Low-Latency Digital Guitar Effects Using Signal Processing with Python in Real Time

Benjamin Åberg

Master's Thesis in Computer Engineering
Supervisor: Jerker Björkqvist
Faculty of Science and Engineering
Åbo Akademi
2024

Abstract

This thesis presents a comprehensive exploration of implementing common guitar effects in real time, using signal processing techniques with Python and some of its libraries. One key focus of the thesis is latency reduction using Cython.

The thesis begins with an overview of digital signal processing (DSP) fundamentals and common effects for the electric guitar, such as distortion, delay and reverberation. Some effects' algorithmic implementation is also discussed, highlighting the main components and parameters required for real-time processing.

Subsequently, the thesis introduces Python as a powerful tool for prototyping and implementing DSP algorithms. Utilising libraries such as NumPy, Sounddevice and Librosa, the feasibility of real-time guitar effects processing within the Python environment is demonstrated. Moreover, the flexibility of Python, facilitating rapid experimentation and algorithm refinement crucial for achieving desired sound characteristics, is also highlighted.

To address the challenge of latency inherent in software-based signal processing, the benefits of Cython, a superset of Python designed to optimise code performance, are explored. Through Cython's capability of compiling Python code to native machine code, significant latency reductions are achieved without compromising computational efficiency.

Experimental results demonstrate the effectiveness of the proposed approach in achieving low-latency digital guitar effects processing in real time. Comparative latency measurements reveal improvements over traditional Python implementations, highlighting the potential adequacy as well as importance of Cython optimisation for latency-sensitive applications.

Keywords: Python, Cython, digital guitar effects, audio processing, signal processing, latency reduction

Abbreviations

A/D	Analogue-to-Digital
API	Application Programming Interface
CD	Compact Disc
CPU	Central Processing Unit
D/A	Digital-to-Analogue
DAW	Digital Audio Workstation
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
EQ	Equaliser
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
IEM	In-Ear Monitoring
IFT	Inverse Fourier Transform
IIR	Infinite Impulse Response
I/O	Input / Output
LPC	Linear Predictive Coding
LSB	Least Significant Bit
OS	Operating System
PCM	Pulse Code Modulation
STFT	Short-Time Fourier Transform
VM	Virtual Machine

Table of Contents

1. Introduction 1

1.1 Guitar Effects – A Brief History 2

2. Digital Signal Processing 3

2.1 Digital Audio Processing as a Concept 3

2.2 A/D – D/A Conversion 4

2.2.1 Quantisation 6

2.2.2 Dithering 8

2.3 The Fourier Transform 9

2.4 Filters 11

2.4.1 Low-Pass Filters 12

2.4.2 High-Pass Filters 13

2.4.3 Band-Pass Filters 14

2.4.4 Band-Stop Filters 14

2.4.5 Digital Filters: FIR and IIR 15

2.5 Digital Audio Formats 17

2.6 Audio Quality & Data Rate 19

3. Latency Reduction 21

3.1 Latency of DSP Systems in General 23

3.2 Real-Time Audio Processing in Python 27

3.2.1 PyAudio 27

3.2.2 Cython 29

4. Visualisation of Common Guitar Effects 31

4.1 Overdrive (Distortion) 34

4.2 Reverberation 35

4.3 Delay 36

4.4 Phaser 37
4.5 Wah-wah 38
5. Python Experimentation 39
5.1 Signal Chain 40
5.2 Audio Analysis in Python 41
5.3 Implementation of Guitar Effects 44
5.3.1 Overdrive 45
5.3.2 Reverberation and Echo 46
5.3.3 Harmoniser 47
5.4 Latency Reduction Using Cython 50
6. Results 52
6.1 Evaluation of Guitar Effect Implementations 52
6.2 Latency Reduction 54
7. Discussion 55
8. Conclusion 57
9. Summary in Swedish – Svensk sammanfattning 59
References 62
Appendices 67
Appendix A – Yamaha Pacifica 112J Technical Specifications 67
Appendix B – Focusrite Scarlett 2i2 Technical Specifications 68
Appendix C – AMD Ryzen 7 7800X3D Technical Specifications 70
Appendix D – Audiovisual Presentation of Implemented Effects 71

1. Introduction

Throughout the history of music, there has been a constant drive for innovation among musicians to explore new sounds and techniques when it comes to playing, recording and mixing. Guitar effects have played a crucial role in obtaining new sounds and have opened endless possibilities for guitarists wanting to explore their musical abilities. With the help of computers and digital signal processing solutions in recent years, an even broader spectrum of options for audio experimentation is now available.

Digital signal processing solutions have brought consistency and reliability for musicians, allowing effects to be replicated, tweaked and combined in ways not possible using their analogue counterparts [1]. Worth noting is also the availability of highly powerful processors in today's world, allowing more advanced effects to be discovered through experimentation, that has not been possible to implement in real time previously.

An important aspect of audio processing to take into consideration is latency and minimising its presence as well as impact. In compliance with the *precedence effect*, also known as the *Haas effect*, identical sounds separated by less than 50 ms cannot be distinguished between by the human ear and are instead perceived as a single source [2]. For a live musical performance, keeping latency below this threshold is key for optimal sound and feedback.

This master's thesis explores the world of signal processing relevant to audio processing and manipulation by highlighting key concepts, techniques and solutions available, as well as briefly discussing latency reduction on a theoretical level. The thesis is divided into two theory parts and one experimental part: digital sound processing theory, latency reduction theory and experimentation with audio signals in Python.

The main goal of this thesis is to introduce the signal processing side of Python for implementing real-time guitar effects by presenting hands-on solutions. The thesis also aims to highlight the versatility of digital solutions by implementing an effect not viable to implement using traditional, analogue methods. The focus of the thesis is on giving basic implementations of common guitar effects alongside reducing signal processing latency using Cython.

1.1 Guitar Effects – A Brief History

Audio effects for the electric guitar have been an essential part of not just the instrument but the music industry as a whole throughout history. Effects for the electric guitar were originally discovered shortly after the invention of the electric guitar amplifier back in the 1930s, when guitarists noticed that turning up the amplifier would result in an extorted “overdrive” sound [3]. During the 40s and 50s, effects such as reverb, echo and tremolo became standard effects built into guitar amplifiers with the help of transistor technology. In the 60s and the 70s, the guitar effect market took off with numerous new effects such as chorus, flanger and ring modulation entering the market [3]. At this time, standalone guitar effects built into pedals that could be operated separately from the amplifier while standing or sitting became popular [2]. During the 80s and the 90s, digitalised solutions enabled manufacturers to start creating all-in-one effect pedals and devices that set the foundation for modern guitar effect systems. In the early 2000s, these pedals increased heavily in popularity and became a common choice among many guitarists [3].

In modern times, guitar effects remain an essential part of every guitar player’s toolbox and are becoming more and more both versatile and affordable. From 2005 to 2021, the sales of guitar effect pedals sold in the United States saw a steady increase from around 1.1 million units sold in 2005 to around 1.5 million units sold in 2021 [4]. Today, digital solutions for amplifiers exist with built-in effects such as the Vox Air GT that supports simulating the sound of different amplifier brands and models as well as a wide range of effect pedals [5].

2. Digital Signal Processing

The concept of digital signal processing (DSP) has been key for the development of new technologies across many fields since digital computers became available in the 1960's, with some of the original applications being radar systems, space exploration and medical equipment to name a few [1]. Today's digital world relies more than ever before on DSP, with modern applications such as telecommunication, image processing as well as audio processing, the main topic of this thesis.

Section 2.1 discusses DSP applied to audio signals on a theoretical basis, highlighting key concepts and techniques used. The topics of Sections 2.2 and 2.3 respectively, are A/D (analogue to digital) & D/A (digital to analogue) conversion as well as the Fourier transform. Sections 2.4 and 2.5 respectively discuss filtering as well as digital audio formats. Audio quality and data rate are the topics of Section 2.6.

2.1 Digital Audio Processing as a Concept

As defined in [6, p. 1], digital audio processing refers to "the digital representation of signals and the use of digital hardware to analyse, modify or extract information from these signals." As also mentioned in [6], the key benefits of digital signal processing solutions are:

1. Flexibility: digitally modified signals can easily be modified while analogue signals often require hardware changes.
2. Reproducibility: digital signals can be identically reproduced from one system to another while analogue signals rely on hardware that may vary in performance.
3. Reliability: digital solutions do not change over time as electrical components of analogue systems may do.
4. Complexity: processing-heavy applications such as machine learning-based solutions in modern times would not be feasible on analogue systems.

Within the field of music, DSP plays an important role during the whole process from the recording session to the final mix, with features such as filtering, equalisation and compression being key. Today's musical world also relies more heavily on modern

technology than ever before, with machine learning-based solutions finding their way into audio processing and mixing. A recent example of this, at the time of writing this thesis, is separating different instruments and voice recordings from a single track into separate tracks (e.g. the song “Now And Then” by The Beatles, November 2023).

2.2 A/D – D/A Conversion

Converting an analogue signal to a digital signal and vice versa is what makes digital audio processing possible in the first place. Audio signals as they exist in nature are analogue, *continuous* signals and need to be converted into digital, *discrete* signals in order to be processed digitally (A/D) [1]. Vice versa, obtaining the processed signal for listening requires the digital signal to be converted to analogue (D/A).

A continuous or continuous-time signal is a signal with a parameter that can obtain values over a continuous range (e.g. voltage) related to time and can be expressed as a function $x(t)$ [1] [7]. This continuous signal can be converted into a sequence of numbers known as a discrete, or discrete-time, signal expressed as $x(n)$ by *sampling* and the same process can be performed in reverse [7]. This is the idea of A/D & D/A converters in their simplest forms. An illustration of a typical, generalised DSP system can be found in Figure 2.1 below.

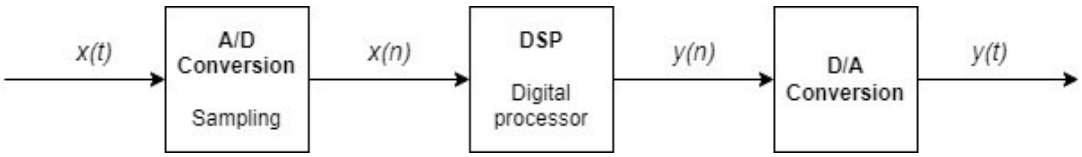

Figure 2.1: Block diagram of a general DSP system.

Furthermore, *low-pass filters* are often applied to the input signal as well as to the output signal according to Figure 2.2 below. This is done in order to eliminate high frequencies outside the spectrum of interest that could result in *aliasing*, the overlapping of frequency components in the sampled signal [8]. Filters are discussed in more detail in Section 2.4.

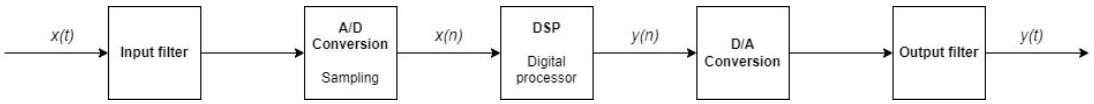

Figure 2.2: Block diagram of a general DSP system with filters.

The sampling of a continuous signal with the highest frequency component f_{max} can be done according to the *Nyquist-Shannon Sampling Theorem*, which states that the minimum sampling rate f_s should be at least double the size of f_{max} , or $f_s > 2f_{max}$, while still keeping the signal's information content intact [7] [8]. f_{max} is known as the *Nyquist frequency* at which point aliasing occurs if the signal is sampled with a sampling rate less than $2f_{max}$ [8].

The amplitude of the signal during the sampling period $T_S = 1/f_s$ needs to be *quantised*, meaning converted into a number sequence $x(n)$ by a *quantiser* (see Section 2.2.1). This number sequence is used during the digital signal processing of the signal. Figure 2.3 by [7] below shows a schematic diagram of the analogue-to-digital, as well as back-to-analogue, conversion process.

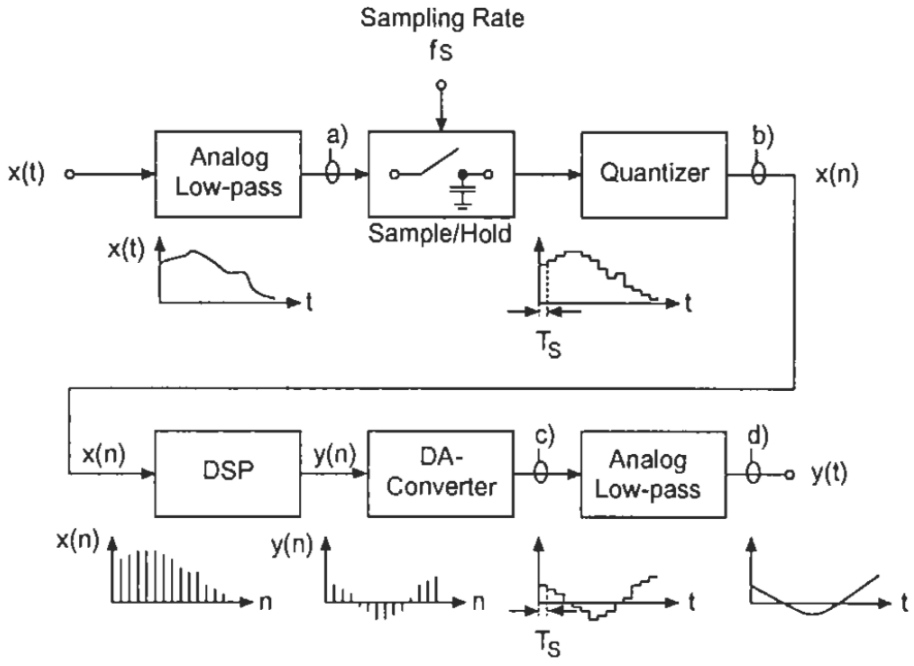

Figure 2.3: Schematic diagram of the A/D – D/A process [7].

Digital-to-analogue conversion is required post-processing if the goal is to reproduce the signal sonically, for example, by driving a speaker. Below is an example of a D/A converter producing the analogue output signal by mapping the digital code to the corresponding analogue value [8]. This produces a staircase-like shape of the signal, as illustrated below in Figure 2.4 by [8], since the value is held during the time T for each converted value. This effect, known as *imaging*, can be reduced with a low-pass filter functioning as an *anti-imaging filter* to smooth out the steps [8].

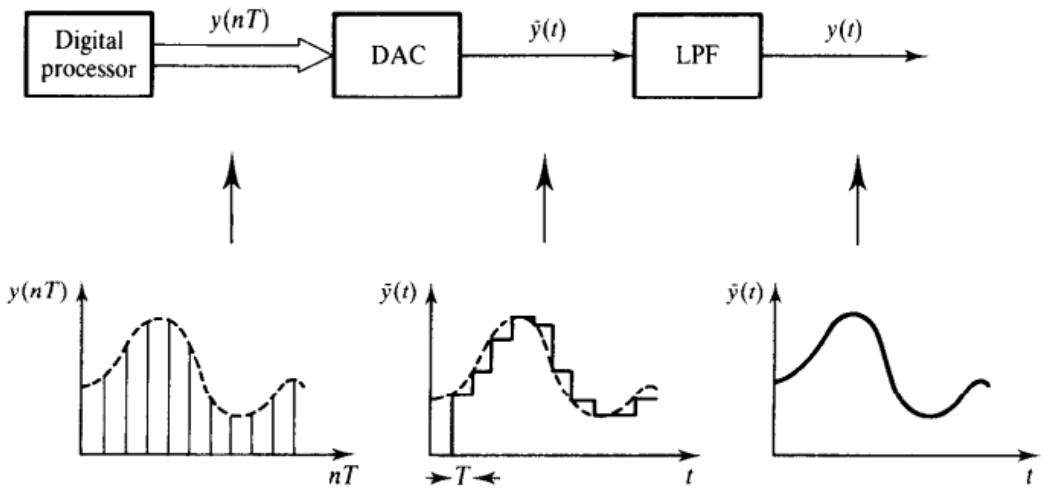

Figure 2.4: Digital-to-analogue conversion process [8].

This type of D/A converter is known as *zero-order hold*, meaning that the converter holds each value during the duration of a time sample until the next sample is received [8]. To overcome this drawback of zero-order hold D/A converters, a digital processor can be used to interpolate the signal between the different samples.

2.2.1 Quantisation

The idea behind quantisation is approximating the analogue signal's amplitude using the discrete samples in order to digitise the signal [9]. A continuous signal with an amplitude consisting of values ranging from $-\infty$ to $+\infty$ can be modelled discretely by approximating the amplitude from a finite set of real numbers at each sample of the

signal, with a maximum error of ±½ LSB (Least Significant Bit) [1] [8] [9]. A quantiser's precision is measured in bits that are used for representing the output signal in binary words according to $N = 2^n$, where N is the number of output points and n is the bit precision [1] [9]. For example, a 12-bit quantiser has $2^{12} = 4096$ possible output values.

Quantisation of a signal x in practice is equivalent to the addition of a uniform distributed random noise signal e (error) and the unquantised input signal x [1] [7]. As a result, this model of quantisation is only applicable if the error can be treated as random, meaning the signal does not remain the same over numerous consecutive samples. This is illustrated in Figure 2.5 by [1] below, where (a) represents the original analogue signal, (b) represents the sampled analogue signal, (c) represents the digitised signal and (d) represents the quantisation error. The output of the A/D converter in this example is the signal represented by (c), where noise (d) has been added to the sampled signal (b). For slowly varying signals, *dithering* can be utilised (see Section 2.2.2) [1].

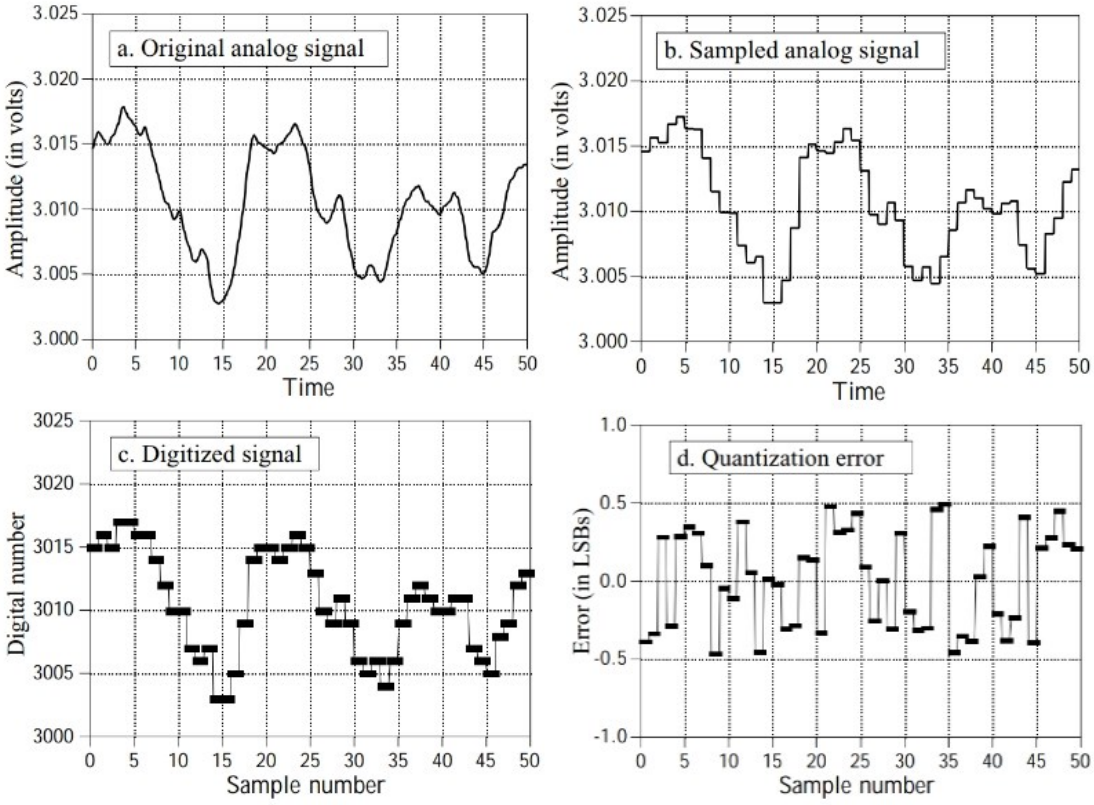

Figure 2.5: Example of the digitisation process [1].

2.2.2 Dithering

Digitising a signal using the quantisation error is not feasible if the signal is of slowly varying nature, meaning that it remains around the same value across many samples [1]. To overcome this, a technique known as dithering can be used to introduce random noise to the signal. As illustrated in Figure 2.6a by [1] below, the digitised signal values do not follow the analogue signal due to the original signal varying less than ±½ LSB. By introducing noise to the signal, as shown in Figure 2.6b by [1], the changes in the original signal become more apparent in the digitised signal, as shown in Figure 2.6c by [1]. The noise introduced in this case is normally distributed and has a standard deviation of ⅔ LSB, which results in a peak-to-peak amplitude of 3 LSB [1].

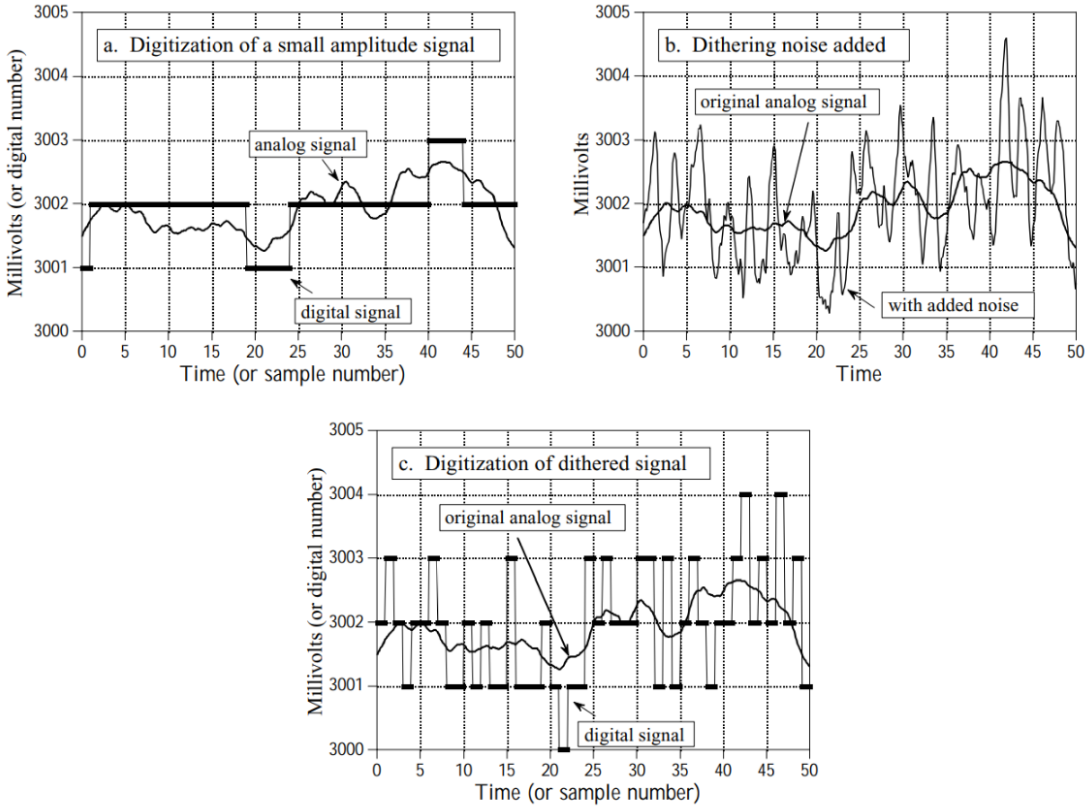

Figure 2.6: Illustration of signal dithering [1].

2.3 The Fourier Transform

One key aspect of signal processing is analysing the frequency spectrum of signals for the purpose of, for example, extracting frequencies of interest, analysing the behaviour of the signal or making sure that the signal behaves as expected. For continuous, periodic waveforms (Figure 2.7), the *Fourier series* has traditionally been used to model the signals mathematically. However, most waveforms found in nature are non-periodic (Figure 2.8), meaning that the Fourier series in its basic form is not applicable for signal analysis. As a result, one of the most fundamental and most widely used concepts of signal processing was formed, the *Fourier transform* – a modified version of the Fourier series [8] [10].

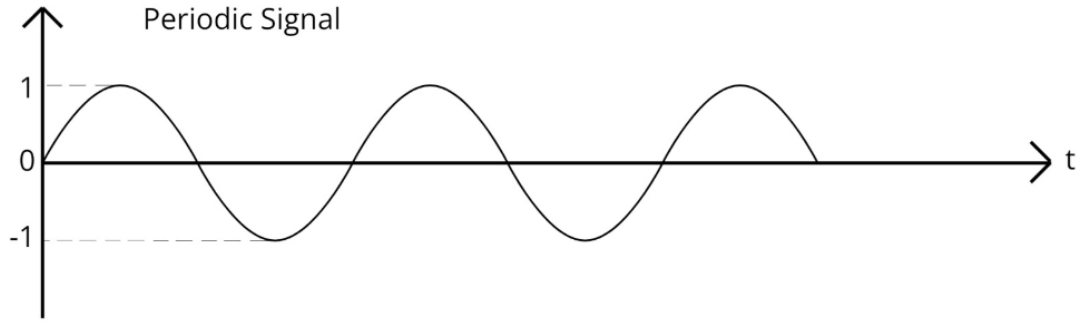

Figure 2.7: Example of a periodic waveform [11].

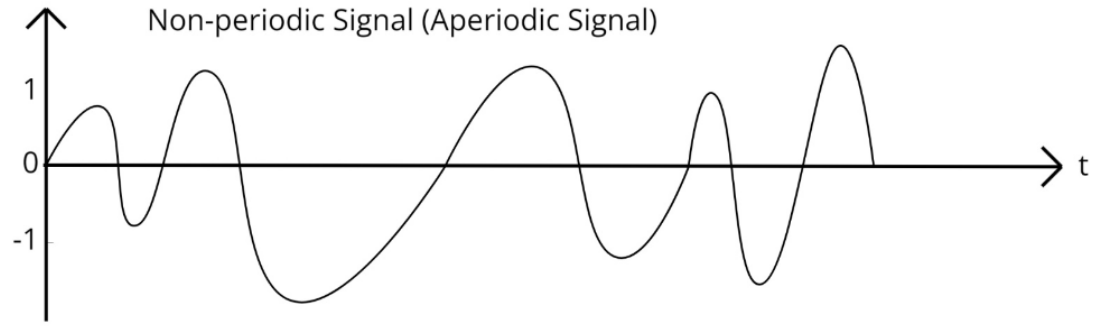

Figure 2.8: Example of a non-periodic waveform [11].

Simply put, the Fourier transform is a function for extracting the frequencies present in a signal by analysing the amplitude and phase components of each sinusoid [12]. It can be used to convert a function of time $h(t)$ to a function of frequency $H(f)$ and is described mathematically in Equation 2.3.1 [8] [10].

$$\boldsymbol{H(f) = \int_{-\infty}^{\infty} h(t) e^{-2\pi j f t}\, dt = H(j\omega) = \int_{-\infty}^{\infty} h(t) e^{-j\omega t}\, dt} \tag{2.3.1}$$

Similarly, its inverse (IFT) can be used to convert a function of frequency $H(f)$ back to time-space $h(t)$, as described mathematically in Equation 2.3.2 [8] [10].

$$\boldsymbol{h(t) = \int_{-\infty}^{\infty} H(f) e^{2\pi j f t}\, df = h(t) = \int_{-\infty}^{\infty} H(j\omega) e^{j\omega t}\, df} \tag{2.3.2}$$

Due to the nature of analogue signals containing infinitely many data points, applying the Fourier transform directly on analogue signals is highly impractical [8]. For this reason, the *Discrete Fourier Transform* (DFT) was developed for use on discrete data instead of the standard Fourier transform that can only be used on continuous data. As a result, the signal to be examined must first be sampled into discrete data points, as previously discussed in Section 2.2 [8]. A commonly used algorithm for calculating the DFT is the *Fast Fourier Transform* (FFT). The DFT formula is described mathematically in Equation 2.3.3.

$$\boldsymbol{F[n] = \sum_{k=0}^{N-1} f[k] e^{-j\frac{2\pi}{N}nk} \ (n = 0, 1, \dots, N-1)} \tag{2.3.3}$$

$F[n]$ is the DFT of the sequence $f[k]$ and n represents the harmonic number of the transform component [8]. Similarly as for the standard Fourier transform, the inverse DFT can be expressed mathematically, as shown in Equation 2.3.4.

$$\boldsymbol{f[k] = \frac{1}{N} \sum_{n=0}^{N-1} F[n] e^{j\frac{2\pi}{N}nk}} \tag{2.3.4}$$

2.4 Filters

Filtering is an essential part of both analogue and digital signal processing. Analogue filters work by running the signal through individual, electrical components that interact with the signal in the desired way while digital filters work by manipulating the signal digitally, for example, by using mathematical algorithms on signal data stored in computer memory [8] [13]. Signal filtering can be useful during the whole signal processing phase, as previously shown in Figure 2.2, and can, for example, be used for (1) filtering of the raw, analogue signal, (2) filtering of the digitally replicated signal and (3) filtering of the reconstructed, analogue signal.

Analogue and digital filters each have advantages and disadvantages over the other, depending on various factors. The main advantages of digital filters, as presented in [8], are:

1. Can implement certain features that are practically impossible using analogue filters due to inconsistencies in electrical components.
2. Performance is independent of external factors such as temperature, which eliminates the need for periodic calibration.
3. Filter variables such as frequency response can be adjusted programmatically.
4. Data can be stored in memory for future use, both filtered and original.
5. Filter performance is reproducible, as it does not depend on electrical components which may have slight inconsistencies.

Analogue filters may, however, be preferred in certain situations, with the main advantage over digital filters presented in [8] being speed. Digital filters depend on the speed of the underlying processor while analogue filters do not. Hence, this factor has to be considered, especially for real-time signal processing.

Filters are generally classified based on their frequency selectivity, meaning the frequencies they let through. Depending on the frequency selectivity, filters can have (1) *low-pass*, (2) *high-pass*, (3) *band-pass* or (4) *band-stop* response, as described in [13] as:

(1) Only passes through low frequencies and high frequencies are significantly reduced.
(2) Only passes through high frequencies and low frequencies are significantly reduced.

(3) Only passes through frequencies in the middle band of the frequency range, while frequencies outside the band are significantly reduced.
(4) Only passes through frequencies outside the middle band of the frequency range, while frequencies inside the band are significantly reduced.

Frequency selective filters are an essential tool within the music industry and are commonly used in *equalisers* (EQs) for elevating or cutting specific frequency bands in the audio signal [2]. For this reason, digital solutions have become widely available and are a standard in most *Digital Audio Workstations* (DAWs) today [2]. For guitarists, EQ pedals are available that allow the player to directly tune the frequency bands present in the signal coming from the guitar. These are essential as they allow the player to mix the frequency in real time according to the environment.

The above-mentioned filter types are described more in detail and visualised in Sections 2.4.1 – 2.4.4. Section 2.4.5 gives an introduction to the two main digital filter types, namely *finite impulse response* filters (FIR) and *infinite impulse response* filters (IIR).

2.4.1 Low-Pass Filters

The frequency plane can typically be divided into three areas: the *passband*, the *stopband* and the *transition area*. The passband defines the frequency range included in the filtered signal, the stopband defines the frequency range excluded from the filtered signal and the transition area is the frequency range between these [13]. The low-pass filter, as previously mentioned, filters out frequencies above the defined passband edge frequency f_{pass}, with the stopband ranging from f_{stop} to infinity. The preferred signal gain in the passband should be between 0 dB and a_{pass} while the gain in the stopband should be between $-\infty$ and a_{stop} [13]. Low-pass filters are typically used when it is of interest to eliminate frequencies of a signal above a certain threshold. During A/D & D/A conversion, low-pass filters are commonly used both before the sampling process to keep the signal frequency below the Nyquist frequency and after the D/A conversion to eliminate imaging, as previously discussed in Section 2.2. Figure 2.9 by [13] below illustrates an example of a low-pass filter specification.

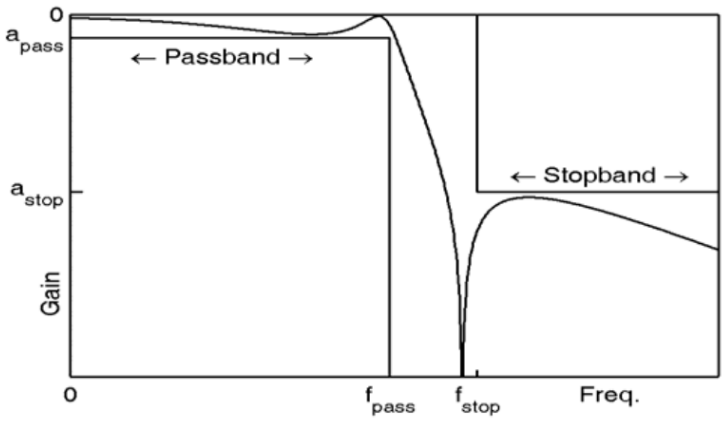

Figure 2.9: Example of a low-pass filter specification [13].

2.4.2 High-Pass Filters

Similarly to the principle of low-pass filters, high-pass filters allow frequencies in the range of f_{pass} to infinity to pass, while filtering out frequencies in the range 0 to f_{stop} and keeping the signal gain below a_{pass} in the passband and a_{stop} in the stopband. As previously mentioned, high-pass filters are used to eliminate frequencies of a signal below a certain threshold, for example, unwanted low-frequency noise such as engine rumble. An example of a high-pass filter specification is illustrated in Figure 2.10 by [13] below.

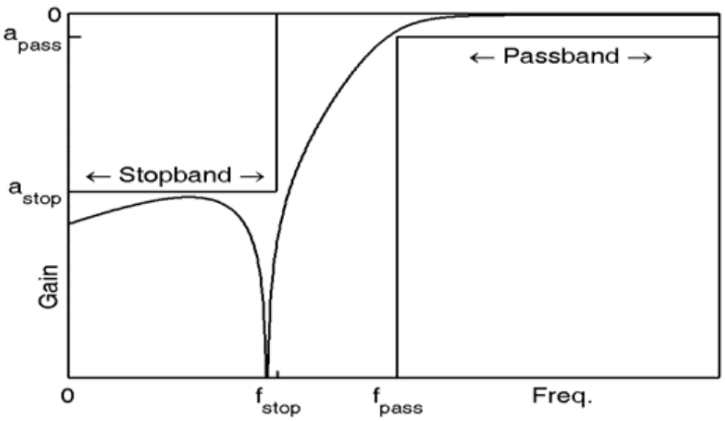

Figure 2.10: Example of a high-pass filter specification [13].

2.4.3 Band-Pass Filters

As previously mentioned, band-pass filters filter out frequencies outside the defined band, with a frequency range of f_{pass1} to f_{pass2}. The band-pass filter has two stopbands, namely 0 to f_{stop1} and f_{stop2} to infinity [13]. Similarly to the other filters, the band-pass filter also implements the defined gain parameters a_{pass} and a_{stop} for determining signal attenuation. For certain applications, it can be useful to define separate stop-band gain parameters a_{stop1} and a_{stop2}, in order to have different attenuation depending on stopband [13]. Stopbands are used to keep the frequencies of a signal between f_{pass1} and f_{pass2} and an example of this is voice recordings, as the human voice typically ranges from 300 Hz to 3000 Hz [13]. An example of a band-pass filter specification is illustrated in Figure 2.11 by [13] below.

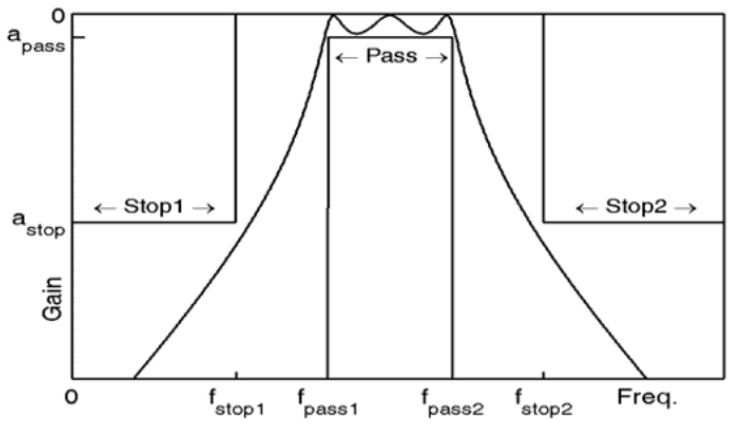

Figure 2.11: Example of a band-pass filter specification [13].

2.4.4 Band-Stop Filters

Finally, band-stop filters are used to filter out frequencies in the defined band between f_{pass1} and f_{pass2}. The frequencies allowed to pass are 0 to f_{pass1} as well as f_{pass2} to infinity. Opposite to the band-pass filter, the band-stop filter has a single defined gain parameter a_{stop} for the stopband while the passband may have individual gain parameters a_{pass1} and a_{pass2} if necessary [13]. Band-stop filters are typically used to

filter out undesired signal noise at a certain frequency (notch filters). Figure 2.12 by [13] below illustrates an example of a band-stop filter specification.

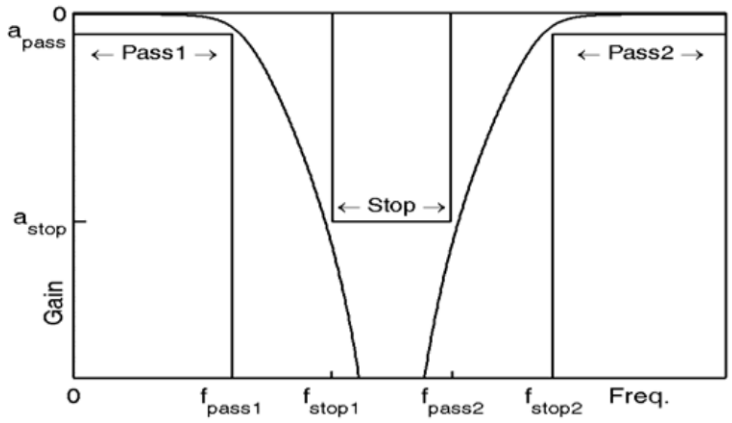

Figure 2.12: Example of a band-stop filter specification [13].

2.4.5 Digital Filters: FIR and IIR

Digital filters are typically categorised based on their impulse response, which in simple terms can be described as how the output signal is affected when the filter reacts to an input signal impulse. *Finite Impulse Response* (FIR) filters are digital filters that only depend on past and current values of the input signal by calculating the filter coefficients directly, based on the preferred frequency response of the filter. As a result, the filter is said to be of non-recursive nature [13]. The impulse response $h(k)$ of these filters is generally finite, hence the name. The impulse response of a FIR filter, as presented by [8], is described mathematically in Equation 2.4.1, where $y(n)$ is the output signal.

$$\boldsymbol{y(n) = \sum_{k=0}^{N-1} h(k)\, x(n-k) \quad k = 0, 1, \ldots} \tag{2.4.1}$$

Another type of digital filter is the *Infinite Impulse Response* (IIR) filter, meaning that the impulse response in theory can be infinite. IIR filters depend on not just previous

and current values of the input signal but also previous values of the output signal and are, as a result, recursive in nature [13]. Similarly to FIR filters, IIR filters can in theory be mathematically modelled according to Equation 2.4.2 below [8].

$$\boldsymbol{y(n) = \sum_{k=0}^{\infty} h(k)\, x(n-k) \quad k = 0, 1, \ldots} \tag{2.4.2}$$

In practice, however, it is not practical to compute the output using Equation (2.4.2) due to its infinite nature. Instead, the output of IIR filters can be computed recursively according to Equation 2.4.3 below [8]. Note that the output signal of the filter depends on both (previous and current) input and output signals.

$$\boldsymbol{y(n) = \sum_{k=0}^{\infty} h(k)\, x(n-k) = \sum_{k=0}^{N} a_k\, x(n-k) - \sum_{k=0}^{N} b_k\, y(n-k)} \tag{2.4.3}$$

Parameters a_k and b_k denote the filter coefficients, which are the main components for filter calculations for IIR filters, whereas $h(k)$ is of interest for FIR filter calculations [8]. It should also be noted that when b_k is set to zero in Equation 2.4.3, the equation becomes equivalent to Equation 2.4.1 for FIR filters.

FIR filters are in some DSP applications preferred over IIR filters due to their main advantage of being able to implement exactly linear phase response, due to their finite structure [14]. FIR filters are generally also stable in nature if they are realised non-recursively according to Equation 2.4.1 [8]. IIR filters are, therefore, not guaranteed to be stable. FIR filters do, however, have the disadvantage of being more computation heavy, due to having more filter coefficients required for processing [14]. Another benefit of IIR filters is that analogue filters can in many cases be directly transformed into IIR filters with similar specifications, due to their infinite nature [8].

As a conclusion, FIR filters are preferred when the number of filter coefficients are few and limiting phase shifting is important [8]. IIR filters, in turn, are typically preferred when sharp frequency cut-offs and high throughput is desired [8].

2.5 Digital Audio Formats

The process of converting an analogue signal to a digital signal, as described in Section 2.2, is known as *Pulse Code Modulation* (PCM) [15]. PCM allows for representing the signal digitally using discrete signal samples according to the sampling rate (see Figure 2.13 below). As previously discussed, the advantages of digital audio are consistency as well as opportunity for easy modification through DSP solutions.

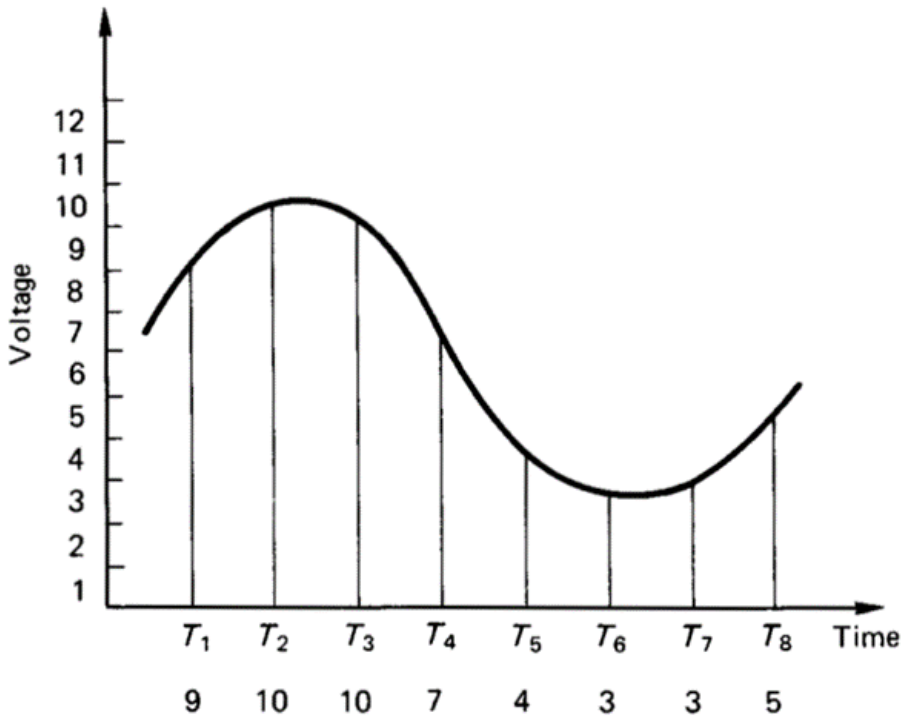

Figure 2.13: Principle of Pulse Code Modulation (PCM) visualised [15].

Due to digital audio just being numbers representing the original audio signal, it can be stored on practically any digital media with storage capabilities. One of the largest breakthroughs in digital audio history was the *Compact Disc* (CD), released in 1982 [15]. The CD is a digital optical disc which can be read with a laser by detecting change in the reflected light coming from the disc. The detected reflections can then be converted into binary 1s and 0s in order to reconstruct the audio signal digitally [15].

CDs can generally be categorised into three main categories, namely CD-ROM (Compact Disc – Read Only Memory), CD-R (Compact Disc – Recordable) and CD-RW (Compact Disc – Rewritable) [16]. CD-ROMs are discs containing pre-written data that cannot be altered. CD-Rs are discs that allow for writing once, while CD-RWs are rewritable discs [16]. CDs are written by focusing a laser on the turning disc in order to create marks in the physical discs. Depending on the disc type, different technologies are used in order to permanently or temporarily create the marks [15]. CDs support 44.1 kHz 16-bit PCM audio with no musical degradation compared to the original digital recording, since it carries the identical series of numbers as those recorded [15].

When digital audio grew even more popular with computers becoming mainstream, a huge variety of audio file formats evolved due to different performance needs, especially in computer games [17]. Over time, system manufacturers developed standard audio formats optimised for their own systems, such as:

- AIFF (Audio Interchange File Format) – Uncompressed audio data format developed by Apple for usage in MacOS-based systems. The file format contains a header with information about the number of channels, sampling rate, bits per sample etc. The related format AIFF-C (AIFC) allows for storing compressed data [17].
- WAV (Waveform Audio File Format), also known as RIFF WAVE – Uncompressed audio data format developed by Microsoft and IBM for usage in Windows-based systems. WAV also contains a header with similar information about the data as AIFF [17].

Due to the size of raw, uncompressed audio data (PCM) and limitations in storage capacity, compression has been used to shrink the audio data, with the consequence of quality reduction. An example of a common compressed audio format is MPEG (Moving Picture Experts Group), commonly associated with MP3 files. The MPEG audio format uses lossy compression by removing data points with the compression type noted in the file header [17]. The MPEG data consists of frames in sequence where each frame consists of a 32-bit header with information about the data such as sampling frequency, a 16-bit CRC check word for error detection, the layered audio data, as well as, an ancillary data field carrying optional data [17]. An example of a typical MPEG frame is illustrated in Figure 2.14 by [17] below, where (a) illustrates the whole MPEG frame, (b) illustrates the audio data in layer 1 and (c) illustrates the audio data in layer 2.

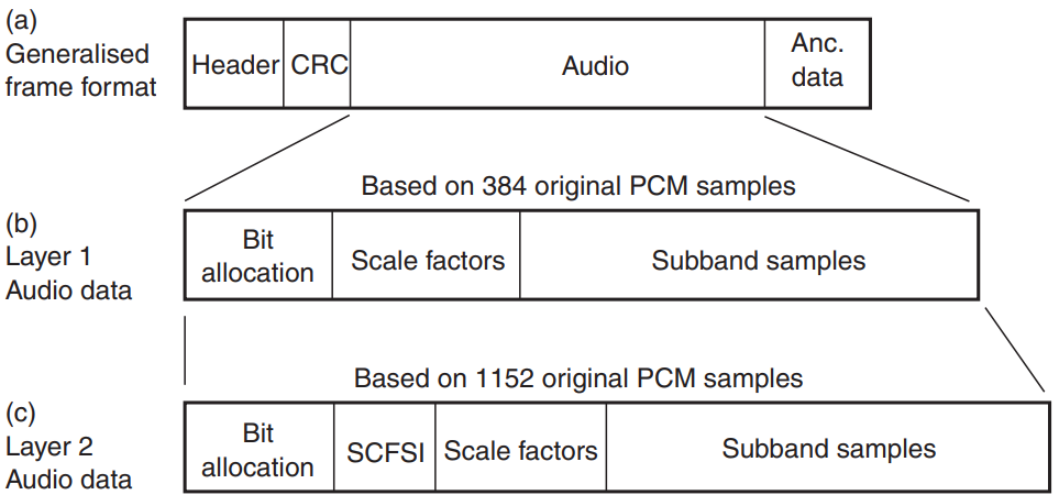

Figure 2.14: MPEG frame format [17].

2.6 Audio Quality & Data Rate

The importance of audio quality varies from application to application depending on the desired end result. For example, telephone communication requires high enough audio quality for the speech to sound natural, while at the same time maintaining low data rate. Meanwhile, high audio quality is one of the most important aspects within the music industry, which means that avoiding degradation of audio quality is essential and must be taken into consideration during audio processing.

The generally defined hearing range for humans is 20 Hz – 20 kHz, meaning that a high sampling frequency, resulting in a high data rate, is required in order to capture and reproduce all sounds within the human hearing spectrum at high quality (sampling rate of at least 40 kHz according to the Nyquist-Shannon sampling theorem) [1]. As previously mentioned, CDs support a sampling rate of 44.1 kHz with 16-bit precision per sample, resulting in a data rate of 44.1 kHz × 16 bits × 2 channels (stereo) = 1 411 200 bits/sec, or 1.411 kbits/sec. This is generally considered more than enough for the human hearing to not be able to notice any degradation in audio quality. For telecommunication, a sampling frequency of 8 kHz is commonly used, since natural sounding speech only requires a bandwidth of around 3.2 kHz [1]. Sample-precision is also generally reduced to 12 bits (or even 8 bits using a technique known as *companding* by making the quantisation levels unequal) in telecommunication systems,

resulting in minimal noticeable drop in audio quality while keeping the data rate low [1].

It is possible to reach lower levels of required data rates using compression, resulting in low data rates but poorer quality. This is typically used when very low data rates are required and a noticeable drop in quality is tolerated, for example, in certain military communication systems [1]. One way to achieve this is using *Linear Predictive Coding* (LPC) for estimating audio samples based on previous samples [1] [18]. LPC can be mathematically modelled according to Equation 2.6.1 below, where n is the model order and y denotes the predictor coefficients [18].

$$x(i) = \sum_{k=1}^{n} y_k x(i-k) \tag{2.6.1}$$

Table 2.1 by [1] sums up the relation between the required audio quality and bandwidth, sampling rate, number of bits per sample and data rate (bits/sec).

Table 2.1: Relation between audio quality, bandwidth, sampling rate, number of bits and data rate [1]

Sound Quality Required	Bandwidth	Sampling rate	Number of bits	Data rate (bits/sec)
High fidelity music (compact disc)	5 Hz to 20 kHz	44.1 kHz	16 bit	706k
Telephone quality speech	200 Hz to 3.2 kHz	8 kHz	12 bit	96k
(with companding)	200 Hz to 3.2 kHz	8 kHz	8 bit	64k
Speech encoded by Linear Predictive Coding	200 Hz to 3.2 kHz	8 kHz	12 bit	4k

3. Latency Reduction

As previously mentioned, the threshold for humans being able to distinguish between two separate audio sources is around 50 ms (the Haas effect). In practice, this means that audio played back in, for example, a stereo setting will be perceived as one source, as long as the maximum delay between the two channels is 50 ms [2]. This number, however, is derived from experiments conducted on a random sample of the general population and could in practice be lower for trained musicians [19]. The Haas effect also generally only applies to audio playback, not live performance. When producing the audio yourself in a live setting, a delay between hitting the strings on a guitar and hearing it played through a speaker introduces the element of cognitive dissonance - how the sound is perceived as feedback to physical contact with the instrument.

In an experiment presented by [19], the subjects, who were professional musicians, were presented with eight different latency levels introduced to an audio channel coming from a musical instrument played by the subject. The different latency levels were then graded by the subject on a scale from "Excellent" to "Horrible" depending on the auditory feedback received, where the grades were described in [19, p. 3] as:

- "Excellent: Artefacts are imperceptible. Delay as well as artefacts cannot be identified.
- Good: Some artefacts are perceptible, but not necessarily delay. The artefacts, though perceptible, are not annoying and do not contribute badly to the musician's performance.
- Fair: Delay and/or artefacts are perceptible. The delay and/or artefacts are slightly annoying, but in most cases would not affect the musician's performance.
- Bad: A considerable amount of delay is perceptible. The delay is annoying and is detrimental to the musician's performance.
- Horrible: A musician cannot work under these conditions."

The experiment was conducted using both floor wedges and *In-Ear Monitors* (IEMs) for playback of six different instruments (vocal, saxophone, drums, keyboard, electric bass & electric guitar). The floor wedges were placed 120 – 180 cm away from the subject, introducing an initial delay of circa 4.5 – 6.75 ms due to the speed of sound in air at room temperature.

Looking at the results and focusing on the electric guitar, as it is most relevant to this thesis, shows a perceived latency of up to only 4.5 ms using IEMs graded as “Good” with 85% confidence (see Figure 3.1 and Table 3.1 below). For floor wedges, a latency of up to 6.5 ms (plus additional latency due to placement as mentioned earlier) is perceived as “Good” with 85% confidence. For a “Fair” grade, the numbers are 14.5 ms for IEMs and 16 ms for floor wedges respectively. This is significantly lower than the Haas effect threshold of 50 ms.

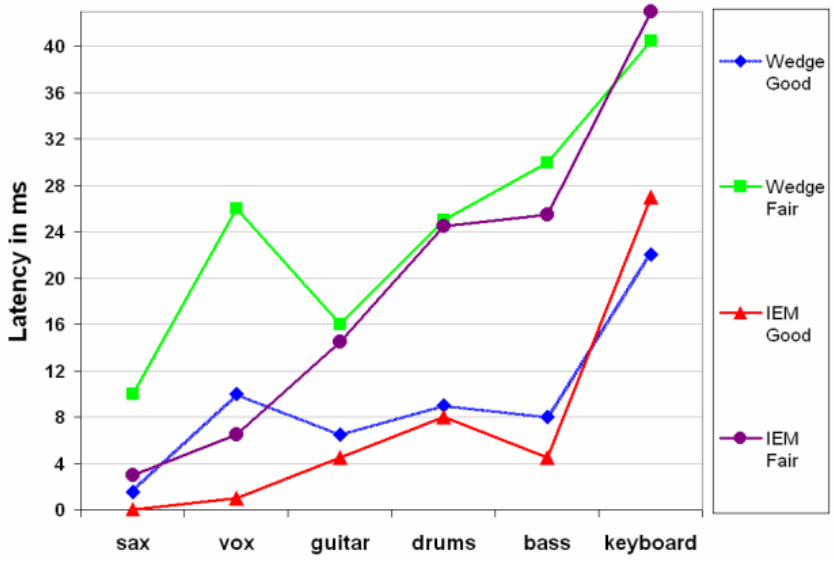

Figure 3.1: Instrument comparisons at 85% confidence level [19].

Table 3.1: Instrument comparison table at 85% confidence level [19]

Latency (ms)	Sax	Vocals	Guitar	Drums	Bass	Keys
IEM Good	0	1	4.5	8	4.5	27
Wedge Good	1.5	10	6.5	9	8	22
IEM Fair	3	6.5	14.5	24.5	25.5	43
Wedge Fair	10	26	16	25	30	40.5

As can be concluded by looking at the results of the experiment presented above, latency plays a crucial role in musical perception and playability of instruments. This is an important aspect to take into account in the experimental part further down in this thesis.

3.1 Latency of DSP Systems in General

The general DSP chain consists of a handful of steps that each contributes to the overall latency. These main steps are presented and discussed in [20]:

- Analogue-to-digital conversion
- Digital-to-analogue conversion
- Digital sample rate conversion
- Digital signal processing

An alternative method for A/D and D/A conversion to PCM in real-time systems is *delta-sigma modulation* (ΔΣ), which is an oversampling technique used where the performance of Nyquist-rate converters (e.g. PCM-based) is not sufficient, such as for higher resolution and linearity [21]. The typical ΔΣ-based converters have a latency of 0.25 – 1 ms, depending on the performance of the internal filters [20]. The digital sample rate conversion process consists of several up- and down-sampling filters, again introducing a latency of 0.5 – 10 ms, depending on filter performance [20]. Finally, digital signal processing mainly depends on processing power and clock speed for block-based processing, with typical buffer sizes ranging from 64 – 2048 samples [20]. This generally introduces a latency of 5 – 10 ms, depending on DSP performance [20].

For computer-based signal processing, such as in the experimental part of this thesis, hardware performance plays a major role in achieving satisfying results. Apart from the *central processing unit's* (CPU) clock speed, the overall CPU load may also impact the DSP performance negatively [20]. One way of overcoming this challenge is using multi-threaded solutions. The usage of proper soundcards is another way of upping processing performance, by introducing audio *application programming interfaces* (APIs) that may eliminate factors affecting system latency [20].

Apart from hardware-related factors, the *operating system* (OS) scheduling on kernel-level can also heavily affect latency, which is something that must be accounted for. According to [20], buffering incoming audio samples prior to audio processing is a method of reducing the computational load. Our implementation will rely on a similar block-based sample processing, using trial and error to determine the minimum buffer size the system can handle without affecting audio quality. The technical details of the audio processing solution for digital guitar effects in real time presented in this thesis will be discussed in more detail in Chapter 5.

Another major factor related to software is the choice of DSP algorithm. According to [20], the three main latency sources related to the DSP algorithm are:

1. Block-based processing – Audio processing using FFT blocks may introduce noticeable latency to the system. This can generally be solved by adaption of the *Short-Time Fourier Transform* (STFT) by separating time segments into smaller segments of equal length. Audio effects, such as convolutional reverberation based on impulse responses of FIR filters, are not feasible using only the FFT, due to latency. A solution for efficient convolution without input/output delay has been proposed by [22].
2. Phase delay – The signal latency of a linear response FIR filter is directly proportional to the number of filter coefficients N in accordance with the expression $d = (N - 1) \,/\, 2$, where d is the signal latency. Latency introduced by digital filters is discussed in more detail below.
3. Architecture delay – Algorithm implementation architecture is a contributing factor to overall DSP latency. A common approach in real-time processing is delaying the audio stream by the same amount as the look-ahead buffer.

Simplified block diagrams of block-based processing systems based on FFT and STFT, such as the ones mentioned above, are represented in Figures 3.2 and 3.3 below respectively.

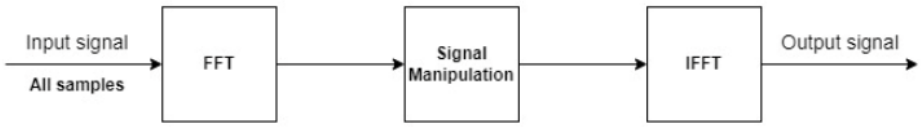

Figure 3.2: Block diagram of block-based audio processing (no time segmentation).

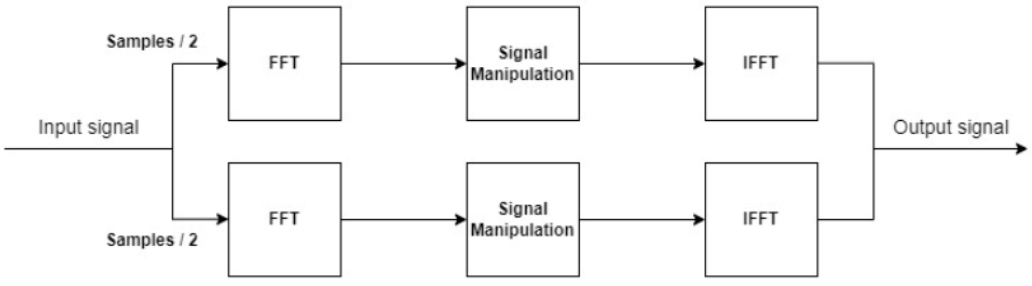

Figure 3.2: Block diagram of block-based audio processing (time segmentation).

Filter performance plays a significant role for reducing latency in the overall DSP system. Professional-grade A/D and D/A converters today are typically based on the multi-bit ΔΣ architecture, which is commonly implemented using linear phase FIR filters [20]. As mentioned above, the general signal latency d of a linear phase FIR filter can be defined as $d = (N - 1) / 2$. This is due to the fact that d is closely linked to the system *group delay* $D(\omega)$ (Equation 3.1.2), which can be derived from the *phase delay* $P(\omega)$ (Equation 3.1.1) of the linear time invariant system, where $\theta(\omega)$ denotes the phase response of the filter [23].

$$\boldsymbol{P(\omega) = -\frac{\theta(\omega)}{\omega}} \tag{3.1.1}$$

$$\boldsymbol{D(\omega) = -\frac{d}{d\omega}\theta(\omega)} \tag{3.1.2}$$

The group delay becomes constant when the phase response is a linear function of the frequency ω. This delay is the main cause of latency in the ΔΣ-based A/D and D/A conversion process [20].

For non-recursive (FIR), linear phase digital filters with order N, the transfer function $H(z)$ can be written as in Equation 3.1.3, and in its difference equation form as in Equation 3.1.4 according to [20].

$$\boldsymbol{H(z) = \prod_{i=1}^{N}(z - z_i) = z^N + h_1 z^{N-1} + h_2 z^{N-2} + \ldots + h_N} \tag{3.1.3}$$

$$\boldsymbol{y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + \ldots + b_N x(n-N)} \tag{3.1.4}$$

In Equation 3.1.4, $y(n)$ denotes the output signal and $x(n)$ the input samples. The filter coefficients b_n also function as the impulse response of the FIR filter (compare with Equation 2.4.1) [20]. Figure 3.3 by [20] below illustrates the filter components (impulse response) for a generic FIR filter.

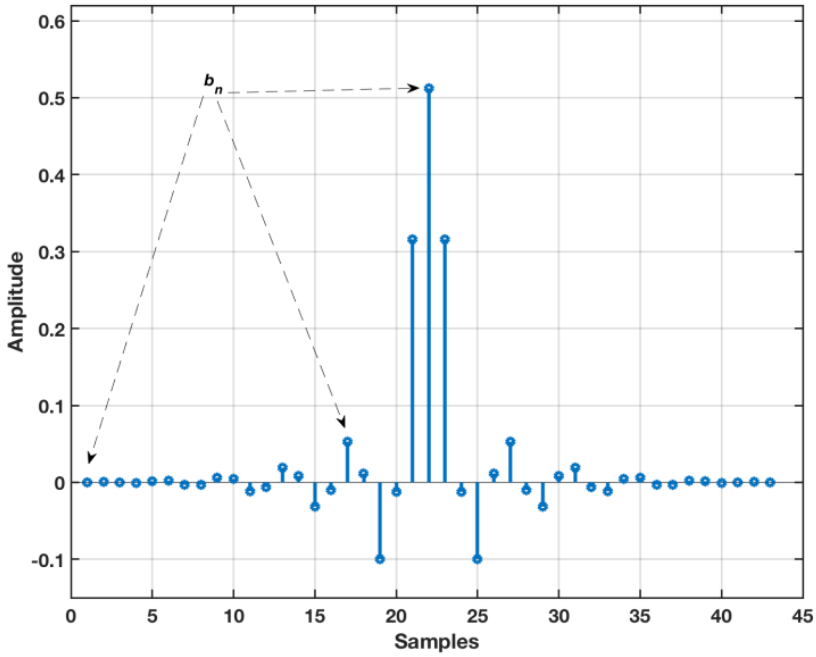

Figure 3.3: Filter coefficients (impulse response) for a generic FIR filter [20].

As can be seen from Equation 3.1.2, the group delay is a function of the phase response $\theta(\omega)$, which is defined as the phase (angle) of the frequency response and described mathematically in Equation 3.1.5 [20].

$$\boldsymbol{\theta(\omega) = \angle(H(e^{j\omega}))} \tag{3.1.5}$$

The filter's magnitude response is defined as the absolute value of the transfer function as well as the function of ω, as described in Equation 3.1.6 below [20].

$$\boldsymbol{G(\omega) = \left|H(e^{j\omega})\right| = \left|\prod_{i=1}^{N} z - z_i\right|} \tag{3.1.6}$$

Filters used in professional-grade audio conversion are typically very high in order, which results in a common practice of implementing *multi-staging* to reduce order and, thereby, required computational power. This is done by separating the filtering across different stages of sampling frequencies [24]. This process, however, does not reduce overall filter group delay [20].

3.2 Real-Time Audio Processing in Python

The Python programming language has in recent years grown to a popular choice among programmers, partly due to its short prototyping time and readability of code [25]. For real-time applications, however, compiled languages, such as C and C++, have for a long time been preferred due to efficiency [26]. Historically, the challenges with implementing real-time solutions in Python stem from slow language interpretation during execution time, which is common among interpreted languages in general [25] [26]. Due to an active community of developers, however, today numerous Python-based solutions exist for real-time audio processing, such as the audio libraries PyAudio [27] and Sounddevice [28], which provide easily accessible Python support for the multi-platform library PortAudio [29] for audio *Input/Output* (I/O) [25]. PyAudio is presented in more detail in Section 3.2.1 to give an introduction to audio processing in Python. Sounddevice will, however, be used in the experimental part of this thesis, as discussed in Section 5.2.

A solution for improving execution efficiency in Python is the Cython programming language, a compiled language based on Python for running Python code at the speed of C [25] [30]. Cython allows, for example, implementing low-level numerical loops running at C speeds, which is infeasible using traditional Python [30]. The Cython language will be utilised for improving DSP efficiency in the experimental part of this thesis and is discussed in more detail in Section 3.2.2.

3.2.1 PyAudio

PyAudio, as previously mentioned, is an audio library for Python providing bindings for the PortAudio library. PyAudio handles audio I/O streams in real time and allows for grouping audio samples at low level into chunks (buffers) for processing [25]. Buffer size is selected based on system capacity and performance requirements, and optimal values can be determined using trial and error, typically ranging from 64 to 2048 samples [26]. PyAudio supports both blocking and non-blocking read and write operations for multi-threaded solutions using asynchronous callbacks [26]. An example of an asynchronous callback for processing an audio buffer of a sample file at a given sampling rate is presented in Listing 3.1 by [26] below.

```
def callback(i_d, frame_count, t_info, f):
      data = wavefile.readframes(frame_count)
      samples = pcm2float(data)
      y = process(data)
      out = float2pcm(y)

      return (out, pyaudio.paContinue)
```

Listing 3.1: Asynchronous callback for audio buffer processing [26].

In the example given by [26] in Listing 3.1, the relevant variables are `frame_count`, `data`, `samples`, `y` and `out`, where `frame_count` represents the buffer size, `data` represents the available samples read from the file (`wavefile.readframes()`), `samples` represents the samples read, converted from the raw PCM format to float values (`pcm2float()`), `y` contains the processed data (`process()`), and `out` stores the final, processed data converted back to raw PCM data (`float2pcm()`).

The example presented in Listing 3.1 processes samples from a pre-recorded audio file, thus, does not represent real-time processing. For live audio processing, a `stream` can be opened using PyAudio, according to Listing 3.2 below, instead of using a file (`wavefile` in Listing 3.1).

```
stream = pyAudio.open(
            format = pyaudio.paInt16,
            channels = 2,
            rate = 44100,
            input = True,
            frames_per_buffer = 1024,
            input_device_index = 0
            )
```

Listing 3.2: Example of PyAudio live data stream.

The example for opening a live data stream with PyAudio given in Listing 3.2 contains parameters `format`, `channels`, `rate`, `input`, `frames_per_buffer` and `input_device_index`, where `format` is the audio data format, `channels` is the number of audio channels (1 = mono, 2 = stereo), `rate` is the sampling rate, `input` represents stream type (input stream true/false), `frames_per_buffer` represents buffer size and `input_device_index` is the index of input audio device to use.

The audio processing in the experimental part of this thesis will be based on the above-mentioned fundamentals using the audio library Sounddevice.

3.2.2 Cython

For efficiency-critical, real-time applications one of the largest challenges when using Python is execution speed. This is due to it being an interpreted language, meaning that the application bytecode is interpreted at runtime by the Python *virtual machine* (VM) and converted into machine code [26] [31]. Compiled languages such as C do not require a VM or interpreter since the C code is already converted into machine code at the compilation stage [31].

The Cython programming language is an extension of the Python language, with the purpose of integrating the speed of C in Python code [30] [32]. Cython works as a bridge between the two, where Cython code can be compiled into machine code and utilised by Python as an *extension module*, a pre-compiled module that can be run by the Python VM without interpretation needed [31]. This provides a solution for mostly keeping the high-level nature and flexibility of Python, while utilising the performance benefits of C [31]. Python code is for the most part already valid Cython code with very few exceptions, which allows for optimising already existing Python code using Cython at a later stage [32].

An example presented by [31] illustrates the similarities and differences between Python (Listing 3.3), C (Listing 3.4) and Cython (Listing 3.5) code, as well as the execution times (Table 3.2), for the different implementations of the function `fib(n)` for computing the *n*:th Fibonacci number.

```
def fib(n):
    a, b = 0.0, 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

Listing 3.3: Python implementation of fib(n) [31].

```
double fib(int n) {
    int i;
    double a = 0.0, b = 1.0, tmp;
    for (i = 0; i < n; ++i) {
        tmp = a; a = a + b; b = tmp;
    }
    return a;
}
```

Listing 3.4: C implementation of fib(n) [31].

```
def fib(int n):
    cdef int i
    cdef double a = 0.0, b = 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

Listing 3.5: Cython implementation of fib(n) [31].

As can be seen from Listings 3.3 and 3.5, the original Python implementation of `fib(n)` has been modified to implement declaration of the static C variables `i, a` and `b` (Listing 3.4). The execution times for the different implementations are summarised in Table 3.2 by [31] below.

Table 3.2: Execution time (ns) for the different implementations of fib(n) [31]

Version	fib(0) [ns]	Speedup	fib(90) [ns]	Speedup	Loop body [ns]	Speedup
Pure Python	590	1	12,852	1	12,262	1
Pure C	2	295	164	78	162	76
C extension	220	3	386	33	166	74
Cython	90	7	258	50	168	73

The “fib(0)” column measures the call overhead of the function, meaning how long purely the function call takes [31]. The “Pure Python” and “Pure C” rows contain results for running the function in Python and C respectively. The “C extension” row contains results for using an extension module for the implementation written in C and the “Cython” row contains results for the Cython-based implementation. These require conversion of Python objects to C data, computation of the Fibonacci number and conversion back to Python data. The Cython-based solution, however, has a call-overhead of about 2.5 times less than the extension-based implementation, and provides a speedup up to a factor of about 50 over the implementation based on pure Python for “fib(90)”.

In the experimental part of this thesis, the fundamental concepts of Cython will be utilised to explore the potential performance improvement when using Cython over pure Python, for the real-time implementations of guitar effects.

4. Visualisation of Common Guitar Effects

Plucking a guitar string results in a standing wave, sounding at its fundamental frequency f_1 as well as harmonic frequencies (overtones) $f_n = nf_1, n = 1, 2, 3 \ldots$ [33]. These harmonics are produced at different relative intensities across different guitars, distinguishing one instrument from another [33]. The low E string on a guitar plucked individually produces the E2 note with a fundamental frequency of 82.4 Hz alongside its harmonics E3 (164.8 Hz), B3 (247.2 Hz), E4 (329.6 Hz), G#4 (412 Hz), B4 (494.4 Hz), D5 (576.8 Hz) etc. [34]. This is visualised in Figures 4.1 – 4.3 below using the open-source, digital audio editor *Audacity* [35], when plucking the low E string (E2) of an electric guitar. The recording process chain consists of the Yamaha Pacifica 112J electric guitar (Appendix A) played through the Focusrite Scarlett 2i2 (3rd Gen) audio interface (Appendix B) and recorded with Audacity. Figure 4.1 below shows the waveform of the recorded guitar signal when the low E string is plucked, with a signal length of approximately 2.5 s.

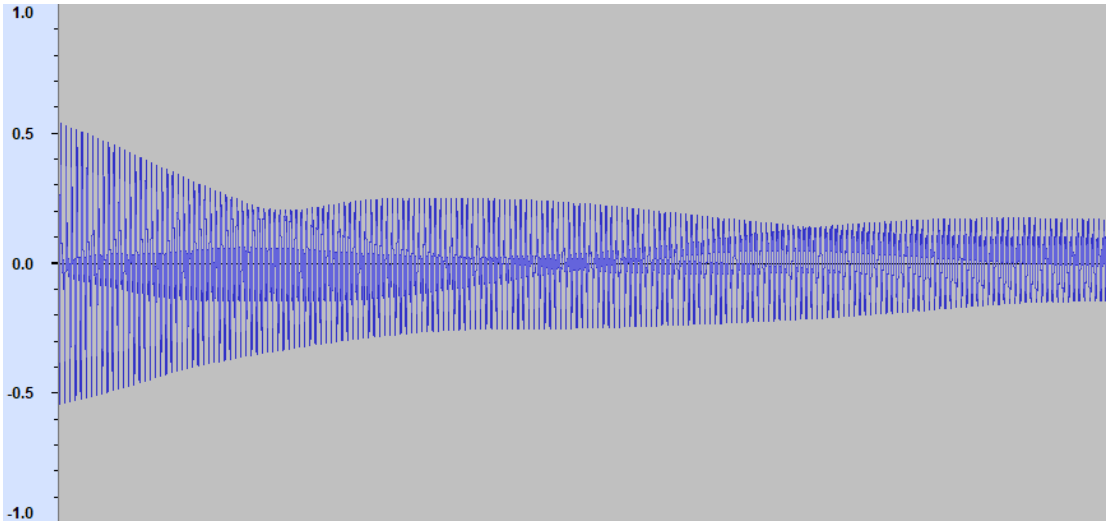

Figure 4.1: Waveform of the plucked low E string (E2) on the electric guitar.

The spectrum in Figure 4.2 below utilises Audacity's frequency analysis tool for plotting the spectrum. The selected algorithm chosen is "Spectrum" and a buffer size of 8192 samples is used, with a window function of type "Hanning".

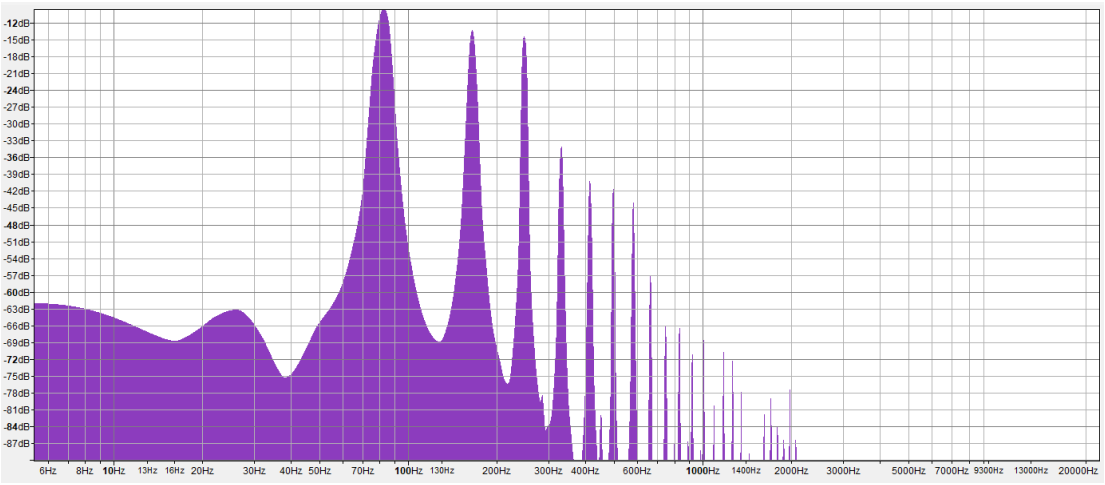

Figure 4.2: FFT of the plucked low E string.

Analysing the frequency spectrum of the signal shows the frequency magnitudes of the signal (Figure 4.2). The fundamental frequency 82.4 Hz is of largest magnitude and the harmonic frequencies are clearly visible. Similar results are seen from the spectrogram of the signal using Audacity's spectrogram tool, as seen in Figure 4.3 below, using identical settings as in Figure 4.2.

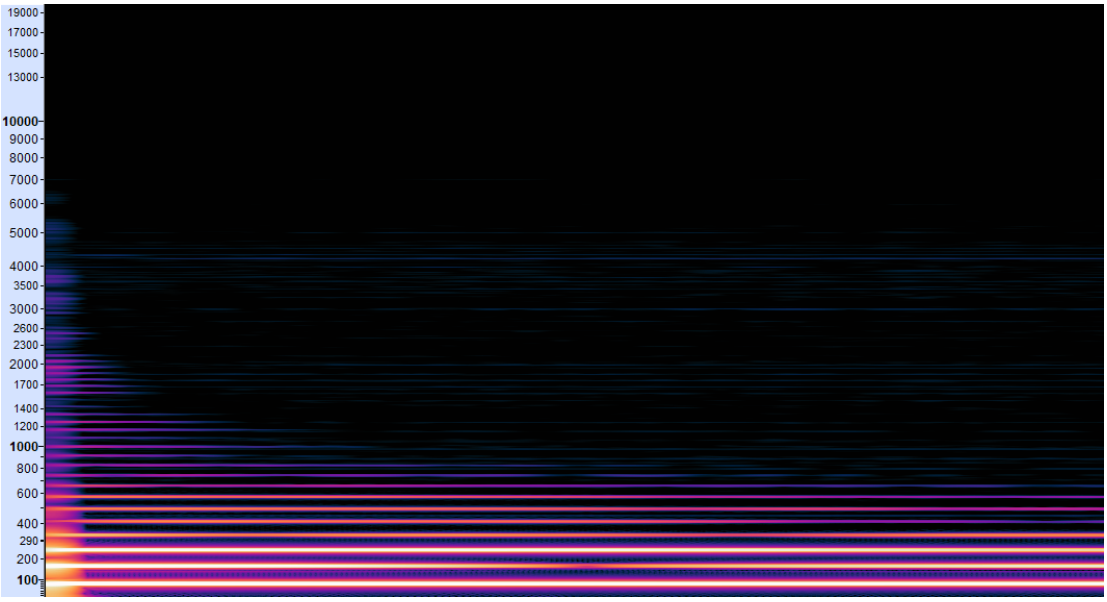

Figure 4.3: Spectrogram of the plucked low E string.

Audacity's spectrogram tool will be further utilised for visualising energy across the frequency band in Sections 4.1 – 4.5, where guitar effects are analysed for a strummed E chord. These sections introduce five popular guitar effects using visualisation of the signal waveform as well as signal spectrogram for the five common guitar effects:

- Overdrive (distortion)
- Reverberation
- Delay
- Phaser
- Wah-wah

Figure 4.4 below illustrates the approximately three-second-long signal waveform of an open E chord with clean sound (raw guitar input) and Figure 4.5 illustrates the spectrogram of the same signal. This is the reference audio used in Sections 4.1 – 4.5, for signal processing using Audacity's built-in effects.

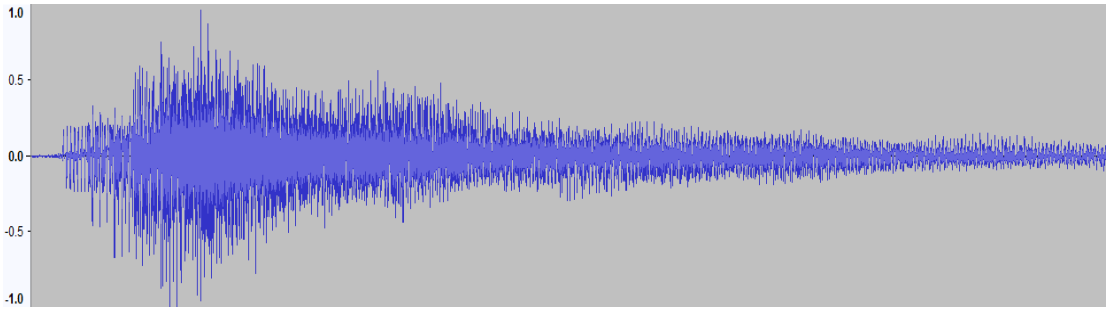

Figure 4.4: Guitar signal (clean).

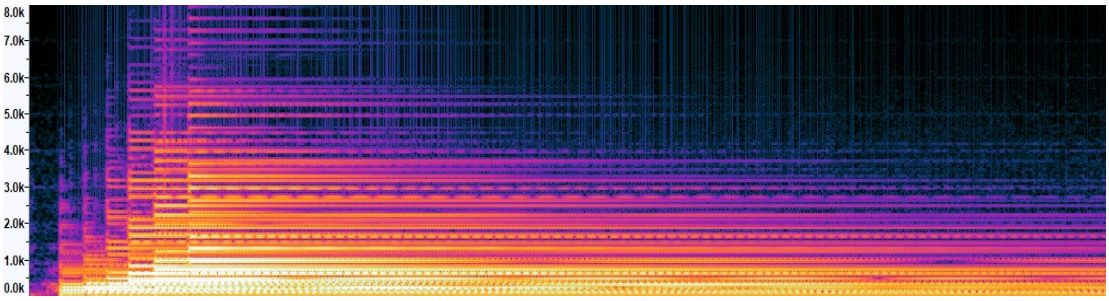

Figure 4.5: Spectrogram of guitar signal (clean).

4.1 Overdrive (Distortion)

The *Overdrive* effect is one of the most popular guitar effects used, especially in rock music. It is classified as a non-linear effect and, from a technical point of view, it can be described, according to [36, p. 117], as: "a first state where a nearly linear audio effect device at low input levels is driven by higher input levels into the non-linear region of its characteristic curve." The main cause of the distinct sound is, therefore, the non-linear part of the signal [36].

The effect is generally divided into three different types, namely *overdrive*, *distortion* and *fuzz*. *Overdrive* typically operates in the linear as well as non-linear region, resulting in a warm and smooth sound [36]. *Distortion* operates for the most part in the non-linear region reaching the upper limits, resulting in sounds all the way from warm *overdrive* to heavy, metallic sounds, typically associated with metal and grunge [36]. *Fuzz* is another type of distortion as a result of completely non-linear behaviour of the signal [36].

Figures 4.6 and 4.7 below describe the signal waveform and spectrogram respectively for the signal processed using Audacity's "Distortion" effect. As can be seen from Figure 4.6, non-linear amplification has been applied to the signal, resulting in a distorted sound and higher energy output (Figure 4.7).

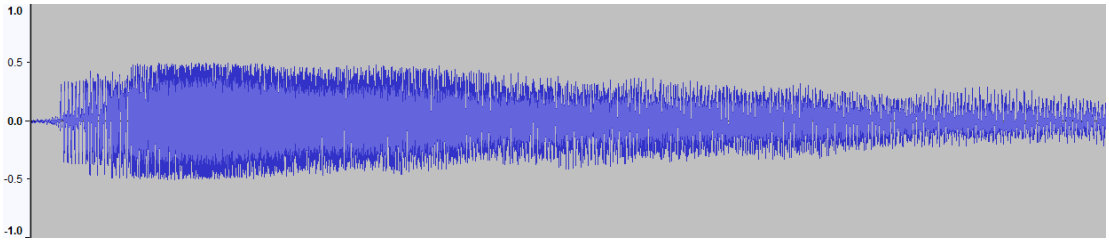

Figure 4.6: Guitar signal (overdrive).

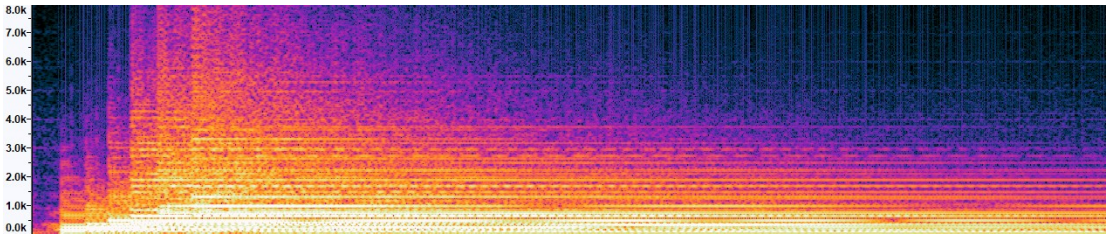

Figure 4.7: Spectrogram of guitar signal (overdrive).

4.2 Reverberation

Reverberation, or *reverb*, falls into the category of spatial effects, which refers to modification of the audio signal localisation cues [37]. Audio signals are typically affected by the environment, as they travel to the listener from the source, reflecting off various objects on the way in various directions, and the resulting effect is generally referred to as reverberation [36]. Reverberation is affected by the size and shape of the surrounding environment as well as the objects within it that interact with the audio signal [36].

Artificial reverberation has existed since the 1960s and was originally initiated by Manfred Schroeder, inventor of the *Schroeder Reverberator* [38]. This reverberator consists of a series of connected all-pass filters, parallel feedback comb filters (IIR) as well as a mixing matrix [38]. The signal waveform and spectrogram, illustrated in Figures 4.8 and 4.9 below, show the results of Audacity's "Reverb" effect, which is based on the "Freeverb" algorithm by "Jezar at Dreampoint", based on the Schroeder Reverberator. The algorithm uses eight parallel feedback comb filters followed by four all-pass filters in series [38]. As can be seen from Figure 4.9, the signal energy contains more variation across the audio spectrum due to the artificial reverberation effect. This results in a larger sound perceived compared to the original clean sound.

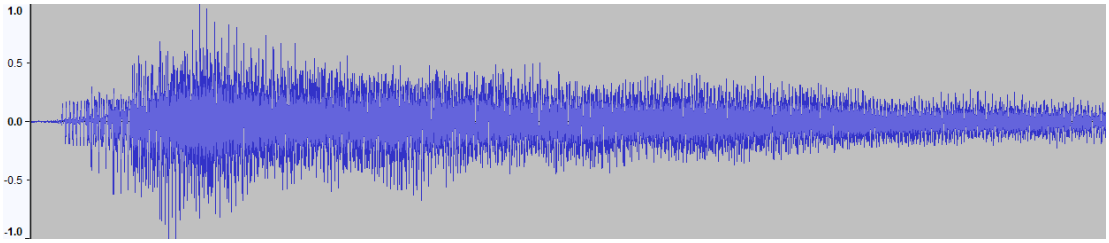

Figure 4.8: Guitar signal (reverb).

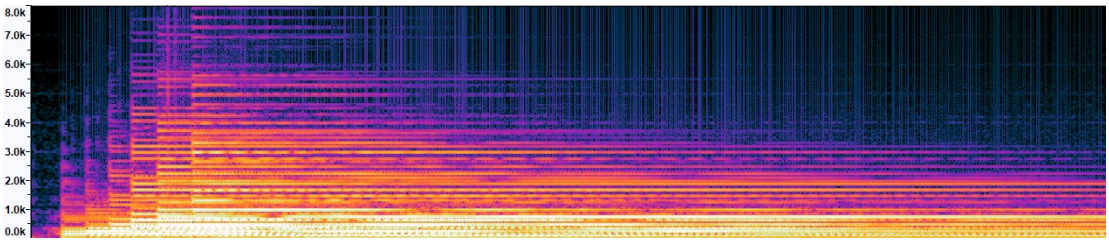

Figure 4.9: Spectrogram of guitar signal (reverb).

4.3 Delay

Delay, or *echo*, is the effect of audio waves echoing for an extended period of time. This happens naturally similarly to the reverberation effect, where sound waves are reflected back to the source with time delay. Over large distances this results in an audible echo. Delay pedals typically offer the ability to tune delay time (how often the sound is echoed) and duration (how long the echoing lasts).

Since reverberation is a form of delay, the implementation of echo can be done using comb filters [36]. FIR comb filters work by adding a time-delayed signal back to the input signal while IIR comb filters implement a feedback loop, feeding the delayed signal back to the input signal [36]. The two tuning parameters typically used are for tuning time delay as well as relative amplitude of the delayed signal to control how fast the echo fades out [36].

Figures 4.10 and 4.11 below illustrate the waveform and spectrogram of a signal processed with Audacity's "Echo" effect, using a time delay of 0.6 seconds and delay factor of 0.5, reducing the amplitude by half each time. In Figure 4.11, it is clearly visible when the audio is echoed at equal intervals.

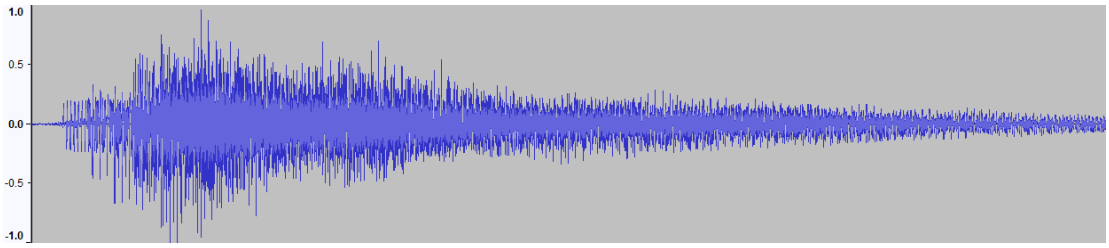

Figure 4.10: Guitar signal (delay).

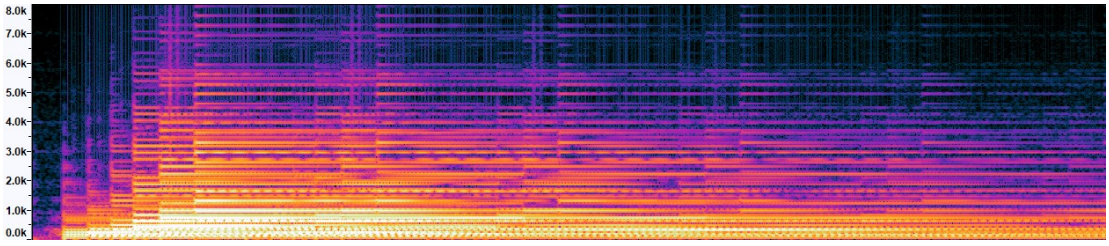

Figure 4.11: Spectrogram of guitar signal (delay).

4.4 Phaser

The *phaser* effect is an effect of time-varying filters and is typically realised using notch filters, a type of band-stop filter that filters out select frequencies [13] [36]. Typically, the input signal is processed by a set of notch filters and then combined with the unprocessed signal, causing phase cancellations and enhancements that result in a clearly audible effect known as *phasing* [36]. The phaser signal chain is illustrated in Figure 4.12 by [36] below.

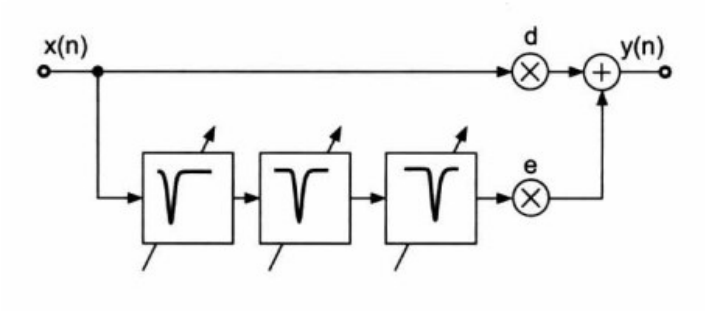

Figure 4.12: Phaser signal chain [36].

Figures 4.13 and 4.14 below illustrate the signal waveform and spectrogram for a signal processed using Audacity's "Phaser" effect. The phase cancellation is clearly visible in the spectrogram of the signal, illustrated in Figure 4.14.

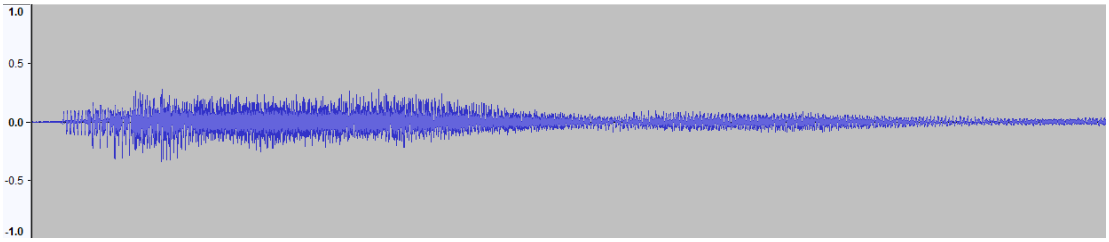

Figure 4.13: Guitar signal (phaser).

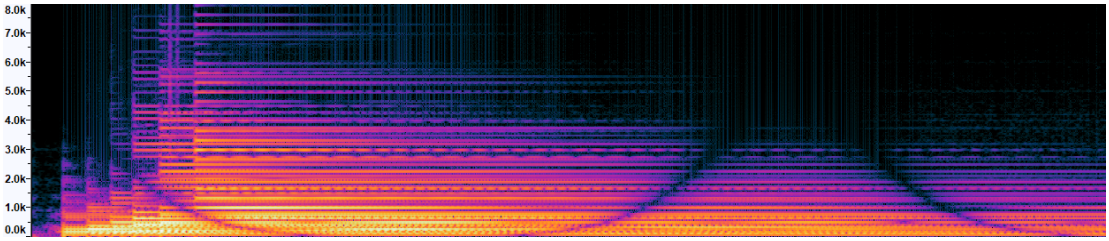

Figure 4.14: Spectrogram of guitar signal (phaser).

4.5 Wah-wah

Another effect based on time-varying filters is the *wah-wah* effect. This effect is the result of a low-bandwidth band-pass filter with variable cut-off frequency [36]. Wah-wah pedals allow the player to shift the cut-off frequency of the filter on the input signal up and down using the foot [36]. The filtered signal is finally added to the input signal, as illustrated in Figure 4.15 below, resulting in the typical wah-wah sound.

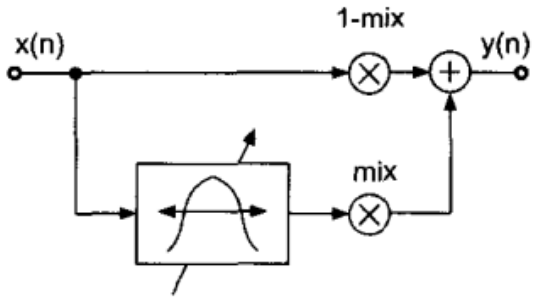

Figure 4.15: Wah-wah signal chain [36].

Figures 4.16 and 4.17 below illustrate the signal waveform and spectrogram for a signal processed using Audacity's "Wah-wah" effect. The effect on the frequency spectrum is clearly visible in Figure 4.17.

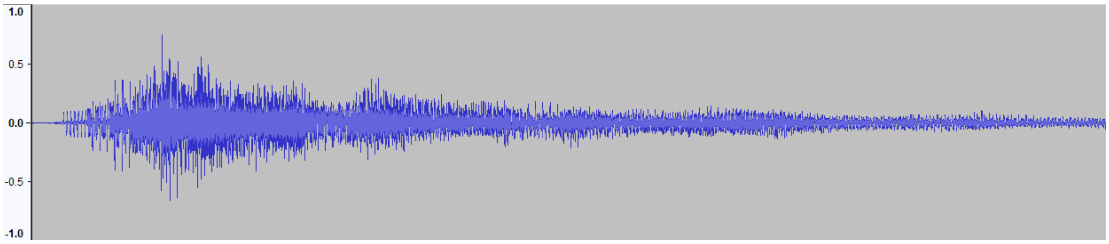

Figure 4.16: Guitar signal (wah-wah).

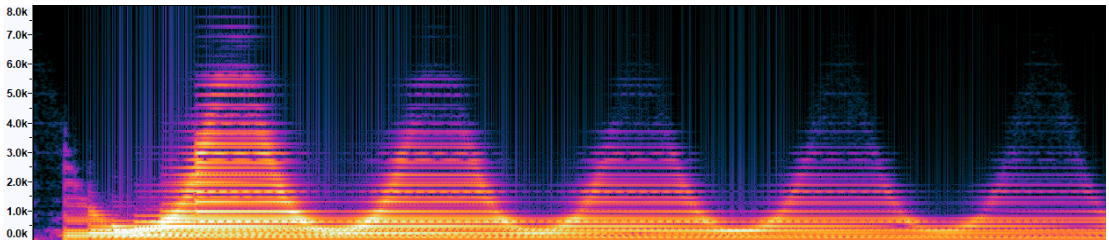

Figure 4.17: Spectrogram of guitar signal (wah-wah).

5. Python Experimentation

This chapter contains the experimental part of this thesis. The main focus is implementing low-latency guitar effects in real time, using the concepts presented above, with the goal of answering the three following questions:

1. How can common guitar effects such as overdrive, reverberation and delay be implemented digitally using Python?
2. How can signal latency be reduced during the DSP process using Cython?
3. How can a guitar effect be implemented digitally that would not be feasible using traditional, analogue methods?

The first question aims to implement the already existing guitar effects overdrive, reverberation and delay using Python, in order to give a foundation for the rest of the experiment to build upon. The goal is to overcome any challenges related to the digital implementation of the effects at an early stage. Python offers numerous libraries for signal processing, such as NumPy [39] and Librosa [40], which both will be used for audio processing.

One of the most essential parts of real-time signal processing is minimising signal latency, as discussed in Chapter 3. The aim is to integrate Cython to optimise the sections responsible for DSP computations to address this challenge. The overall latency for the processed signal is measured using both the Focusrite Control [41] software, as well as using Python's Time library, both before and after processing and optimisation.

The final goal of the experiment is to implement a signal *harmoniser* that lowers or raises the signal frequency in octaves. Frequency manipulation has proven to be difficult using traditional, analogue methods but can effortlessly be done digitally. Digital solutions also offer more versatility than their analogue counterparts and allow for continuous modification and tuning.

The gear and libraries used, as well as their parts in the system chain, are listed and described in Section 5.1. The underlying OS on the computer used in the experiment is Microsoft's Windows 11 [42].

5.1 Signal Chain

The signal chain of the experiment consists of both hardware and software. The main hardware used in the experiment include:

- Yamaha Pacifica 112J electric guitar (Appendix A)
- Focusrite Scarlett 2i2 audio interface (Appendix B)
- AMD Ryzen 7 7800X3D CPU (Appendix C)

The electric guitar features an HSS (humbucker, single coil & single coil) pickup arrangement, with a combination of the neck and middle pickups used for the recordings in the experiment. The signal is amplified and run through an A/D converter in the audio interface and forwarded to the CPU.

The software used in the experiment (including Python libraries) include:

- Python (Sounddevice, Time, NumPy & Librosa)
- Cython
- Focusrite Control
- Audacity

Python is the main programming language used for implementation, with use of Cython to optimise DSP computations. Audacity is mainly used for spectrum analysis as well as spectrogram visualisation of the processed signal. The Focusrite Control software is used to measure signal latency, in addition to Python's Time library. The hardware and software system chains are visualised in Figures 5.1 and 5.2 below respectively.

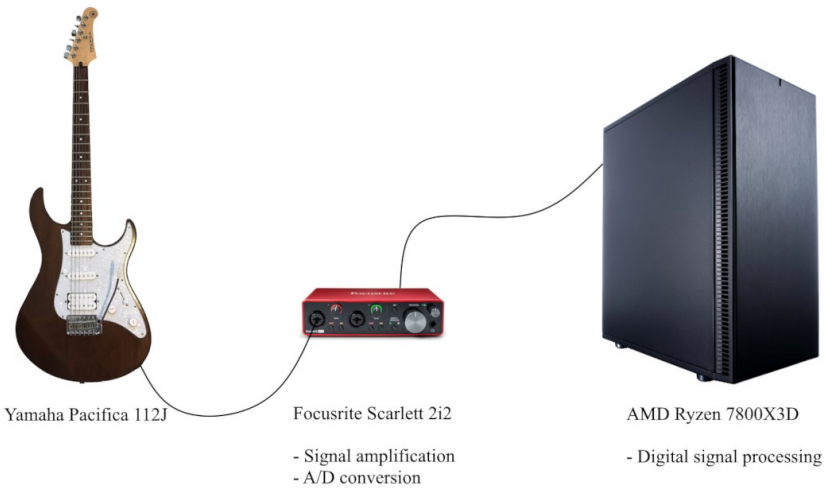

Figure 5.1: Hardware signal chain used in the experiment [43] [44].

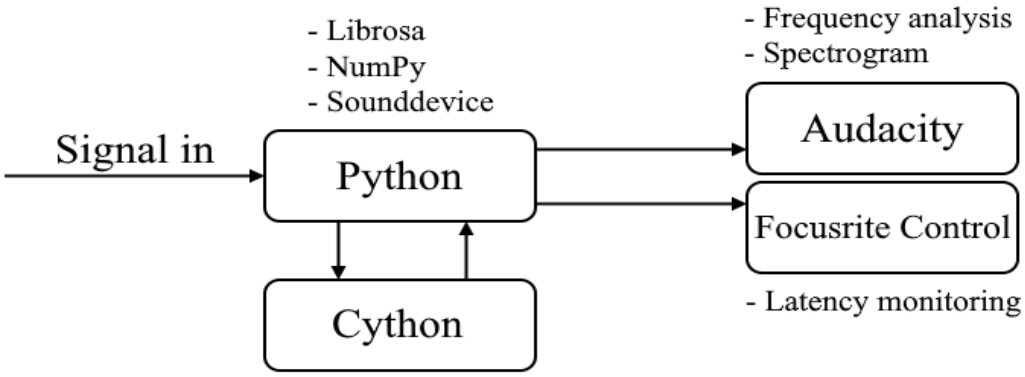

Figure 5.2: Software signal chain used in the experiment.

5.2 Audio Analysis in Python

The Sounddevice audio library provides bindings to the PortAudio library much similarly to PyAudio, as discussed in Section 3.2.1. The reason Sounddevice is chosen over PyAudio is mainly due to the better support for live audio capture using Sounddevice, whereas PyAudio is better suited for reading audio files. Setting up a continuous I/O audio stream in Python using Sounddevice in its most basic form can be done as shown in Listing 5.1, using a modified version of the example code given by [28]:

```
def callback(indata, outdata, frames, time, status):
    if status:
        print(status)

    outdata[:] = indata


sd.Stream(device=(input_device, output_device),
          samplerate=samplerate,
          blocksize=blocksize,
          latency=latency,
          channels=channels,
          callback=callback)
```

Listing 5.1: Required Python code for opening a continuous I/O stream using the Sounddevice audio library [28].

Sounddevice's *Stream* class opens a PortAudio stream for simultaneous input and output using NumPy arrays [28]. The parameters required are `device, samplerate, blocksize, latency` and `channels,` where `device` specifies which input and output device to use (by index), `samplerate` specifies the signal sampling rate, `blocksize` specifies the block (buffer) size of the stream, `latency` specifies the desired latency between input and output signal in seconds, and `channels` specifies the number of channels to use (mono/stereo). The Stream class also takes a function `callback` as input parameter for consuming the stream. This is where the signal processing is applied on the continuous stream.

Apart from minimising latency introduced by the signal processing, the focus in Section 5.4, eliminating other sources of latency at this stage is key. This mainly includes reducing system-induced latency stemming from the OS and audio APIs. As shown by [45], the usage of the *Audio Stream I/O* (ASIO) API [46] by Steinberg can significantly reduce latency, especially on Windows-based operating systems. The ASIO protocol enables communication directly between the hardware and the software in question, bypassing OS-specific audio APIs, which in turn can significantly reduce latency [46]. ASIO is supported by both the Focusrite Scarlett 2i2 audio interface and the Sounddevice audio library out of the box and is utilised in the experiment for low-latency audio I/O.

Signal latency can further be reduced by choosing a high enough sampling rate along with a small enough buffer size. The Focusrite Scarlett 2i2 audio interface supports sampling rates between 44.1 kHz and 192 kHz, with buffer sizes between 16 and 1024 samples. Testing different combinations of sampling rate and buffer size can be done to determine system capability and performance, with the goal being as high as possible sampling rate, combined with as low as possible buffer size for minimal latency.

The latency test is performed by opening the stream using different combinations of sampling rates and buffer sizes. The latency reported by Focusrite Control is the round-trip latency for the signal, meaning the time taken between input and output of the signal [47]. The sampling rates used are 48 kHz, 96 kHz, 176.4 kHz and 192 kHz, with buffer sizes of 32, 64, 128, 256 and 512 samples. Figure 5.3 below shows the Focusrite Control software, with latency information listed at the bottom. The results of the latency measurement are presented in Table 5.1 below, highlighting the latency measured at each sampling rate and buffer size.

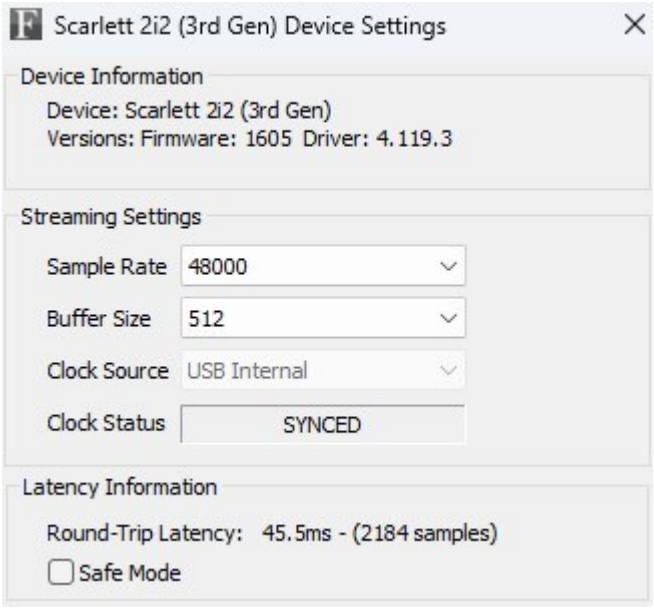

Figure 5.3: Focusrite Control user interface.

Table 5.1: Latency (ms) using different combinations of sampling rate and buffer size

Sample Rate (Hz)	Buffer Size (samples)	Latency (ms)
48 000	32	4.5
48 000	64	6.9
48 000	128	13.5
48 000	256	24.9
48 000	512	45.5
96 000	32	3.4
96 000	64	4.0
96 000	128	6.4
96 000	256	13.0
96 000	512	24.4
176 400	32	3.0
176 400	64	3.3
176 400	128	4.1
176 400	256	6.5
176 400	512	13.4
192 000	32	2.9
192 000	64	3.2
192 000	128	3.9
192 000	256	6.2
192 000	512	12.9

None of the tests performed above, with results listed in Table 5.1, showed signs of negatively impacting audio quality, such as introducing crackling or distortion, meaning that the CPU and audio interface seemingly can keep up with all sampling rates and buffer sizes included in the tests. For this reason, a balanced solution with low enough latency and high enough buffer size is a sampling rate of 96 kHz with a buffer size of 128 samples. This combination is used for the DSP in Section 5.3 below.

5.3 Implementation of Guitar Effects

Using the code highlighted in Listing 5.2 below, a continuous audio I/O stream can be opened for signal processing. The parameters used are listed below:

- Sampling rate: 96000 Hz
- Buffer size: 128 samples
- Channels: 2 (stereo)
- Latency: 0 seconds (Sounddevice internal latency between I/O)

```
Sounddevice.Stream(device=(self.input_device, self.output_device),
                   samplerate=self.rate,
                   blocksize=self.block,
                   latency=self.latency,
                   channels=self.channels,
                   callback=self.callback,
                   dtype=numpy.float32)
```

Listing 5.2: Python code used for opening continuous I/O stream.

A general overdrive effect is implemented in Section 5.3.1 and a combined reverberation and delay effect is implemented in Section 5.3.2. The attempt of implementing a signal harmoniser is described in Section 5.3.3. Time and latency measurements are not the focus of this section and are instead presented in Section 5.4. The main focus of this thesis is not the implementation of the guitar effects themselves but rather low-latency, real-time audio processing and streaming using Python. The effects presented below are not full-fledged effects but rather implemented in their simplest form, without proper filtering that would normally be used before and after processing the input signal.

5.3.1 Overdrive

The audible overdrive effect, as mentioned in Section 4.1, is the result of non-linear behaviour of the audio signal. A common approach for emulating the analogue overdrive effect digitally is using the non-linear, trigonometric function hyperbolic tangent $tanh$ [48]. Using a *gain* variable for input signal amplification and *threshold* variable for limiting the upper- and lower signal values (clipping), a generic overdrive effect can be implemented as described in Listing 5.3 below.

```
def overdrive(self, input_signal, gain, threshold):
    output_signal = numpy.tanh(input_signal * gain) * threshold

    return output_signal
```

Listing 5.3: Python implementation of the overdrive effect.

The implemented overdrive effect works by amplifying the signal according to the provided *gain* and returns the values of the hyperbolic tangent of the input signal values, with clipping at the provided *threshold*. The waveform of the processed signal, using a gain parameter of 10 with a threshold value of 0.5, is visualised in Figure 5.4 below for a three-second-long strummed E chord. The spectrogram for the processed signal is illustrated in Figure 5.5. Reference audio is presented in Appendix D.

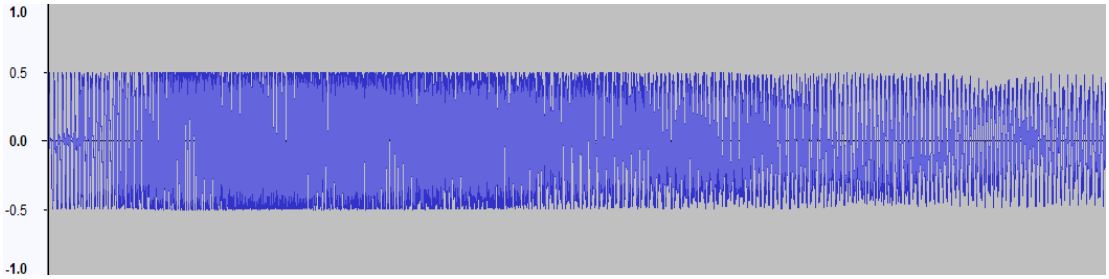

Figure 5.4: Waveform of signal with implemented overdrive effect.

Figure 5.4 clearly shows that the signal has been amplified and clipped at a level of 0.5, resulting in the desired overdrive sound. Similar signal behaviour was previously observed in Figure 4.6.

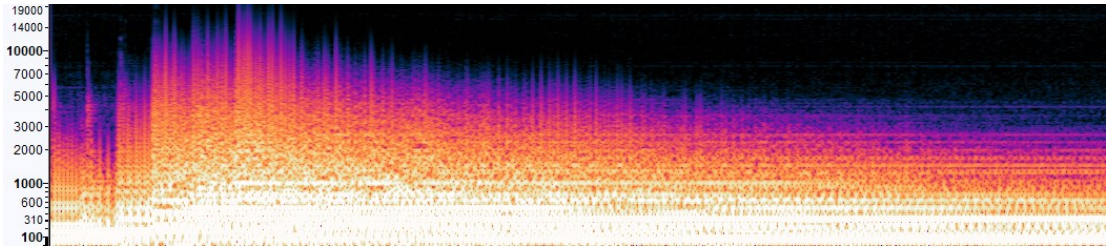

Figure 5.5: Spectrogram of signal with implemented overdrive effect.

Comparing the spectrograms visualised in Figure 5.5 and Figure 4.7, similar behaviour can be observed, where the signal energy is significantly larger compared to the spectrogram for the clean signal visualised in Figure 4.5.

5.3.2 Reverberation and Echo

As previously discussed in Section 4.3, the reverberation and delay (echo) effects can be realised using FIR comb filters, which work by adding the time-delayed signal back to the input signal. Due to a small buffer size of 128 samples and a sampling rate of 96000 Hz, the buffer can only store the last, approximately, 1.3 ms of audio data. This can be overcome using a circular buffer of length *c_buffer_max* that is continuously updated with audio samples to store for further processing [49]. The implementation of the echo effect is described in Listing 5.4 below. The circular buffer's length determines the length of the repeated echo.

```
def reverb_and_echo(self, input_signal, delay, decay):
    delay_samples = int(delay * self.rate)
    num_samples = len(input_signal)
    output_signal = numpy.zeros_like(input_signal)
    i = 0

    while i < num_samples:
        delayed_index = (self.buffer_index - delay_samples)
                        % self.c_buffer_max
        output_signal[i] = input_signal[i] + self.c_buffer[delayed_index]
        self.c_buffer[self.buffer_index] = input_signal[i] + decay
                                        * self.c_buffer[self.buffer_index]
        self.buffer_index = (self.buffer_index + 1) % self.c_buffer_max
        i += 1

    return output_signal
```

Listing 5.4: Python implementation of the combined reverberation and echo effect.

The implementation echoes the input signal with a given delay in seconds as well as amplitude decay between 0 and 1. The spectrogram for the processed signal, using a delay of 0.5 seconds with decay 0.5, is presented in Figure 5.6 below for a three-second-long strummed E chord. Reference audio is presented in Appendix D.

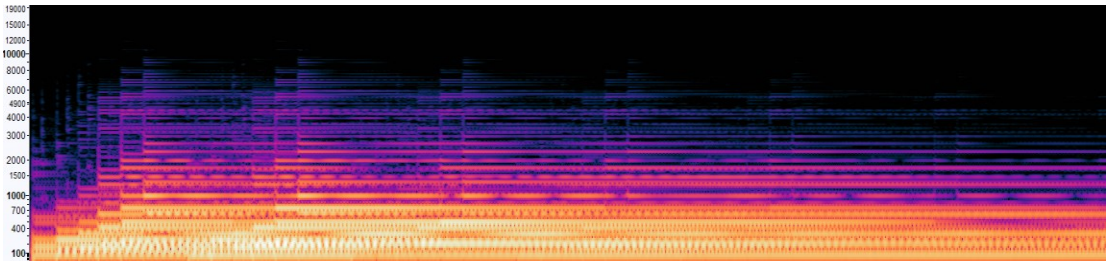

Figure 5.6: Spectrogram of signal with implemented reverberation and echo effect.

The spectrogram for the implemented echo effect in Figure 5.6 clearly shows the echo effect repeating the signal at equal intervals of the given delay of 0.5 seconds.

5.3.3 Harmoniser

The final guitar effect to be implemented is the harmoniser, an effect based on pitch shifting. Pitch shifting is the procedure of changing the signal pitch while keeping the original signal length [50]. The reason this effect is chosen is that real-time pitch shifting for the electric guitar has proven to be difficult using analogue methods [50]. Digital solutions, however, can be implemented effortlessly, as will be highlighted below.

For this experiment, the pitch-shifting technique will be used to increase the signal pitch one octave while also keeping the original signal untouched. The original guitar signal in mono will then be assigned to one channel of the output signal while the pitch-shifted signal will be assigned to the other channel, resulting in the fundamental frequency as well as the first overtone sounding simultaneously in stereo. It should be noted, however, that the goal of the experiment is to showcase a working, but basic, implementation of the harmoniser effect using pitch shifting in Python and not an entirely acoustically pleasing ensemble.

The pitch shifting in this experiment is inspired by [51] and implemented using time stretching as well as time-domain-based resampling. Time stretching is the operation of changing signal duration while keeping the original pitch, and is, therefore, essentially the opposite of pitch shifting [50]. Since the focus of the experiment is not on the implementation itself but rather a real-time solution for a harmoniser effect, the DSP algorithms utilised are implemented by the Python library Librosa.

Pitch shifting the signal up one octave requires resampling the signal at half the original sampling rate f_s / 2 [51]. The original buffer size is kept intact if the signal is first time-stretched to double its length. For these operations, Librosa offers the functions `time_stretch(y, rate)` (where `y` is the input array to be stretched and `rate` is the stretch factor, which indicates if the signal should be slowed down or sped up) and `resample(y, orig_sr, target_sr)` (where `y` is the input array to be resampled, `orig_sr` is the original sampling rate and `target_sr` is the target sampling rate). For the algorithms to work properly, the input signal buffer size must be increased to 512 samples, resulting in a round-trip latency, as reported by Focusrite Control, of 24.4 ms. The implementation for the pitch-shifting algorithm is described in Listing 5.5 below, utilising the time-stretch implementation described in Listing 5.6.

```
def pitch_shift(self, input_signal, shift):
    target_sampling_rate = self.rate * shift
    stretched_signal = self.time_stretch(input_signal, shift)
    resampled_signal = librosa.resample(
                            stretched_signal,
                            orig_sr=self.rate,
                            target_sr=target_sampling_rate)

    return resampled_signal
```

Listing 5.5: Python implementation of the pitch shifting function.

```
def time_stretch(self, input_signal, stretch_factor):
    output_signal = librosa.effects.time_stretch(
                            input_signal,
                            rate=stretch_factor)

    return output_signal
```

Listing 5.6: Python implementation of the time stretching function.

This implementation produces a pitch-shifted output signal one octave higher than the input signal. However, as also discovered in [51], the resampling process introduces data loss, which results in uneven transitions between audio segments at equal intervals the size of the buffer. These discontinuities are visualised in Figure 5.6 and result in audible clicks.

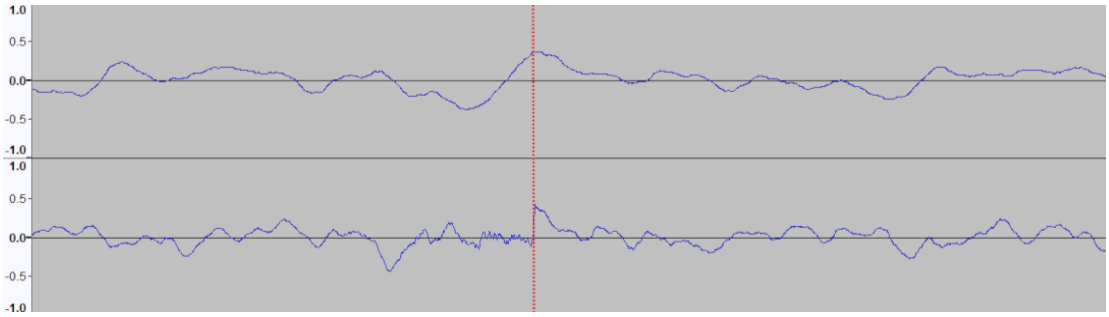

Figure 5.7: Output signal discontinuity in lower (right) channel marked with a red line.

The artefacts can be partially eliminated using the same approach as utilised in [51], known as crossfading. This approach involves scaling the resampling of the signal up by a factor of two once more before creating two signals phase-shifted by 180 degrees, as shown in Listing 5.7 [51]. The next step is modulating the signal by a triangular distribution, as also shown in Listing 5.7, before finally combining the signals [51]. Crossfading improves the audio quality substantially but the crossfade modulation is still obvious in the output signal, as can be heard from the reference audio presented in Appendix D.

```
# Create two signals phase-shifted by 180 degrees
signal1 = resampled_signal[::2]
signal2 = resampled_signal[1::2]

# Modulate amplitude using triangular distribution
fade_in = numpy.linspace(0, 1, len(signal1) // 2)
fade_out = numpy.linspace(1, 0, len(signal1) // 2)
fade = numpy.concatenate((fade_in, fade_out))

signal1 *= fade
signal2 *= fade[::-1]  # Reverse fade for second signal

output_signal = signal1 + signal2
```

Listing 5.7: Python code for crossfading (inside the pitch_shift() function).

5.4 Latency Reduction Using Cython

Due to Python's interpreted nature, loops are where the largest performance gains can be seen when utilising Cython [30]. For the implementation of guitar effects presented in Sections 5.3.1 – 5.3.3, this means that the largest optimisations can be achieved for the combined reverberation and echo effect, as it is the only effect implemented using loops. The other effects rely on library-specific implementations of the algorithms, such as NumPy's *tanh()* and Librosa's *resample()*. This approach was chosen, since the main focus of this thesis is not on the actual implementation of the guitar effects in itself but rather on the concept of real-time audio processing in Python as a whole.

Optimising the implementation of the combined reverberation and echo effect algorithm, described in Listing 5.4, using Cython mainly requires setting data types for the variables. This can be done using the custom Cython syntax *cdef*. Setting data types significantly reduces execution time by eliminating the Python overhead otherwise required for wrapping Python objects during runtime [52]. The Cython-optimised implementation of Listing 5.4 is given in Listing 5.8 below.

```
DYTPE = np.float32
ctypedef np.float32_t DTYPE_t

cpdef np.ndarray[DTYPE_t, ndim=2] echo(np.ndarray[DTYPE_t, ndim=2]
input_signal, double delay, double decay, int rate,
np.ndarray[DTYPE_t, ndim=2] c_buffer, int buffer_index,
int c_buffer_max):

    cdef int delay_samples = int(delay * rate)
    cdef int num_samples = input_signal.shape[0]
    cdef int i, delayed_index
    cdef np.ndarray[DTYPE_t, ndim=2] output_signal

    output_signal = np.zeros_like(input_signal)

    while i < num_samples:
        delayed_index = (buffer_index - delay_samples) %
                         c_buffer_max
        output_signal[i] = input_signal[i] +
                           c_buffer[delayed_index]
        c_buffer[buffer_index] = input_signal[i] + decay *
                                 c_buffer[buffer_index]
        buffer_index = (buffer_index + 1) % c_buffer_max
        i += 1

    return output_signal
```

Listing 5.8: Cython-optimised implementation of Listing 5.4.

Execution time measurements are done using the function *perf_counter()* from Python's Time library. This method returns the system-wide time taken (in seconds) to perform the execution between two points in the code [53]. Execution time is measured for both the combined reverberation and echo effect algorithm (Listing 5.4) and the Cython-optimised algorithm (Listing 5.8), using different buffer sizes to compare execution time over loops with different numbers of iterations. The mean execution time (MET) over 500 method calls, for both the non-optimised Python implementation of the algorithm and the optimised implementation, is given in Table 5.2 below, for varying buffer sizes between 128 and 1024 samples.

Table 5.2: Execution time for the Python and Cython implementations of the combined reverberation and echo effect algorithm using different buffer sizes

Buffer Size (Samples)	Python MET (ms)	Cython MET (ms)
128	0.382435	0.006452
256	0.758102	0.007027
512	1.469831	0.007394
1024	2.858569	0.007854

As can be seen from the results, the Cython implementation is significantly faster than its counterpart implemented in Python. The real-world impact of this speedup using the Cython-optimised algorithm is further discussed in Chapter 6.

6. Results

This chapter presents the subjective and objective results of the experiment performed in Chapter 5. The results of the experiment can be evaluated separately for the guitar effect implementations in Section 5.3 and for the latency reduction in Section 5.4. Section 6.1 below discusses the guitar effect implementations from both a subjective and objective point of view, as it is difficult to evaluate the effects implemented strictly objectively. The result of the latency reduction is presented in Section 6.2.

6.1 Evaluation of Guitar Effect Implementations

One focus of the thesis was implementing the three common guitar effects overdrive, reverberation and delay in Python using real-time signal processing. This was achieved using audio libraries Sounddevice, NumPy and Librosa, since the focus was not strictly limited to the actual implementations but rather the concept of implementing real-time effects as a whole. The reverberation and delay effects were implemented as a single algorithm due to their similar nature, as discussed in Sections 4.2 and 4.3.

The implementations, as illustrated in Listings 5.3 and 5.4, were tested on the AMD Ryzen 7 7800X3D CPU (Appendix C) with a sampling rate of 96 kHz and a buffer size of 128 samples for minimum latency, using the Focusrite Scarlett 2i2 audio interface (Appendix B). This resulted in an initial latency, as reported by Focusrite Control, of 6.4 ms.

Using the Time library from Python, the mean execution time (MET) for each effect was monitored (Table 6.1). For the overdrive effect implementation, the MET over 500 method calls was reported as approximately 0.000016 seconds, or 0.016 ms. This delay is so insignificant that it can be omitted from the results. Similarly, for the combined reverberation and delay effect, a MET over 500 method calls of approximately 0.000382 seconds, or 0.382 ms, was reported. This adds up to a total delay of 6.78 ms, without taking other factors such as OS-level scheduling and other processing delays into account. This delay is well below the Haas threshold and would most likely be graded somewhere between “Good” and “Fair” in a similar test as the one performed in [19], as discussed in Chapter 3. As this effect is loop-based it was later the target for latency reduction, as discussed in Section 5.4.

Another focus of the thesis was implementing a harmoniser effect, due to the difficulty of realising it using analogue methods. The goal was to show a basic, but working, digital implementation of a harmoniser. The harmoniser implementation, based on time stretching and pitch shifting as shown in Listings 5.5 and 5.6, required a buffer size of 512 samples, resulting in an initial latency of 24.4 ms as reported by Focusrite Control. The reported MET over 500 method calls was approximately 0.002152 seconds, or 2.152 ms. This resulted in a total latency of approximately 26.35 ms, without counting external factors also contributing to latency. Although this latency is still below the Haas threshold, it would likely affect the performance of a professional musician negatively.

Table 6.1: Mean execution time (MET) for each implemented effect

Effect	Buffer Size (Samples)	MET (ms)	Total Latency (ms)
Overdrive	128	0.016	6.42
Reverb + Delay	128	0.384	6.78
Harmoniser	512	2.152	26.35

As for the subjective evaluation of the implemented effects, satisfactory results were achieved regarding sound. As previously mentioned, perfect implementations of the effects were not the goal of the thesis. Instead, showing that it can be done at low latency using Python, was the focus. Audiovisual presentations of the implemented effects are listed in Appendix D.

The implemented overdrive effect utilises the hyperbolic tangent *tanh* for creating a non-linear output signal. This yielded a distorted sound typical for the overdrive guitar effect. The implementation of the combined reverberation and delay effect also resulted in the desired effect. The harmoniser effect, however, did contain unwanted audible features due the attempt of eliminating signal discontinuities through crossfading, as described in Section 5.3.3.

Overall, the effects implemented resulted for the most part in the desired sound at low latency. No real attempt, however, was made at monitoring the system-wide latency from input signal to output signal. Therefore, it is impossible to determine the real-world latency of the implementation.

6.2 Latency Reduction

The experiment utilised Cython to optimise the Python implementations for faster execution speed. The combined reverberation and delay effect was the only effect optimised due to its loop-based implementation, as Cython's largest improvements can be seen with loops. The speedup of the Cython-optimised implementation is presented in Table 6.2, an extended version of Table 5.2.

Table 6.2: Extended version of Table 5.2 including speedup

Buffer Size (Samples)	Python MET (ms)	Cython MET (ms)	Speedup
128	0.382435	0.006452	59.27
256	0.758102	0.007027	107.88
512	1.469831	0.007394	198.79
1024	2.858569	0.007854	363.96

As can be seen from the results, the speedup of the Cython-optimised implementation becomes increasingly significant when the number of loop iterations grows. For the implementation presented in Section 5.3.2, the number of iterations is equal to the buffer size used, resulting in an algorithmic complexity of $O(n)$. Without optimisation, this results in increasingly higher latency as the buffer size increases.

Further optimisation could possibly have yielded additional speedup for the combined reverberation and delay effect implementation. However, since latency reduction was not the main focus of the thesis, more effort was not put into it. The thesis still presented a method for reducing latency in Python for real-time signal processing using Cython, and the results of the performed optimisation showed reduction in latency. The objective of the experiment concerning latency reduction was, therefore, satisfied.

7. Discussion

The goal of this thesis was to explore the feasibility of implementing real-time guitar effects at low latency using Python. The proposed solutions implemented in Section 5.3, using Cython for latency reduction, as implemented in Section 5.4, yielded results that answer the proposed questions of Chapter 5:

1. How can common guitar effects such as overdrive, reverberation and delay be implemented digitally using Python?
2. How can signal latency be reduced during the DSP process using Cython?
3. How can a guitar effect be implemented digitally that would not be feasible using traditional, analogue methods?

The effect implementations in Section 5.3 satisfied the initial objective by implementing basic versions of the guitar effects overdrive, reverberation and delay. As presented in Section 6.1, the overall latencies of the implementations were well below the Haas threshold, with a system-wide latency of 6.42 ms for the overdrive effect and 6.78 ms for the combined reverberation and delay effect. This was further reduced to closer to 6.4 ms for the combined reverberation and delay effect using Cython.

A basic harmoniser effect was implemented in Section 5.3.3 to highlight one of the benefits of using digital methods over traditional, analogue methods. As discussed in Section 5.3.3, real-time pitch shifting has proven difficult using analogue solutions, while digital solutions are effortlessly implemented using algorithms. The harmoniser implementation relied on a combination of time stretching and resampling, where the signal was doubled in length and resampled at half the sampling rate, resulting in a pitch-shifted signal one octave higher than the original signal. As also mentioned in Section 5.3.3, the resampling process introduced audible discontinuities between audio segments, which were eliminated using crossfading, as presented in [51].

The implementations of the guitar effects provided satisfactory results when evaluated on a subjective basis. The performance of the CPU and audio interface used in the experiment, at a sampling rate of 96 kHz with a buffer size of 128 samples, resulted in no audible artefacts, apart from the audible modulation for the harmoniser effect due to the crossfading process. Audiovisual presentations of the implemented effects are given in Appendix D.

Due to the focus of this thesis being the concept of low-latency, real-time guitar effects using Python, the implemented effects were not complete, sophistically implemented, effects. This was especially obvious for the harmoniser effect because of the audible modulation resulting from the implemented crossfading. Proper filtering could have been utilised on the input and output signal for each effect, resulting in a cleaner sound. This limitation was mentioned multiple times throughout the thesis.

Another limitation of the experiment was the system-wide latency measurement. The latency measured only accounted for the reported round-trip latency reported by Focusrite Control, as well as, the reported execution time for each algorithm implementation. As discussed in Section 5.2, the usage of ASIO significantly reduced latency by allowing communication between audio sources and libraries to bypass OS-specific APIs, as shown in [45].

The implementations in Chapter 5 resulted in unexpectedly low latency, even without latency reduction using Cython. This was probably due to the powerful CPU used, combined with the ASIO protocol, as mentioned above. A potential follow-up to this experiment would be investigating the latency using CPUs commonly used in digital audio effect pedals, such as SHARC processors [54]. The low latency was also a result of the high sampling rate of 96 kHz used, combined with a buffer size of only 128 samples, which may not be feasible using a less powerful processor.

In general, the experiment answered the proposed questions regarding implementing real-time guitar effects using Python, at acceptable latency. The experiment showed that low-latency, real-time digital signal processing is possible using Python, while also presenting new opportunities for research within the area.

8. Conclusion

This thesis explored real-time signal processing in Python, with a focus on the implementation of common guitar effects digitally, including overdrive, reverberation and delay. Since implementing some guitar effects, such as real-time frequency shifting, has shown to be difficult using traditional, analogue techniques, this thesis also aimed to implement a signal harmoniser, an effect that pitch-shifts the signal up one octave. Furthermore, the thesis explored latency reduction using the programming language Cython, a superset of Python, with the goal of bringing the Python execution speed closer to that of C.

Signal processing as a concept was presented and discussed in the theoretical part of the thesis, highlighting the basics and common techniques. Areas relating to the experiment conducted were introduced, such as filtering, audio quality and data rate. Latency reduction was also briefly discussed, presenting common sources of latency in general, as well as, introducing Cython and ASIO.

Latency evaluation was performed based on the experiment conducted in [19], where professional musicians graded the perceived latency from “Horrible” to “Excellent” based on the auditory feedback received from a guitar at different latency levels. The threshold for a “Good” grade was 6.5 ms, using floor wedges for audio monitoring. Therefore, a sampling rate of 96 kHz with a buffer size of 128 samples was chosen for the experimental part of this thesis, resulting in an initial latency of 6.4 ms for the guitar effect implementations. The latency measurement, however, was one of the limitations of the thesis, since the total latency measured was only a measurement of the round-trip latency reported by Focusrite Control, combined with the execution time of each effect implementation, as measured with Python’s Time library.

The effects implemented performed at acceptable latencies, apart from the harmoniser effect. As presented in Table 6.1, the measured latency for the overdrive effect was 6.42 ms and for the combined reverberation and delay effect 6.78 ms. The harmoniser implementation resulted in a measured latency of 26.35 ms, due to a higher buffer size of 512 samples required for audio processing. Although still significantly below the Haas threshold [2], this latency would be noticeable to a trained ear, as concluded by [19].

Based on the results, it was concluded that Python implementations of low-latency guitar effects is possible, but highly dependent on hardware capabilities. Low latencies were received mostly due to a combination of high sampling rate and small buffer size, which may not be feasible to use on less powerful, more resource-limited systems. A topic for future research within the field was proposed, with the focus on implementing similar effects using Python as presented in this thesis on typical DSP processors, such as SHARC processors.

9. Summary in Swedish – Svensk sammanfattning

Digitala gitarreffekter med låg latens genom användning av signalbehandling med Python i realtid

Signalbehandling har historiskt sett spelat en viktig roll inom musiken, speciellt med tanke på ljudeffekter. I och med digitalteknikens framträdande har även många områden inom musiken till stor del digitaliserats. För elgitarrens del har digitala effekter under de senaste decennierna blivit alltmer vanliga, i och med utvecklingen av digitala realtidslösningar för traditionella, analoga effekter. Som nämns i [6] medför digitala metoder diverse fördelar, bland annat:

1. Flexibilitet: digitala signaler kan enkelt modifieras medan analoga signaler ofta kräver hårdvaruförändringar.
2. Reproducerbarhet: digitala signaler kan återskapas identiskt medan analoga signaler beror på de underliggande hårdvarukomponenterna.
3. Pålitlighet: digitala system förändras inte med tiden på samma sätt som elektriska komponenter i analoga system kan göra.
4. Komplexitet: komplexa system, såsom system baserade på maskininlärning, går inte enkelt att förverkliga med analoga system.

Den främsta nackdelen med digitala lösningar att ta i beaktande, speciellt vad beträffar realtidsapplikationer, är latens, eller fördröjning. I och med konverteringsprocesserna för en signal från analog till digital (A/D) och digital till analog (D/A) uppstår fördröjning som beror på samplingsfrekvens och sampelbuffertstorlek [7]. Utöver denna fördröjning uppstår även fördröjning på grund av själva signalbehandlingen, med en storlek beroende på signalprocessorns prestanda.

Huvudmålet med denna avhandling var att utforska de möjligheter Python erbjuder för signalbehandling, och med hjälp av dessa utveckla basimplementationer för några av de vanligaste gitarreffekterna: overdrive (distorsion), eko och reverb. I och med att Python är ett tolkat språk har man traditionellt sett föredragit kompilerade språk såsom C eller C++ för realtidsapplikationer, på grund av den fördröjning som härstammar från tolkningsprocessen [26]. I dagens läge finns ett brett utbud av ljudbibliotek i Python som fungerar som API (applikationsprogrammeringsgränssnitt, eng. *application programming interface*) för det C-baserade ljudbiblioteket *PortAudio* [29], bland annat

PyAudio [27] och *Sounddevice* [28]. *PortAudio* stöder också drivrutinen ASIO (*Audio Stream Input/Output)* [46], som är ett låg-latens alternativ till operativsystemets egna ljuddrivrutiner [45]. *Sounddevice* tillsammans med ASIO utnyttjades i denna avhandling för implementationen av gitarreffekter.

Bortsett från implementationen av gitarreffekter var också latensreduktion ett mål med avhandlingen. För att reducera fördröjning som uppstår i samband med tolkningen av Python introducerades Cython, ett Python-baserat programmeringsspråk som möjliggör exekvering av kod skriven i Python med C-hastighet [30]. Detta uppnås genom kompilering av koden till C-kod som kan anropas direkt från Python. Cython användes främst för att optimera implementationen av eko- och reverbeffekterna, vars implementation utnyttjade en loop.

Effekterna implementerades i Python med hjälp av signalbehandlingsbiblioteken *NumPy* [39] och *Librosa* [40]. En samplingsfrekvens på 96 kHz användes kombinerat med en buffertstorlek på 128 sampel. Overdrive-effekten förverkligades genom att tillämpa den hyperboliska funktionen *tanh* på signalen, på grund av dess icke-linjära natur, som resulterade i distorsion. Eko- och reverbeffekterna implementerades som en och samma algoritm på grund av deras liknande natur, med olika värden på parametrar för att uppnå det önskade ljudet. I och med buffertstorleken på endast 128 sampel krävdes en extern sampelbuffert för att spara ljuddata längre än de senaste cirka 1.3 ms ($1\ s/\frac{96000\ Hz}{128}$), vilket implementerades med hjälp av en loop. En audiovisuell presentation av de implementerade effekterna ges i Appendix D.

Ett sista mål med avhandlingen var att implementera en gitarreffekt som traditionellt sett varit komplicerad att förverkliga med analoga metoder. För detta ändamål valdes en harmoniseringseffekt, som bygger på tonhöjdsförändring (eng. *pitch shift*) i realtid. Den implementerade effekten, som krävde en större buffert på 512 sampel, höjde gitarrsignalens tonhöjd en oktav. Implementationen förverkligades dessutom i stereo, med den oförändrade signalen i vänster kanal och signalen med en oktav högre tonhöjd i höger kanal. På grund av den filtrering som användes för att eliminera hörbara klick, som resultat av tonhöjdsförändringen, resulterade implementationen även i en hörbar modulation (se Appendix D).

Resultaten av de implementerade gitarreffekterna sett från både ett subjektivt och ett objektivt perspektiv uppfyllde avhandlingens mål. Python-implementationer för de ovannämnda gitarreffekterna i realtid presenterades och latensreduktion med Cython förverkligades för den kombinerade eko- och reverbeffekten. Cython-

implementationen visade sig vara mellan 59,27 (buffertstorlek på 128 sampel) och 363,96 (buffertstorlek på 1024 sampel) gånger snabbare än dess motpart. De totala uppmätta fördröjningarna för de olika effekterna var 6,42 ms för overdrive-effekten, 6,78 ms för den kombinerade eko- och reverbeffekten samt 26,37 ms för harmoniseringseffekten, på grund av den större buffertstorleken på 512 sampel gentemot 128 sampel för de andra effekterna.

Den främsta bristen med experimentet var saknaden av mätning av fördröjning över hela signalkedjan från början till slut. I avhandlingen uppmättes för signalen endast den initiala tur och returtid som ljudgränssnittet rapporterade kombinerat med exekveringstiden för varje enskild effektimplementation. Detta gav endast en helhetsbild över den fördröjning som ljudgränssnittet bidrog till kombinerat med den fördröjning som introduceras av signalbehandlingen. Experimentet i avhandlingen utfördes inte heller på en typisk processor för ljudbehandlingsapplikationer, såsom en SHARC-processor [54], utan på en processor avsedd för skrivbordsdatorer (se Appendix C). Detta kunde vara ett framtida forskningsobjekt för vidare forskning inom området.

References

[1] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, San Diego: California Technical Publishing, 1999.

[2] J.-M. Reveillac, *Musical Sound Effects: Analog and Digital Sound Processing*, Hoboken, NJ: John Wiley & Sounds, Inc, 2018.

[3] C. Karren, "Analog Versus Digital Guitar Pedals, Shaping Guitar Tones and Sparking Debates," Digital Commons, 2020.

[4] "Number of guitar effects pedals sold in the United States from 2005 to 2021," Statista Research Department, March 2022. [Online]. Available: https://www.statista.com/statistics/448499/number-of-guitar-effects-pedals-sold-in-the-us/. [Accessed 7 November 2023].

[5] "VOX Adio Air GT," [Online]. Available: https://voxamps.com/product/adio-air-gt/. [Accessed 7 November 2023].

[6] M. S. Kuo and H. B. Lee, *Real-Time Signal Processing*, Chichester: John Wiles & Sons Ltd., 2001.

[7] U. Zölzer, *Digital Audio Signal Processing*, Chichester: John Wiles & Sons, Ltd, 1997.

[8] E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing: A Practical Approach*, Addison-Wesley Publishing Company Inc., 1993.

[9] A. Gersho, "Principles of Quantization," *IEEE Transactions on Circuits and Systems,* vol. 25, no. 7, pp. 427-436, 1978.

[10] A. L. Schoenstadt, An Introduction to Fourier Analysis: Fourier Series, Partial Differential Equations and Fourier Transforms, 2006.

[11] "Electronics Projects," [Online]. Available: https://electronicsprojects.in/signals_and_systems/periodic-and-non-periodic-signal-difference-diagram-and-information/. [Accessed 16 November 2023].

[12] R. N. Bracewell, “The Fourier Transform,” *Scientific American,* vol. 260, no. 6, pp. 86-95, 1989.

[13] L. Thede, *Practical Analog and Digital Filter Design*, Artech House, Inc., 2004.

[14] T. Saramäki, S. Mitra and J. Kaiser, *Handbook for Digital Signal Processing*, John Wiley & Sons, Inc, 1997.

[15] J. Watkinson, *The Art of Sound Reproduction*, Focal Press, 1998.

[16] K. C. Pohlmann, *The Compact Disc Handbook*, Madison, Wisconsin: A-R Editions, Inc., 1992.

[17] F. Rumsey, *Desktop Audio Technology: Digital Audio and MIDI Principles*, Focal Press, 2004.

[18] M. W. Spratling, “A review on predictive coding algorithms,” *Brain and Cognition,* vol. 112, pp. 92-97, 2017.

[19] M. Lester and J. Boley, “The Effects of Latency on Live Sound Monitoring,” 2007.

[20] Y. Wang, “Low Latency Audio Processing,” 2017.

[21] R. Schreier and G. C. Temes, *Understanding Delta-Sigma Data Converters*, John Wiley & Sons, Inc, 2005.

[22] W. G. Gardner, “Efficient Convolution without Input-Output Delay,” Perceptual Computing Section, MIT Media Lab, Cambridge, MA, 1995.

[23] J. O. Smith, *Introduction to Digital Filters with Audio Applications*, W3K Publishing, 2007.

[24] Y. Wang and J. D. Reiss, “Time Domain Performance of Decimation Filter Architectures for High Resolution Sigma Delta Analogue to Digital Conversion,” *Audio Engineering Society Convention 132,* 2012.

[25] Y. D. Pra, F. Fontana and M. Simonato, “Development of Real-Time Audio Applications Using Python,” in *Machine Sounds, Sound Machines XXII Collowuio di Informatica Musicale*, 2018.

[26] Y. D. Pra and F. Fontana, “Programming Real-Time Sound in Python,” *Applied Sciences,* 2020.

[27] PyPI, “PyAudio,” [Online]. Available: https://pypi.org/project/PyAudio/. [Accessed 18 January 2024].

[28] “Python-sounddevice,” [Online]. Available: https://python-sounddevice.readthedocs.io/. [Accessed 13 February 2024].

[29] PortAudio, “PortAudio,” [Online]. Available: https://www.portaudio.com/. [Accessed 18 January 2024].

[30] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn and K. Smith, “Cython: The best of both worlds,” *IEEE Computing in Science and Engineering,* 2011.

[31] K. W. Smith, Cython: *A Guide for Python Programmers*, Sebastopol, CA: O'Reilly Media, Inc., 2015.

[32] M. I. WIlbers, H. P. Langtangen and Å. Odegård, “Using Cython to Speed up Numerical Python Programs,” 2009.

[33] P. Perov, W. Johnson and N. Perova-Mello, “The physics of guitar string vibrations,” 2015.

[34] I. Simola, “Tablature Notation From Monophonic Guitar Audio Using CNN,” 2023.

[35] “Audacity,” [Online]. Available: https://www.audacityteam.org/. [Accessed 30 January 2024].

[36] U. Zölzer, *DAFX - Digital Audio Effects*, Chichester: John Wiley & Sons, Ltd, 2002.

[37] A. Politis, T. Pihlajamäki and V. Pulkki, “Parametric Spatial Audio Effects,” in *Proc. of the 15th Int. Conference on Digital Audio Effects (DAFx-12)*, York, UK, 2012.

[38] J. O. Smith, *Physical Audio Signal Processing for Virtual Musical Instruments and Audio Effects*, W3K Publishing, 2010.

[39] N. team, “NumPy,” [Online]. Available: https://numpy.org/. [Accessed 13 February 2024].

[40] Librosa, “Librosa,” [Online]. Available: https://librosa.org/doc/latest/index.html. [Accessed 5 March 2024].

[41] Focusrite, “Focusrite Control,” [Online]. Available: https://downloads.focusrite.com/focusrite/scarlett-3rd-gen/scarlett-2i2-3rd-gen. [Accessed 20 February 2024].

[42] Microsoft, “Windows 11,” [Online]. Available: https://www.microsoft.com/en-us/windows/. [Accessed 20 February 2024].

[43] “Focusrite Scarlett 2i2 (3rd Gen),” [Online]. Available: https://focusrite.com/products/scarlett-2i2-3rd-gen. [Accessed 25 January 2024].

[44] F. Design, “Define C,” [Online]. Available: https://www.fractal-design.com/products/cases/define/define-c/black/. [Accessed 13 February 2024].

[45] Y. Wang, R. Stables and J. Reiss, “Audio latency measurement for desktop operating systems with onboard soundcards,” in *Audio Engineering Society: Convention Paper 8081*, London, UK, 2010.

[46] Steinberg, “ASIO,” [Online]. Available: https://forums.steinberg.net/t/asio-what-is-it/201552. [Accessed 20 February 2024].

[47] A. v. Powerlord, “Measurement of MIDI-to-Audio Latency and Jitter of Synthesizer Plug-ins inside of Selected Live Performance Hosts at Different ASIO Buffer Sizes,” 2021.

[48] D. T. Yeh, J. S. Abel and J. O. Smith, “Simplified, Physically-Informed Models of Distortion and Overdrive Guitar Effects Pedals,” in *Proc. of the 10th Int. Conference on Digital Audio Effects (DAFx-07)*, Bordeaux, FR, 2007.

[49] E. Zeki, “Digital Modelling of Guitar Audio Effects,” 2015.

[50] T. Royer, “Pitch-shifting algorithm design and applications in music,” 2019.

[51] C. Alto and E. Cheveer, “Signal Harmonization by Resampling to Design a Guitar Effect Pedal,” 2016.

[52] Cython, “Extension Types,” [Online]. Available: https://cython.readthedocs.io/en/latest/src/tutorial/cdef_classes.html. [Accessed 9 March 2024].

[53] Python, “Time,” [Online]. Available: https://docs.python.org/3/library/time.html#time.perf_counter. [Accessed 9 March 2024].

[54] A. Devices, “SHARC Audio Processors / SoCs,” [Online]. Available: https://www.analog.com/en/product-category/sharc-audio-processors-socs.html. [Accessed 19 March 2024].

[55] “Yamaha Pacifica 112J,” [Online]. Available: https://usa.yamaha.com/products/musical_instruments/guitars_basses/el_guitars/pacifica/specs.html. [Accessed 25 January 2024].

[56] AMD, “Ryzen 7800X3D,” [Online]. Available: https://www.amd.com/en/products/apu/amd-ryzen-7-7800x3d. [Accessed 13 February 2024].

Appendices

Appendix A – Yamaha Pacifica 112J Technical Specifications

Technical specifications for the Yamaha Pacifica 112J electric guitar [55].

Construction: Bolt-on

Scale Length: 25-1/2” (648 mm)

Body Materials: Alder

Neck Materials: Maple

Fingerboard Materials: Rosewood

Fingerboard Radius: 13-3/4” (350 mm)

Fret Wire: Medium

Number of Frets: 22

Nut Materials: Urea

Neck Width at 0 Fret / 12th Fret: 41/51.4 mm

Thickness at 1st Fret / 12th Fret: 20.9/22.9 mm

Neck Pickup: Single Coil/Ceramic

Middle Pickup: Single Coil/Ceramic

Bridge Pickup: Humbucker/Ceramic

Controls: Master Volume, Master Tone

Pickup Switch: 5-Position Lever Switch

Bridge: Vintage-Style Tremolo

String Spacing: 10.5 mm

Tuning Machines: Die-Cast

String Gauge: Ernie Ball Super Slinky 009-042

Appendix B – Focusrite Scarlett 2i2 Technical Specifications

Technical specifications for the Focusrite Scarlett 2i2 (3rd Generation) USB Audio Interface [43].

Overview

Protocol: USB 2.0

Simultaneous I/O: 2x2

A/D Resolution: 24-bit/192 kHz

Number of Pre-Amps: 2

Phantom Power: Yes

Instrument Inputs: 2

Line Inputs: 2

Analogue Inputs: 2

Headphone Outputs: 1

Bus Powered: Yes

Supported Sample Rates: 44.1, 48, 88.2, 96, 176.4, 192 kHz

Microphone Inputs

Frequency Response: 20 Hz – 20 kHz ± 0.1 dB

Dynamic Range: 111 dB (A-weighted)

Maximum Input Level: 9 dBu (at minimum gain)

Gain Range: 56 dB

Impedance: 3 kΩ

Line Inputs

Frequency Response: 20 Hz – 20 kHz ± 0.1 dB

Dynamic Range: 110.5 dB (A-weighted)

THD+N: < 0.002 %

Maximum Input Level: 12.5 dBu (at minimum gain)

Gain Range: 56 dB

Impedance: 60 kΩ

Instrument Inputs

Frequency Response: 20 Hz – 20 kHz ± 0.1 dB

Dynamic Range: 110 dB (A-weighted)

THD+N: < 0.03 %

Maximum Input Level: 12.5 dBu (at minimum gain)

Gain Range: 56 dB

Impedance: 1.5 MΩ

Line/Monitor Outputs

Dynamic Range: 108 dB

THD+N: < 0.002 %

Maximum Output Level (0 dBFS): 15.5 dBu

Impedance: 430 Ω

Headphone Outputs

Dynamic Range: 104 dB (A-weighted)

THD+N: < 0.002 %

Maximum Output Level (0 dBFS): 7 dBu

Impedance: < 1 Ω

Appendix C – AMD Ryzen 7 7800X3D Technical Specifications

Technical specifications for the AMD Ryzen 7 7800X3D CPU [56].

Overview

# of CPU Cores: 8

# of Threads: 16

Max. Boost Clock: 5.0 GHz

Base Clock: 4.2 GHz

L1 Cache: 512 KB

L2 Cache: 8 MB

L3 Cache: 96 MB

Default TDP: 120 W

Appendix D – Audiovisual Presentation of Implemented Effects

Reference audio of the implemented guitar effects.

Overdrive - Reference Audio

https://youtu.be/wLSomkoesTo?si=ybWMvZJg-m5Nt0Or&t=5

Delay & Reverberation - Reference Audio

https://youtu.be/wLSomkoesTo?si=5EKQ5zMquOAsTf_5&t=73

Harmoniser Using Pitch Shifting - Reference Audio

https://youtu.be/wLSomkoesTo?si=QWME4bQmWTXqCY78&t=132