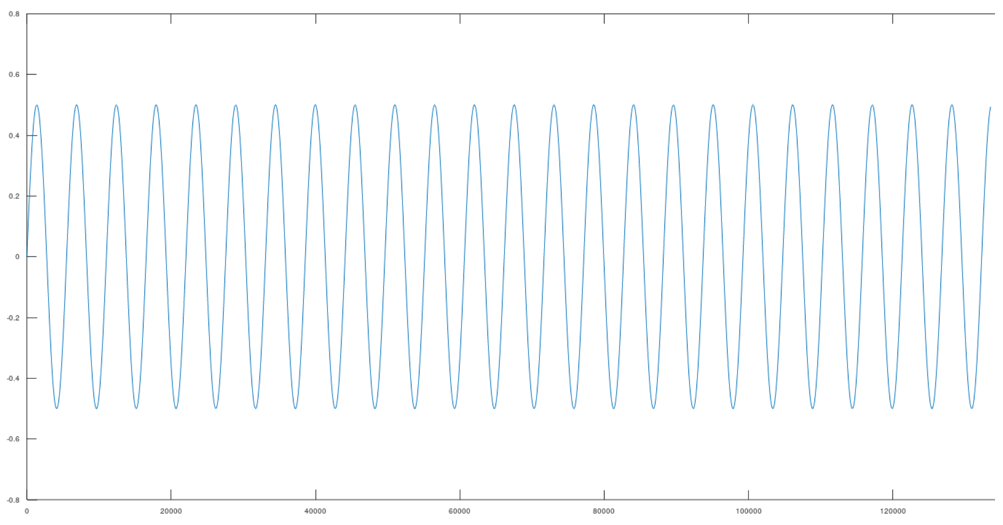

Figure 15. The plot of output signal from example patch.

In this example, the engine processes one sample at a time, triggering a VCO module to generate the waveform, before passing it to the VCA to alter the amplitude. There is a much greater amount of complexities in the open-source VCV engine; however, this proof of concept should suffice to explain the basic workings of the VCV engine and simple patches.

5.2 VCO

This VCO module is responsible for generating waveforms. The waveforms can act as the building material in a patch, or as modulating agents. Figure 16 shows the panel of the module.

Outputs, inputs, and parameters

The module has four outputs for four common waveforms, including sine, triangle, saw, and square waves. The FREQ knob controls the base frequency of output signal. The module has four inputs, namely V/OCT (pitch control), FM (frequency modulation), sync, and PWM (pulse width modulation) which is only available for square wave.

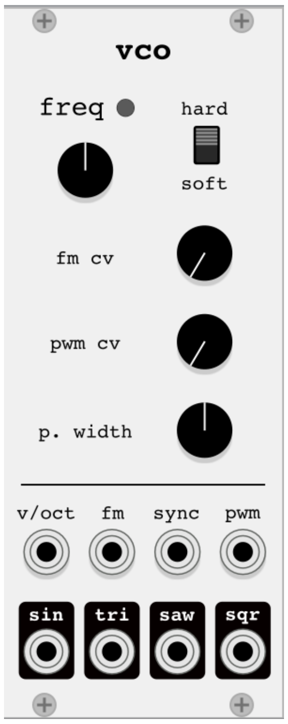

Figure 16. The panel and settings of the VCO module.

The base frequency can be set in the range of pitch A0 to A8, equivalent to 27.5 Hz – 7040 Hz, using the FREQ knob. The vertically aligned knobs control the level of corresponding modulation, ranging from 0 to 100 percent. Then there is a switch to change the synchronization between hard and soft mode.

Pulse-width modulation

Pulse width modulation is a method for controlling the average level of power in the signal via its on-off state. The pulse-width parameter should be between 0 and 100 percent. Theoretically, at 100 percent, the signal becomes a DC value. Figure 17 below illustrates the waveform of a square wave with customized pulse width. The pulse-width setting can be altered autonomously by a modulating source as input.

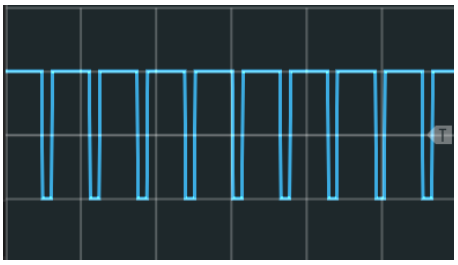

Figure 17. A square wave with pulse width set to 80%.

Oscillator synchronization

Oscillator synchronization, or oscillator sync, allows the period of the output waveform to be controlled based on the period of another waveform. The result of this is having the output waveform at the same base frequency as the modulating input, or also called the master. This leads to an irregular cycle in output waveform, or so called the slave, and creates harmonically rich sounds, with complex timbres.

Oscillator synchronization can have two modes: hard and soft sync. Hard sync means the slave wave’s period is always reset based on the master. On the other hand, soft sync does not necessarily reset the period of the slave wave, but in this implementation, it reverses the slave’s phase at the moment the sync is triggered. Figure 18 shows an example of how soft and hard sync affect the waveform.

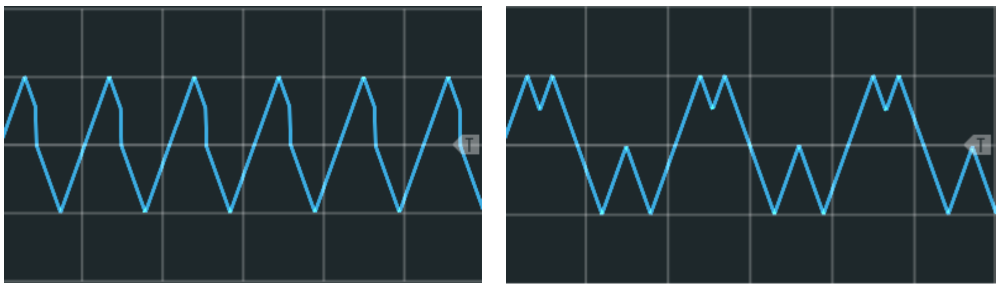

Figure 18. From left to right. A 107 Hz triangle waveform synced by a 123 Hz sine wave in hard and soft mode.

Implementation

First, the template for the module source code was generated based on the guidelines [18]. A struct named VCO was extended from the generic data structure module from VCV Rack SDK. IDs needed to be set for the module's inputs, outputs, and parameter. The widget for the GUI of the module should not be a concern at this point. Mostly it is the process function that is of major importance. The initial template for this module is as followed.

```
Struct VCO : Module {
	enum ParamIds {
		FREQ_PARAM, FM_PARAM, PWM_PARAM,
		PWIDTH_PARAM, SYNC_PARAM, NUM_PARAMS
	};
	enum InputIds {
		PITCH_INPUT,
		FM_INPUT,
		SYNC_INPUT,
		PWM_INPUT,
		NUM_INPUTS
	};
	enum OutputIds {
		SIN_OUTPUT, TRI_OUTPUT, SAW_OUTPUT,
		SQR_OUTPUT, NUM_OUTPUTS
	};

	VCO() {
		// configure the range of parameter and initial parameter value here
	}

	void process(const ProcessArgs& args) override {
		// computation for output signal here
	}
}
```

Listing 7. VCO module data structure.

There are several instance variables that the module needs to keep track of. These are the base frequency and phase of the output, sync direction, and the last value of sync input (if there is any). The variables are initialized as followed. Default frequency is set to A4 pitch, and phase is set to zero. Sync direction controls hard and soft sync, as for soft sync it will help reversing the phase. The hard sync direction value is 1, and the soft sync direction value is -1.

```
float baseFreq = dsp::FREQ_A4;
float phase = 0.f;
float lastSyncInput = 0.f;
float syncDirection = 1.f;
```

Listing 8. VCO module’s instance variables.

Now with the inputs and parameters, the process method will compute the output signal for the module. First, the frequency of the output waveform should be determined. In the following, the output frequency is calculated based on the pitch parameter, and two possible modulating sources which are V/OCT and FM.

```
float pitch = freqParam;
pitch += inputs[PITCH_INPUT].getVoltage();
pitch += fmParam * inputs[FM_INPUT].getVoltage();
pitch = clamp(pitch, -4.f, 4.f);
float freq = baseFreq * std::pow(2.f, pitch);
```

Listing 9. Calculation for output waveform frequency.

The output frequency is the sum of the base frequency set in the module and the additional variation from other modulating sources. Frequency modulation changes one frequency in line with the amplitude of the other. The level modulation is controlled by multiplication with modulation parameters. The output frequency is limited to eight octaves from A0 to A8, so the calculated frequency needs to be safely clamped in this range. Line 4 and 5 illustrate how this was done.

The following code calculates the phase, and updates sync direction for the next iteration of the process method. The phase is finitely wrapped in the range [-0.5, 0.5] to avoid overloading memory, and resolved in line 1 and 2. Whether the phase decreases or increases is decided by the sync direction, which is updated in the rest of the code.

```
phase += freq * args.sampleTime * syncDirection;
phase -= phase >= 0.5f || phase <= -0.5f ? (1.f * syncDirection) : 0.f;
```

```
bool isHardSync = params[SYNC_PARAM].getValue() > 0;
if (inputs[SYNC_INPUT].isConnected()) {
	float syncInput = inputs[SYNC_INPUT].getVoltage();
	if (lastSyncInput <= 0 && syncInput >= 0) {
		if (isHardSync) {
			phase = 0.f;
		} else {
			syncDirection *= -1.f;
		}
	}
	lastSyncInput = syncInput;
} else {
	syncDirection = 1.f;
}
```

Listing 10. Calculation for phase and sync.

With the phase and frequency calculated, generating the actual waveforms is quite straight forward. The code below includes functions for returning the sample in sine, triangle, saw, and square waves. For non-sinusoidal waveforms, the waveforms are naturally created by composing the Fourier series. To flawlessly create sharp edges in triangle, saw and square waves, theoretically, an infinite series of harmonics is needed. However, this is not computationally possible; thus, the analog waveform tends to have ripples and jump discontinuity due to the Gibbs effect. However, in this implementation, the waveform is generated sample-by-sample, which allows the waveform to be mathematically perfect.

```
float sin(float phase) {
	return std::sin(2.f * M_PI * phase);
}
float tri(float phase) {
	bool posDerivative = -0.25f < phase && phase < 0.25f;
	float x = posDerivative ? phase : (phase < 0 ? (phase + 0.5f) : (phase - 
0.5f));
	return (posDerivative ? 4.f : -4.f) * x;
}
float saw(float phase) {
	return 2.f * phase;
}
float sqr(float phase, float pulseWidth) {
	return phase > (pulseWidth * 1.f - 0.5f) ? -1.f : 1.f;
}
```

Listing 11. Functions for returning sample in output waveforms.

In addition to the square wave generator, the pulse-width level needs to be calculated and passed to the square wave function. It is noteworthy that in the following code, pulse-width level is clamped to the range [0.01, 0.99] to avoid both absolute ends.

```
pulseWidth += pwmParam * pwmInput;
pulseWidth = clamp(pulseWidth, 0.01f, 0.99f);
```

Listing 12. Calculation for pulse-width level.

After having the waveform samples, the values of the samples are assigned to the corresponding outputs of the module. This should finalize the implementation of this VCO, and the module should be fully workable.

5.3 VCA

VCA, or voltage-controlled amplifier/attenuator, is probably the simplest module being discussed in this project. The module basically alters the amplitude of a receiving signal directly, or via amplitude modulation. However, due to the audio clipping level of the internal system, mostly attenuation was done, instead of amplification. The following section describe the module more in detail.

Outputs, inputs, and parameters

The module has only one output and one input for the receiving signal and its result with a modified amplitude. There is one knob LEVEL to control the only parameter of the module, which is the level of amplitude gain. The level of this VCA should be between 0 and 100 percent, which makes it an attenuator. There are also two optional inputs, LINEAR and EXP. They allow the receiving signal to have its amplitude altered linearly, or exponentially, by other modulating sources. Figure 19 provides the designed panel for the module.

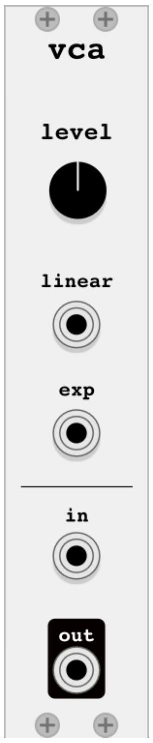

Figure 19. The panel and settings of VCA module.

Implementation

Similar to the module VCO discussed in the previous section, the template for this module's source code is generated as instructed in the guidelines [18]. After having the data structure for the module, configuration for its specific inputs, outputs, and parameters can be made. This is briefly shown in Listing 13.

```
enum ParamIds {
      GAIN_PARAM, NUM_PARAMS
};
enum InputIds {
      LINEAR_INPUT, EXP_INPUT,
      MAIN_INPUT, NUM_INPUTS
};
enum OutputIds {
      MAIN_OUTPUT, NUM_OUTPUTS
};

VCA() {
      config(NUM_PARAMS, NUM_INPUTS, NUM_OUTPUTS, NUM_LIGHTS);
      configParam(GAIN_PARAM, 0.f, 1.f, 0.5f, "Gain", "%", 0.f, 100.f);
}
```

Listing 13. VCA module's initial source code.

Then comes the main work for implementing these modules, the process method. The VCA module, unlike the discussed VCO, only needs to concern about computing a new amplitude for the sample linearly, and exponentially, then set the value to the output.

Changing the amplitude using a linear function is simple, as shown in the following code. It is important to note that the division by ten is due to the voltage standard of the system, as signals should typically have 10 V peak-to-peak value. Dividing them by ten gives the factor in range of 0 to 1, by which the modulating source should affect the output signal.

```
if (inputs[LINEAR_INPUT].isConnected()) {
      output *= inputs[LINEAR_INPUT].getVoltage() / 10.f;
}
```

Listing 14. Computation for linear modulation.

Changing the receiving sample’s amplitude exponentially is more formulaic. The base for exponential function is chosen to be 50, and since this is an attenuator, it is needed that the overall factor for amplitude modification must be equal or less than 1. This computation is handled by the code provided in Listing 15.

```
const float EXP_BASE = 50.f;
// …
float normalizeFactor = 1.f / (std::pow(EXP_BASE, 1) - 1.f);
if (inputs[EXP_INPUT].isConnected()) {
      float expInput = inputs[EXP_INPUT].getVoltage() / 10.f;
      output *= normalizeFactor * std::pow(EXP_BASE, expInput);
}
```

Listing 15. Computation for exponential modulation.

To summarize, the multiplying factor, temporarily called A, can be determined using the following formula. It is important to note that b is the base for the exponent, which is initialized to 50, and x is the modulating factor from the input source. With x in [0, 1], A is assured to stay within [0, 1] too.

$$A = \frac{b^x}{b^1 - 1} \tag{7}$$

After handling the modulation, the gain, which is set by the module’s parameter, can be directly applied to the signal and assigned to the output. The code for this part is provided in Listing 16. This should complete the implementation of the VCA module.

```
float gainLevel = params[GAIN_PARAM].getValue();
outputs[MAIN_OUTPUT].setVoltage(gainLevel * output);
```

Listing 16. Applying gain for signal output.

As can be seen in Figure 20, the VCA module modulates a 220 Hz sine wave input using a 0.3 Hz sine wave. The red line plots the original signal, while the blue one plots the module's output. Note that the output is changing overtime as it is being modulated by another waveform.

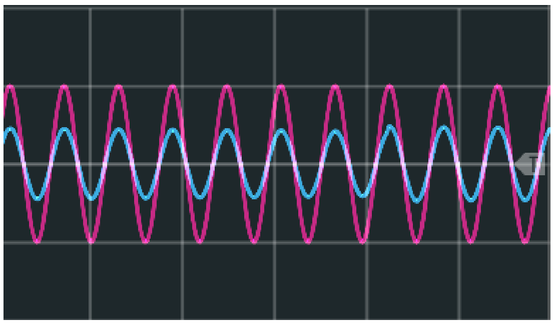

Figure 20. Example of the working of the VCA module.

5.4 Envelope

An envelope, or envelope generator, is a common module being used as a modulation source. The envelope can be used to shape individual notes, change the loudness of audio or its harmonic mixture to give them animation and articulation. Generally, the envelope generator creates a voltage that starts at 0, when it is triggered by a gate signal, and then develops in the following stages.

- Attack: voltage rises to maximum level.
- Hold: voltage stays at maximum level.
- Decay: voltage decays to a stable level.
- Sustain: voltage is sustained at a set level as long as the gate signal is high.
- Release: voltage dies out when the gate signal is low.

Figure 21 illustrates these stages mentioned above and how the gate signal affects each stage. In this implementation, the module being developed consists of all these five stages although it is usual that common modules skip the hold stage. Additionally, it is noteworthy that the attack, decay, and release states are usually determined using an exponential function. Corresponding to these five stages, there are five optional modulations for each source.

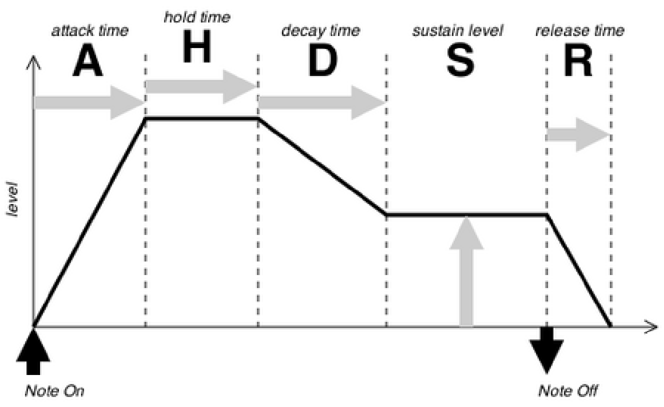

Figure 21. The stages of an envelope controlled by a triggering signal. Copied from [21].

Window function

In essence, an envelope is a window function. In signal processing, a window function is a mathematical function that zero-values samples outside of an interval while applying a weighting to discrete time samples inside. Figuratively, it is comparable to viewing the signal via a window, hence the name.

In theory, window functions are usually used to solve the problem of spectral leakage. However, in music application, the window function’s nature is used in the envelope to give articulation for a specific note and harmonic uniqueness in sound mix.

Outputs, inputs, and parameters

The panel for the envelope module can be seen in Figure 22. The module has only one output for the actual generated envelope. In addition to this, there are also two inputs for the gate signals GATE and RETRIG for the retrigger signal that resets the generator. The module has five knobs in vertical alignment to control five parameters regarding the five stages of the envelope. Accordingly, there are five additional inputs to provide modulation for the mentioned parameters.

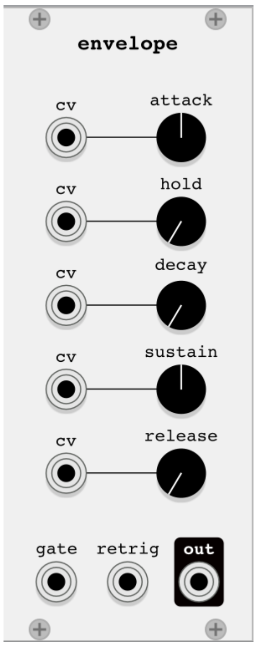

Figure 22. The panel and settings of the envelope generator module.

Implementation

Similar to the development of the discussed modules, the template and initial source code for the envelope module is generated according to the manual [18]. Then the configuration for the parameters of the module can be done, and the development can focus on the process method. It is important to note that the sustain stage will always be maintained if the gate is still on. The parameter for the sustain level is measured by percentage, while attack, decay, hold, and release are stages determined by time interval, in milliseconds.

Since the envelope needs to keep track of the time duration for its stages, a data structure for the envelope clock is included, along with several instance variables for the module's internal logic. The envelope clock accumulates the sample time and is reset in the next period, or when notified by a retrigger signal. The declaration of these properties is illustrated in Listing 17.

```
struct EnvelopeClock {
      float time = 0.f;
      void step(float timeStep) {
            time += timeStep;
      }
      float getClockTime() {
            return time;
      }
      void reset() {
            time = 0.f;
      }
};
… // other template code

// instance variables
dsp::SchmittTrigger trigger;
EnvelopeClock clock;
float lastGateInput = 0.f;
float gateStartValue = 0.f;
float gateEndValue = 0.f;
float openGateDuration = 0.f;
float lastValue = 0.f;
```

Listing 17. Envelope clock and instance variables.

The range for the interval of four stages is as follows: attack, decay, and release are from 1 ms to 10,000 ms. However, the hold stage is optional; thus, its duration can be 0. The values of the parameters are normalized to be in range [0, 1]; therefore, to retrieve the actual duration of the stages, a power function is needed. First, the minimum and maximum time duration, and the base for the power function are declared as follows. Then the base time durations set by the parameters are calculated. It is important to note that the parameter values need to be clamped to stay within the [0, 1] range. These calculations are presented in the code in Listing 18.

```
const float MIN_DURATION = 1e-3f;
const float MAX_DURATION = 10.f;
const float PARAM_BASE = MAX_DURATION / MIN_DURATION // = 10000.f;

float attack = attackParam + inputs[ATTACK_INPUT].getVoltage() / 10.f;
float hold = holdParam + inputs[HOLD_INPUT].getVoltage() / 10.f;
float decay = decayParam + inputs[DECAY_INPUT].getVoltage() / 10.f;
float release = releaseParam + inputs[RELEASE_INPUT].getVoltage() / 10.f;

attack = clamp(attack, 0.f, 1.f);
hold = clamp(hold, 0s.f, 1.f);
decay = clamp(decay, 0.f, 1.f);
release = clamp(release, 0.f, 1.f);

float attackTime = std::pow(PARAM_BASE, attackParam);
float holdTime = std::pow(PARAM_BASE, holdParam) - MIN_DURATION * 1000.f;
float decayTime = std::pow(PARAM_BASE, decayParam);
float releaseTime = std::pow(PARAM_BASE, releaseParam);
```

Listing 18. Computing base time duration for the attack, hole, decay, and release of the envelope set by the parameter.

Next, the ON/OFF state of the gate and retrigger signal need to be identified. Due to the Gibbs effect, the trigger source can falsely trigger the module if the source is bandlimited. Thus, the low and high thresholds for triggering are set to 0.1 V and 1-2 V, as suggested in the VCV Rack documentation [22]. The code for identifying the ON/OFF state of the gate and retrigger signal is shown below.

```
bool isTriggered = trigger.process(rescale(inputs[RETRIGGER_INPUT].getVolt-
age(), 0.1f, 2.f, 0.f, 1.f));
float gateInput = inputs[GATE_INPUT].getVoltage();
bool gateIsOpened = gateInput > 1.f;
```

Listing 19. Identifying the state of the gate and retrigger signal.

Subsequently, the retrigger action needs to be handled. When the module is retriggered by another signal, it generates the envelope stage-by-stage again, with the last value as its initial voltage. The gate signal must be in ON state for this to happen. When the envelope is retriggered, the envelope clock and the opening time duration of the gate are reset to 0. Only then can the sample time be accumulated to the clock. The following code demonstrates this situation.

```
if ((lastGateInput <= 1.f && gateInput > 1.f) || (isTriggered &&
gateIsOpened)) {
      gateStartValue = lastValue;
      openGateDuration = 0.f;
      clock.reset();
}
clock.step(args.sampleTime * 1000.f);
```

Listing 20. The envelope reset when it is retriggered by another signal.

Since the attack, decay, and release stages are determined by an exponential function, it is crucial to vary the exponential function based on the time duration of such a state. The exponential base is retrieved from a separate method, using the parameter value. Formula 8 is customized to dictate the appropriate base for exponential functions of attack, decay, and release. In the formula, b is the calculated base, whereas x is the value of the module’s parameter, varying from 0 to 1.

$$b = 300 \cdot 100^{-x} \tag{8}$$

Now, the actual calculation for voltage in each stage of the envelope can be tackled. The code in Listing 21 below procedurally computes the voltage in each stage using their time duration and exponential base.

```
float time = clock.getClockTime();
float value;

if (gateIsOpened) {
      if (time <= attackTime) {
            float normalizeCoef = 1.f - (1.f - gateStartValue) * std::pow(at-
tackBase, -1.f);
            value = (1.f - (1.f - gateStartValue) * std::pow(attackBase, -time /
attackTime)) / normalizeCoef;
      } else if (time <= attackTime + holdTime) {
            // holding maximum value in hold stage, do nothing
      } else {
            // do decay (approaching sustain level)
            float decayDuration = time - holdTime - attackTime;
            value = (1.f - sustain) * std::pow(decayBase, -decayDuration / de-
cayTime) + sustain;
      }
      openGateDuration += args.sampleTime * 1000.f;
      gateEndValue = value;
} else {
      // do release (approaching 0)
      float releaseDuration = time - openGateDuration;
      value = gateEndValue * std::pow(releaseBase, -releaseDuration / re-
leaseTime);
      gateStartValue = value;
}
```

Listing 21. Computing voltage for each stage of the envelope.

In Listing 21, from line 4 to 19, the attack, hold, decay, and sustain stages are handled, as this should be done in the timeframe when the gate signal is in the ON state. It is noteworthy that the attack stage is calculated using exponential function approach 1, which means that the voltage never really reaches the maximum value of 1. Thus, a coefficient is used as a normalization to ensure that by the end of attack time, the voltage is maximum. The calculation and usage of this coefficient can be seen on lines 6-9. The mathematical formula for the attack stage is summarized in Formula 9. It is important to note that v_0 stands for the initial value of a stage.

$$v = \frac{1-(1-v_0)\cdot b_{attack}^{\frac{-t}{t_{attack}}}}{1-(1-v_0)\cdot b_{attack}^{-1}} \qquad (9)$$

For the hold and sustain stages, there is nothing to do as the voltage remains the same stable value throughout the duration. However, calculation needs to be done for the decay state to reach the voltage sustain level. How this is done can be seen on lines 12-16 in Listing 21. This is summarized in Formula 10.

$$v = (1 - sustain) \cdot {b_{decay}}^{\frac{-t}{t_{decay}}} + sustain \qquad (10)$$

On the other hand, lines 20 to 26 contain the code for handling the action during the off state of the gate signal. When the gate signal is off, the voltage dies out in the release stage, eventually resting at 0. The mathematical formula for this calculation is as defined in Figure 11.

$$v = v_0 \cdot {b_{releasse}}^{\frac{-t}{t_{release}}} \qquad (11)$$

After this is all done, the process method only needs to update the module’s instance variables for the next iteration and set the voltage value to the output. This is shown in Listing 22. By now, the module should be done.

```
// keeping last value for re-trigger
lastValue = value;
lastGateInput = gateInput;

outputs[MAIN_OUTPUT].setVoltage(10.f * value);
```

Listing 22. Finalizing the envelope process method.

Figure 23 illustrates in detail how the developed envelope works. For this setup, a 70% pulse-width square wave is used as the gate signal for the envelope, which means that 70% of the length of a period is in ON state, and the envelope should be released in the remaining 30% of time.

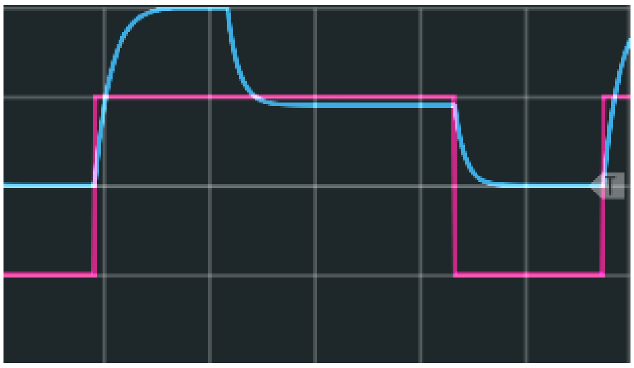

Figure 23. The modulation of an envelope on a 70% pulse-width square wave input.

As can be seen in the figure, the blue line visualizes the envelope, while the red line is a gate signal. The envelope attacks in ~5.8 ms, holds for ~5 ms, decays in ~4.1 ms to a sustained level of 45% of maximum voltage, before entering release stage and dies out in ~3.7 ms.

For practical usage, it is common to use the envelope as a modulating source for a VCA to control the sound's loudness. In this case, usually the gate signal is a note from, for example, a keyboard. This gives life to the note via nuanced loudness.

5.5 VCF

VCF, or voltage-controlled filter, is responsible for filtering out a band of frequencies from an audio signal. Whilst audio filter types can specifically be low-pass, high-pass, band-pass or notch filters, for the sake of simplicity, the module which was developed in the final year project only offers the first two options. The filter's characteristics lie in its cutoff frequency, resonance, and additional gain level, which all can be controlled using external modulating sources. The VCF is not only an important module in the categories of modular synthesizers, but it also is a great example for practical application of digital signal processing.

Outputs, inputs, and parameters

Figure 24 provides the designed panel for the VCF module. It is visible that the module has one main input and one main output for the signal targeted for filtering. Additionally, the module contains two other major parameters, one knob for selecting cutoff frequency, and one switch switching the filter type between high-pass and low-pass. In the vertical alignment are three knobs to control the filter's three parameters, namely additional range for the cutoff frequency, resonance, and DRIVE, which is the added gain for the filtered signal. Corresponding to these three parameters there are three input ports for their modulating sources.

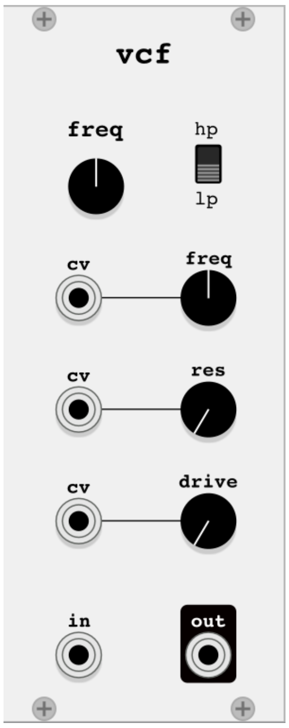

Figure 24. The panel and settings of the VCF module.

Resonance and Q factor

With the option for the resonance level, the VCF being developed can be classified as a filter with resonance, or shortly as a resonant filter. Resonance is the phenomenon in a system whereas maximum amplitude amplification occurs at certain frequencies. In the resonant filter, it usually means that it emphasizes harmonics around the cutoff frequency. Figure 25 visualizes this tendency.

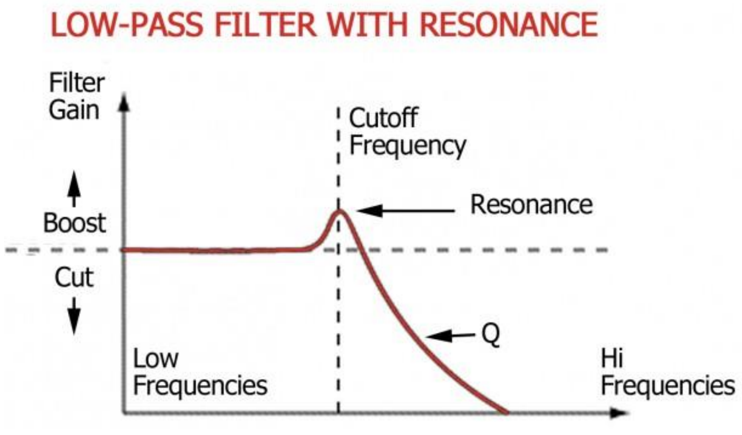

Figure 25. Resonance in the resonant filter. Copied from Gregorio [23].

What makes resonance interesting in sound-making is that it allows the resonant harmonics to surpass the primary harmonic in the original audio. This phenomenon in the resonant filter is called self-oscillation. The audio filter's function is not only to remove harmonics, but also to add colors to the original pitch and create sounds with special effects.

As Figure 11 and the section concerning filter design and the Laplace transform (pp. 23-24) already showed, the amplitude boost around the cutoff frequency, or resonance, is directly related to the position of the system's poles. The resonance level increases as the pole is placed closer to the imaginary axis on the s-plane near the cutoff frequency. However, it is noteworthy that all poles need to be on the negative-real half of the plane or as $\sigma \leq 0$ to ensure a stable system. Thus, a resonant filter with maximum resonance, meaning with $\sigma = 0$, is basically an oscillator.

Resonance creates a peak around the cutoff frequency, and the Q factor, or quality factor, is a parameter measuring that peak. In short, the Q factor describes the resonance behavior of a resonator. Formula 11 provides the definition for the Q factor, whereas f_r is the resonant frequency, and Δf is the resonant bandwidth at which the vibration power is greater than half the power at resonant frequency.

$$Q = \frac{f_r}{\Delta f} \tag{11}$$

From the formula, it is easy to see that the narrower the resonance width, the higher the Q factor. It is also equivalent to stronger resonance, and the filter's self-oscillation is more powerful. This aspect of a linear system should be taken into account as the VCF module is developed.

Implementation

Like other modules, firstly, the template for the module's source code is generated using the provided script [18]. This is straightforward and already familiar. What should mainly be concerned about is the range of value for the parameters. The cutoff frequency stays within 16 and 8192 Hz (2^4 – 2^{13}), which can be lowered or raised by a modulating source. The level for additional cutoff and resonance is within 0 and 100 percent. The DRIVE parameter can provide addition gain by a factor between 0 and 8.

In this project, a pre-developed biquadratic filter included in VCV Rack SDK is used for the main function. It requires a specified type for the filter, a Q factor, and cutoff frequency as inputs. Unfortunately, this thesis is too short to discuss the calculation in detail. In the data structure for the module, the filter is initialized as an instance variable as Listing 23 shows.

```
struct VCF : Module {
      …
      Dsp::BiquadFilter filter;
      …
}
```

Listing 23. Including a pre-made biquadratic filter in the module for internal use.

Next, the process method of the module, which is the main part, is reviewed. Firstly, it is necessary to determine the type and cutoff frequency of the filter. The cutoff frequency is the manually set cutoff parameter, including modulation from an external source. It is important to notice that the cutoff frequency is then converted to a proportion of sampling rate, in the range of 0 – 0.5. The following code demonstrates these steps.

```
float typeParam = params[TYPE_PARAM].getValue();
dsp::BiquadFilter::Type type = typeParam > 0
      ? dsp::BiquadFilter::HIGHPASS
      : dsp::BiquadFilter::LOWPASS;

float cutoff = cutoffParam + freqParam * freqInput / 10.f;
cutoff = clamp(cutoff, 0.f, 1.f);
```

```
float cutoffFreq = cutoffParamToFreq(cutoff) / args.sampleRate;
cutoffFreq = clamp(cutoffFreq, 0.f, 0.5f);
```

Listing 24. Determining cutoff frequency and filter type.

Next, the additional gain of the filter needs to be computed. For the gain, it is only necessary to combine the DRIVE parameter and the DRIVE modulation input. Then, a helper function is used to convert the DRIVE parameter to the actual gain factor. The following code demonstrates this step.

```
float driveParamToGain(float value) {
      return std::pow(1.f + value, 3.f);
}

void process(const ProcessArgs& args) override {
      …
      float drive = driveParam + driveInput / 10.f;
      drive = clamp(drive, 0.f, 1.f);
      float gain = driveParamToGain(drive);
      …
}
```

Listing 25. Determining additional gain for the filtered signal.

On the other hand, it is a little tricky with the resonance and Q factor. At maximum resonance, the filter becomes an oscillator, which makes its Q factor infinitely large. This extreme case is handled as an exception in another helper function that converts the resonance level to the Q factor. It is also noted that the filter should not be naturally overdamped, or in other words, having the Q factor smaller than 0.5. The following code calculates the Q factor of the filter based on its parameter and modulation.

```
float resonanceParamToQFactor(float resonance) {
      // q factor should by default not overdamped (< 0.5)
      // with resonance close to 100%, mimic analog self-oscillation due to in-
finite Q factor
      float q = 0.5f * std::pow(50.f, resonance);
      if (value < 0.9) {
            return q;
      } else {
            return q * std::pow(HUGE_VALF, 5.f * (resonance - 0.9f));
      }
}

void process(const ProcessArgs& args) override {
      …
      float resonance = resParam + resInput / 10.f;
      resonance = clamp(resonance, 0.f, 1.f);
      float qFactor = resonanceParamToQFactor(resonance);
      …
}
```

Listing 26. Determining additional gain for the filtered signal.

Having all the elements, the function can now directly compute the output for the filtered signal. However, there is a small detail to notice. Self-oscillation only happens in analog systems where a certain level of noise is always available. This is not possible in digital systems. Thus, a little noise is intentionally added to bootstrap self-oscillation. Listing 27 demonstrates this imitation and finalizes the process function of VCF module.

```
float in = inputs[MAIN_INPUT].getVoltage();
in = gain * in / 5.f;

// add a bit of noise to bootstrap self-oscillation
in += 1e-3f * (2.f * random::uniform() - 1.f);

filter.setParameters(type, cutoffFreq, qFactor, 0.f);
float output = 5.f * filter.process(in);
outputs[MAIN_OUTPUT].setVoltage(output);
```

Listing 27. Bootstrap self-oscillation and computing the filtered signal.

The following figures 26 and 27 provide an example of the working VCF. The major concerns of this thesis, the two main functions of the VCF, including self-oscillation and harmonics removal, are examined. The figures are followed by descriptions in detail.

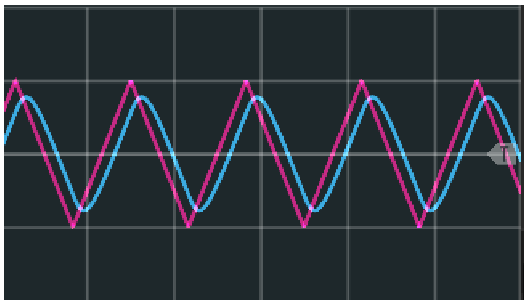

Figure 26. The removal of high frequencies from a triangle wave using the VCF.

In Figure 26, the removal of high frequency from a triangle wave by the VCF is demonstrated. A triangle wave is passed through a low-pass VCF at cutoff frequency 284 Hz. The red line is the original waveform, and the blue line is the output signal. The output signal has a lot less discontinuity due to the removal of high frequencies.

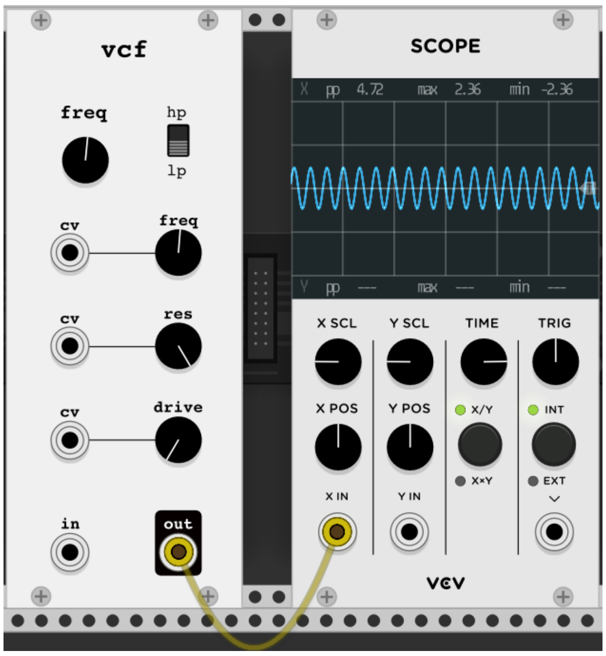

Figure 27. Self-oscillation in resonant filter.

On the other hand, Figure 27 shows how the VCF can self-oscillate. With resonance level set to maximum, the filter generates a perfect sine wave without any input.

6 Results and discussion

6.1 Comments on the outcome of the project

The main goal of this thesis is to introduce digital signal processing and examine it in the context of audio application development. These aims were attained by providing essential information on the field and documenting the development of an open-source synthesizer. The subject was approached using simplistic, minimal terms, in an attempt to convert the challenges of this field into enthusiasm. This thesis hopes to have offered a sufficiently detailed overview. It lays the stepping stones for further research in ICT and gives examples of practical usage, namely modular synthesis and audio processing.

Regarding the application, it was considered that a sound-making patch should be included in the project. The patch should be followed by detailed diagrams and explanation, to show how modular synthesis works and how the signal changes in each step. However, most of the modulations in a patch are only examinable in real-time interaction with the application. Thus, the topic was omitted from the study.

6.2 Incomplete development and possible improvement

Even though the project is finished and fully functional, there are several persisting limitations. These drawbacks can be seen in both the theoretical and project development sections. Although they did not prevent this project from achieving its set goal, they certainly lead to compromises.

The theoretical part deals with the most foundational aspects of the field of DSP. The field itself is immensely large, and at its core, it uses mathematics as its principal language. Due to the scope of the study, core concepts and theories could only be examined mainly based on intuition instead of mathematical proofs. Furthermore, a myriad of theoretical aspects was overlooked to avoid making the thesis excessively long. Especially, the section regarding the Laplace transform and filter design is a truly vast domain; however, it could only be discussed very briefly here.

In addition to this, the development of the project was also compromised. The project was initially aiming to start development from scratch, but the scale was too large and it was decided that the thesis should be based on the proof of concept method. The working of the processing engine, especially its multithreaded programming, is in itself very interesting, but unfortunately this could not be explored deeper here. The section on modules development for the synthesizer was also quite minimal. Hopefully, it builds the foundation for further development although a few more modules should have been included if given additional resources.

Compromises were made, and they also open up a lot of possible improvements. Based on the skeleton of the project, further module development could have been done, and more explanation regarding the theoretical part could have been included in the thesis. Additionally, the modules could have been re-implemented to more closely resemble the original analog products, which produce sounds with more nuances and also help understand continuous signal processing. The targeted topic is inevitably huge, and this thesis aims only to discuss the most foundational aspects. However, it is expected that from the foundation, further work is possible.

7 Conclusion

The aim of this thesis has been to introduce digital signal processing (DSP) and what is known about it. Additionally, the thesis has examined DSP in the context of an audio application, focusing on the usage and practicality of the field. This should illustrate the close relationship between modern audio processing and DSP, showing that the field, although vast and challenging, is practical and captivating to a significant extent.

To achieve the intended goal, the study was conducted from both theoretical and practical aspects. Fundamental knowledge of signal processing, namely the representation of signal and DSP systems, have been explained and clarified, in order to provide useful information about the field. The theories were exemplified and applied in the development of a project, which included the implementation of a collection of audio processing modules for an open-source digital synthesizer. The finished set of modules could be used to create extensible patches of sounds in modular synthesis, which is a common form of modern electronic music.

The thesis has discussed the concepts and direct applications of DSP in the focused area of audio processing. As a whole, the thesis has thoroughly investigated digital signal processing in theory and practice, without evading the essential challenges for a newcomer. Compromises had to be made during the progress of the project, such as the underdevelopment and exclusion of certain modules. Nonetheless, in general, the thesis achieved its set goals.

Regarding the state of constantly advancing ICT technologies, further work can be conducted on signal processing and audio applications based on this thesis. For a deeper understanding of the topic, it is recommended that the reference list should be checked out. Most of the materials discussed in this thesis are available online and they can be easily accessed. In addition, textbooks should be looked at for more detailed explanations and proofs. In the modern era, the significance of information and information processing are only expanding, which makes the knowledge of DSP digital signal processing of great importance.

References

1 Smith S.W. The Scientist and Engineer's Guide to Digital Signal Processing. 2nd Edition. San Diego, California. California Technical Publishing; 1999.

2 Mitra S.K. Digital Signal Processing: A Computer-Based Approach. 4th Edition. New York, NY. The McGraw-Hill Companies, Inc; 2011.

3 Nasir B.M. DSP Based Analysis, Processing and Synthesis of Sound. IEE Colloquium on Audio and Music Technology: The Challenge of Creative DSP. London, UK; 2002.

4 Prandoni Paolo, Vetterli Martin. Signal Processing for Communications. 1st Edition. Italy. EPFL Press; 2008.

5 Fifty Years of Signal Processing: The IEEE Signal Processing Society and Its Technologies 1948-1998 [online]. The IEEE Signal Processing Society; 1998. Available from: https://signalprocessingsociety.org/uploads/history/history.pdf. Accessed 5 January 2020.

6 Luke H.D. The Origins of Sampling Theorem. IEEE Communications Magazine, vol. 37, no. 4; April 1999. pp. 106-108.

7 Shannon C. E. Communication in the Presence of Noise. Proceedings of the IRE, vol. 37, no. 1; January 1949. pp. 10-11.

8 Oppenheim A. V. Algorithm Kings: The Birth of Digital Signal Processing. IEEE Solid-State Circuits Magazine, vol. 4, no. 2; 2012. pp. 34-37.

9 Thon L. E. 50 Years of Signal Processing At ISSCC. 2003 IEEE International Solid-State Circuits Conference, Digest of Technical Papers. ISSCC. San Francisco, CA, USA; 2003. pp. 27-28.

10 Radar – Basic, Types & Applications [online]. Elprocus. Available from: https://www.elprocus.com/radar-basics-types-and-applications/. Accessed 5 January 2020.

11 Irion Klaus Loureiro, Hochhegger Bruno, Marchiori Edson, Porto Nelson da Silva, Baldisserotto Sérgio de Vasconcellos, Santana Pablo Rydz. Chest X-ray and Computed Tomography in the Evaluation of Pulmonary Emphysema. J. bras. pneumol. [online]. vol. 33, no. 6; December 2007. pp. 720-732. Available from: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1806-37132007000600017&lng=en. http://www.scielo.br/img/revistas/jbpneu/v33n6/en_a17fig01ab.jpg. Accessed 5 January 2020.

12 What Is an Audio Compressor [online]? Practical Music Production. Available from: https://www.practical-music-production.com/audio-compressor/. Accessed 5 January 2020.

13 Wang Ge. A History of Programming and Music. In: Collins, Nick and Julio d'Escriván (eds.). Cambridge Companion to Electronic Music. Cambridge: Cambridge

University Press; 2007.
Available from: https://www.researchgate.net/publication/265164952_A_History_of_Programming_and_Music. Accessed 6 January 2020.

14 Pure Dara Floss Manuals [online]. Write.flossmanuals.net
Available from: http://write.flossmanuals.net/pure-data/oscillators/. Accessed 6 January 2020.

15 Barbosa Lucas V. Fourier Transform Time and Frequency Domains [online]. Common.wikipedia.org.
Available from: https://commons.wikimedia.org/wiki/File:Fourier_transform_time_and_frequency_domains_(small).gif. Accessed 6 January 2020.

16 A Filter Primer [online]. Maximintegrated.com.
Available from: https://www.maximintegrated.com/en/design/technical-documents/tutorials/7/733.html. Accessed 20 February 2020.

17 Designing Digital Filters [online]. Eetimes.com.
Available from: https://www.eetimes.com/designing-digital-filters/. Accessed 21 February 2020.

18 Plugin Development Tutorial [online]. Vcvrack.com.
Available from: https://vcvrack.com/manual/PluginDevelopmentTutorial. Accessed 26 February 2020.

19 Andrew Belt. Rack v1 Development Blog [online]. Community.vcvrack.com.
Available from: https://community.vcvrack.com/t/rack-v1-development-blog/1149/493. Accessed 26 February 2020.

20 Panel Guide [online]. Vcvrack.com.
Available from: https://vcvrack.com/manual/Panel#designing-your-panel. Accessed 7 March 2020.

21 AHDSR [online]. Forum.pianoworld.com.
Available from: http://forum.pianoworld.com/ubbthreads.php/topics/2308511/AHDSR.html. Accessed 17 March 2020.

22 Voltage Standards [online]. Vcvrack.com.
Available from: https://vcvrack.com/manual/VoltageStandards. Accessed 17 March 2020.

23 Know Your Filters: Low Pass, Band Pass, High Pass, Resonance [online]. Pedals.thedelimagazine.com.
Available from: https://pedals.thedelimagazine.com/production-advice-know-your-filters-low-pass-band-pass-high-pass-resonance/. Accessed 25 March 2020.