School of Electronic and Electrical Engineering
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

ELEC3875

Project Report

DSP Algorithms for Musical Special Effects

Thomas Milner

SID:	201397578	Project No.	55
Supervisor:	Dr Des McLernon	**Assessor:**	Dr Paolo Actis

School of Electronic and Electrical Engineering
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

ELEC3875 Individual Engineering project

Declaration of Academic Integrity

Plagiarism in University Assessments and the Presentation of Fraudulent or Fabricated Coursework

Plagiarism is defined as presenting someone else's work as your own. Work means any intellectual output, and typically includes text, data, images, sound or performance.

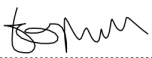
Fraudulent or fabricated coursework is defined as work, particularly reports of laboratory or practical work that is untrue and/or made up, submitted to satisfy the requirements of a University assessment, in whole or in part.

Declaration:

- I have read the University Regulations on Plagiarism [1] and state that the work covered by this declaration is my own and does not contain any unacknowledged work from other sources.
- I confirm my consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to monitor breaches of regulations, to verify whether my work contains plagiarised material, and for quality assurance purposes.
- I confirm that details of any mitigating circumstances or other matters which might have affected my performance and which I wish to bring to the attention of the examiners, have been submitted to the Student Support Office.

[1] Available on the School Student Intranet

Student name: Thomas Milner Project No. 55

Signed:  Date: 14/05/2024

Abstract

This project delves into the DSP theory of some common audio effects: Echo, Distortion, Tremolo, Reverb, and the Vocoder. For each effect, a thorough mathematical analysis is performed to gain an understanding of the DSP techniques at work, and an implementation of the effect is created using MATLAB and/or the VST Plugin format. Finally, demo recordings of each effect applied to an electric guitar input signal are made and evaluated; these recordings are accessible to the reader.

Acknowledgements

First, I'd like to thank my supervisor, Dr Des McLernon, for guiding me through this project. His DSP crash course sessions were not only incredibly valuable, but fun, and instrumental to the success of this project.

Second, I'd like to thank my family and housemates for putting up with my (seemingly never-ending) audio-processing-related ramblings. It seems that now I'll have to find something else to talk about!

Last, but certainly not least, I'd like to thank the Brotherton folk for their inspiration and good company; good times!

Contents

Glossary

F_s Sampling Rate. 23

audio interface Device that digitises a guitar signal for use within a DAW.. 19

Bela A specialised audio processing development board [1]. 20

JUCE An open source C++ framework for developing audio programs [2]. 21

REAPER The DAW of choice for this project [3]. 20

tone The sonic characteristics of a guitar's output. 12

VST plugin A program that runs in a DAW that can modify digital audio signals, and thus produce audio effects. 15

Acronyms

DAW Digital Audio Workstation. 15

DSP Digital Signal Processing. 13

FBCF Feedback Comb Filter. 44

FFCF Feedforward Comb Filter. 24

GUI Graphical User Interface. 21

LFO Low Frequency Osillator. 37

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

To produce sound, an electric guitar produces an analogue electric signal that is output via an instrument cable. The cable is plugged into an amplifier, which amplifies the electric guitar signal and outputs it through a speaker at an audible volume. This series of connections is called a **signal chain**.

The characteristics of the sound of the guitar coming out of the speaker (the **tone** of the guitar) can be modified by manipulating the electric signal at different points in the signal chain. For example, electric guitar amplifiers usually have a set of control knobs allowing a player to adjust the volume of the signal, as well as tweak the strengths of different frequency bands in the signal (low, middle, and high, otherwise known as bass, "*mids*", and treble). A guitarist can use these knobs to adjust how the guitar sounds coming out of the speaker - the tone.

Modifications can be introduced to any part of the signal chain to produce a different tone. Thus, an easy and flexible way to change the tone is to modify the signal between the guitar and the amplifier. This is done using guitar pedals, which are small devices that sit in between the guitar and the amplifier in the signal chain and impose audio effects on the signal. The name "pedal" comes from their physical format; they are floor-based devices that are controlled using the guitarist's feet, to allow the guitarist to keep their hands on the guitar. Different pedals can implement different modifications to the signal, producing different sound effects. Pedals can also be chained together to "stack" effects on top of each other.

Some example effects (all of which will be implemented in this project) are as follows:

Echo

This is an effect where the input signal is mixed with a delayed version of itself before being output, making it sound like there is an echo (confusingly, this can sometimes be called a "Delay" effect).

Distortion

This is an effect widely used in rock music that adds higher frequencies (technically, overtones) to the original signal to make it sound grittier, crunchier, and more aggressive.

Tremolo

This effect rapidly alters the volume of the signal, creating a pulsing or shimmering effect.

Reverb

This effect emulates the sound of the guitar in a large echoing room. It simulates the natural reflections of sound in a physical space, creating a sense of depth and space.

Vocoder

This effect is the most complex of the selection. A vocoder takes two input signals and produces a new signal that has the characteristics of both original signals.

Guitar pedals are traditionally implemented using analogue circuitry. This means that each pedal can usually only implement one effect, as the pedal only contains one circuit. As such, building up a collection of effects can get expensive, as many different pedals are required. A large collection of pedals can also take up a lot of physical space.

Digital Signal Processing (DSP) offers a solution to these problems. By implementing audio effects digitally (using software), one effects processing unit can use different algorithms to implement different effects, thus reducing the need for multiple pedals. This software-based approach also allows for fast development (as specialist circuitry/components are not required), and the divergence from the traditional pedal-based format; the effects can run on any device that the software can run on, including microcontrollers or laptops.

1.2 Project Objectives

The objectives of this project were to produce, for each of the effects listed below:

1. A comprehensive understanding, analysis and evaluation of the effect using DSP theory.
2. A MATLAB implementation of the effect using DSP.
3. A VST Plugin of the effect (for the simpler effects; VSTs are explained in Section 1.4.2).
4. A recording of the effect in action (using either the MATLAB or VST implementation).

The effects in question were: **Echo**, **Distortion**, **Tremolo**, **Reverb**, and a **Vocoder**.

VST plugins were to be produced for Echo, Distortion and Tremolo, but not for Reverb nor the Vocoder. VSTs for these two more complex plugins were omitted because developing a plugin is time-consuming and does not add to the underlying DSP knowledge of an effect, and it was decided that it would be more valuable to use the time to gain a deeper understanding of each effect.

An extension objective of the project was to use the effects to create a physical effects unit, that can be used more like a traditional guitar pedal.

A significant portion of the project duration was spent learning the underlying DSP theory to use to analyse and construct the effects – learning DSP in the context of audio effects was the major objective for this project.

1.3 Why is it Worth Investigating?

The author of this project plays electric guitar and wanted to build up a larger collection of guitar effects. The author also wanted to learn more about DSP theory. By creating guitar effects using DSP, the author could satisfy both objectives: building up a collection of custom effects and learning about DSP in audio.

1.4 Current State of the Art

Using DSP to implement audio effects is common and can come in different forms.

1.4.1 DSP Integrated Pedals

These have the same format as traditional guitar pedals; they are physical foot-controlled devices that sit between a guitar and an amplifier in the signal chain. They contain a combination of hardware and software to digitise the guitar signal, implement any number of audio effects, and convert the signal back into an analogue signal to be carried along the signal chain. As this requires both hardware and software, it would take a lot of work to produce and thus is out of scope for this project.

1.4.2 VST Plugins

This approach moves away from the traditional pedal format by processing the effects on a standard PC. The guitar signal is passed into an audio interface, which digitises the signal and outputs it to the PC. The PC is running a piece of software called a **Digital Audio Workstation (DAW)**, a tool for recording and editing audio. The DAW can run **VST plugins**, which are programs that can modify digital audio signals and thus create audio effects [4]. The VST Plugins are synonymous with effect pedals; they can be chained together to "stack" effects. The modified audio can then be output back out of the audio interface to some speakers.

1.5 Possible Applications

The project aimed to implement the audio effects in MATLAB, as well as produce VST plugins for some of the effects. These plugins work in real-time with a live guitar input signal, meaning they can be used for recording real guitar parts when producing music. They can also be output to a speaker system for live performances.

Chapter 2

Societal Factors

2.1 Sustainability

This project is mainly software and so does not have any large negative sustainability impacts; Many guitar effects can be created just by writing more MATLAB analyses and VST plugins, at no extra resource cost. The project delivers another positive sustainability impact by preventing the future purchasing of analogue guitar pedals, with additional usable effects becoming available though the same software. This would prevent building a collection of pedals that could eventually end up in landfills or e-waste sites.

The project requires an audio interface, which is hardware. Only one is required and although the option of borrowing a device from elsewhere was explored, all attempts were fruitless, and an interface had to be purchased specifically for the project. If it is not repurposed after the project's duration, it could end up as waste: therefore, the audio interface must be given a new purpose after the project, either in the School of Electronics or elsewhere.

2.2 Ethics

This project is unlikely to raise any ethical issues, as it is just a software exploration of pre-existing audio effects. Thus, there are minimal confidentiality risks; the project is unlikely to mimic any proprietary audio processing algorithms (only commonly known algorithms) and the effects are not going to be commercially released.

As the project involves audio signals there is a risk of harm coming to the user through unsafe

volume levels, but this can be easily mitigated by implementing volume limits on the laptop/PC (or in the software).

This project has a low opportunity cost; if the effects were implemented on a dedicated piece of hardware (say, a microcontroller), although it would've been beneficial to have a self-contained effects unit it would've taken much more work to produce the hardware, whilst the result would be the same effects.

VST plugins are made for some of the effects, but not all (see Project Objectives 1.2). This is unfortunate as it means some effects can't be used in real-time with an electric guitar, but proves to be a productive decision as it allows more in-depth DSP analysis of the effects in question, without the time pressure of creating a plugin.

By using the VST interface to create the plugins the flexibility to implement the audio effects in other systems is lost, but this is an acceptable cost.

2.3 Risk Management

2.3.1 Lack of Information

DSP audio effect algorithms have been mainly pioneered by industry (not academia) and so many of the inner workings of the algorithms are not available to the public. As such, there was a risk there would not be enough publicly available information to implement the effects chosen for this project.

This was an unlikely option, as a lot of time was spent researching the feasibility of the different effects. However, if this problem did arise, the damage would be mitigated by changing the selection of effects; some brief preliminary research was done on effects like **flanging**, **chorus**, **boost**, and **vibrato** and they were decided to be viable alternatives, if necessary.

An alternative approach would have been to consult the project supervisor and work out a way of implementing each effect that is "close enough" to the original, even if not a perfect reconstruction.

2.3.2 VST Plugin Latency Issues

There was a risk with the VST plugins that once they were implemented there would be too much latency in the signal chain (audio interface to laptop, through DAW and VSTs, to speakers) to use the effects in real-time.

To mitigate this risk, at the start of the project the latency of the signal chain was tested with some off-the-shelf VST plugins. The plugins exhibited very little latency, proving the hardware was indeed fast enough and the only bottleneck could be the algorithms produced during the project. In the end there were never any issues with VST Plugin latency, but if there were, the plan was to:

1. Experiment with the sampling rates and buffer sizes in the VST plugins to reduce the number of samples needing to be processed per second, thus decreasing the computing demand.
2. Research more efficient DSP algorithms, and detail this research in the final report.

2.3.3 DSP Algorithms Could Be More Complicated Than Anticipated

There was a risk the DSP theory behind the effects would be too complex for the author to understand. To mitigate this risk, time was spent before the project started to familiarise with basic DSP theory, in the hopes of building a solid foundational knowledge base.

2.4 Equality, Diversity, and Inclusion

Equality, diversity, and inclusivity considerations were not relevant to this project, as the project was objective research focused on the technical aspects of DSP. The technology produced was used exclusively by the project author for research purposes only – it was not, and will not be, available to a wider audience. There was no human-based or human-centric research; all the required learning came from textbooks, library resources, online resources, or the project supervisor. Thus, there were no instances where considerations concerning equality, diversity and inclusivity were relevant.

Chapter 3

Environment Setup

To start, research was undertaken to learn what technology and equipment was available to produce the VST plugins and MATLAB effects.

3.1 Equipment

All the equipment purchased for the project can be seen in Figure 3.1.

Figure 3.1: The purchased equipment. Left: "AKG K52" headphones [5]. Top Right: "Bela" Audio Processing Platform [1]. Bottom Right: "Focusrite Scarlett Solo Audio Interface" [6].

3.1.1 Audio Interface

An audio interface was required to get the signal from the electric guitar into the DAW. There were very loose requirements for the audio interface; it just needed to be able to convert a single

electric guitar signal for use within a DAW with minimum distortion, which all audio interfaces do. The University of Leeds School of Music helpdesk was contacted, who recommended the "Focusrite Scarlett Solo"[6]. Further research was conducted in the form of browsing online sources, as well as reaching out to local musicians. All sources agreed that a refurbished Scarlett solo would be sufficient for this project. Thus, one was purchased.

3.1.2 Headphones

"AKG K52 headphones"[5] were purchased to use with the audio interface, as the audio interface required a specific output connector (1/4-inch headphone jack).

3.1.3 Bela Audio Development Platform

An extension objective of the project was to implement some of the researched effects on an embedded device. A "Bela"[1] audio development board was purchased (at the suggestion of the project's supervisor) so that the hardware would be available if it was eventually required.

The Bela was chosen as it contains a lot of the circuitry required for sampling and processing an analogue audio signal, allowing the author to focus on the effects.

3.2 Digital Audio Workstation (DAW)

The next step was to research the DAW to use. Again, there were minimal requirements for this project component: it needed to be able to record and playback signals and support applying VST plugins to a signal. All modern-day DAWs support these features, so the final decision was a matter of personal preference - "REAPER" [3] was chosen. Familiarisation of the DAW was achieved through using it to record guitar signals and applying third-party VST plugins to the signal, to learn the software workflow.

3.3 VST Development Framework

There were two main software frameworks available for developing VSTs:

3.3.1 VST 3 SDK

This is the official barebones C++ SDK for VST development [7] produced by the creators of VST, "steinberg".

3.3.2 JUCE Audio Development Framework

JUCE is "the most widely used framework for [...] plug-in development" [2]. It is an open-source C++ framework that contains vast libraries of audio processing tooling, as well as libraries for Graphical User Interface (GUI) development. It has a much simpler interface than the official VST SDK and has a large community presence providing a plethora of tutorials and guides on how to effectively use the framework [8] [9].

The decision was made to use the JUCE development framework, the main reasoning being that JUCE sits on top of the VST SDK, abstracting away a lot of the boilerplate technical code required for developing a VST plugin. This means that more time could be spent on the DSP implementation of effects, as less time was needed to understand the internal construction of a VST plugin. The community has also produced plenty of tutorials for various parts of JUCE, and the GUI libraries should allow for easier development of GUIs for the plugins. To learn the framework, a simple VST plugin that passed the input directly to the output (zero processing) was created.

Chapter 4

Echo Effect

The first effect undertaken was the simplest: the echo effect. The echo effect makes the guitar sound like it is being echoed; you hear the original signal, and then you hear an attenuated version of the same signal a small time delay later.

4.1 Analysis

4.1.1 Time Domain: Echo Filter

To create the echo effect, the "echo filter" detailed in "Introduction to Signal Processing" [10] was used (Figure 4.1). The filter works by adding the input signal to a delayed, attenuated version of itself.

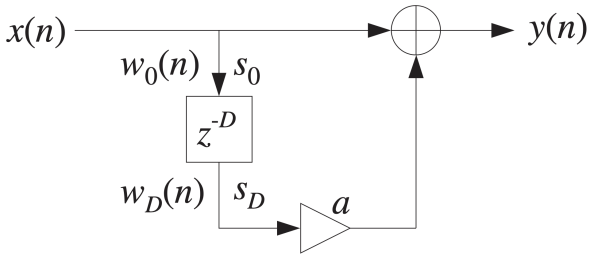


Figure 4.1: Echo Filter Diagram [10].

The linear difference equation of this filter is:

$$y(n) = x(n) + ax(n - D) \tag{4.1}$$

$$a = \text{Gain of the delayed signal}, \quad |a| < 1$$

$$D = \text{Delay constant (how long to wait before replaying the signal).}$$

It is a causal, feedforward filter. To implement this in MATLAB, the impulse function version of Eq. 4.1 was used which is simply:

$$y(n) = \delta(n) + a\delta(n - D) \tag{4.2}$$

$$\delta(n) = \text{Dirac delta (impulse) function.}$$

The impulse response of the echo filter is shown in Figure 4.2.

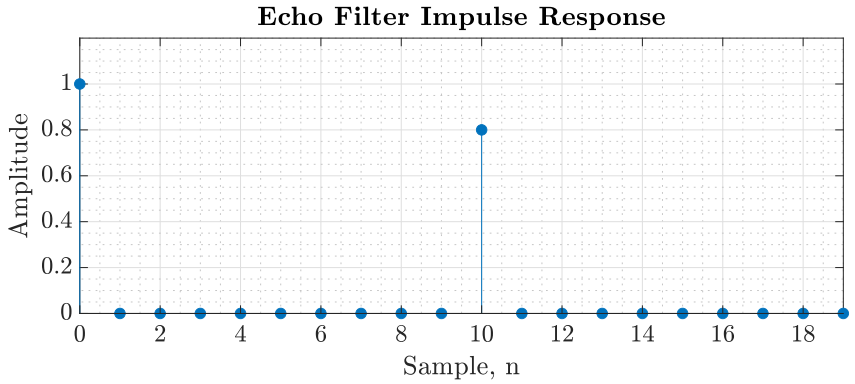


Figure 4.2: Impulse response of the echo filter for $a = 0.8, D = 10$.

Example

With the variables listed below, we get the following echo filter (F_s = sampling rate) (Eq. 4.3):

$a = 0.5$

$F_s = 44100$ Hz

$D = 0.3$ seconds $\times\ F_s = 0.3 \times 44100 = 13230$ samples

$$y(n) = x(n) + 0.5x(n - 13230) \tag{4.3}$$

When applied to a 500 Hz, 0.1 s duration sine-wave, this filter gives the result shown in Figure 4.3.

As can be seen, the original input burst is replayed 0.3 seconds after the original signal, at half the original amplitude, producing an echo.

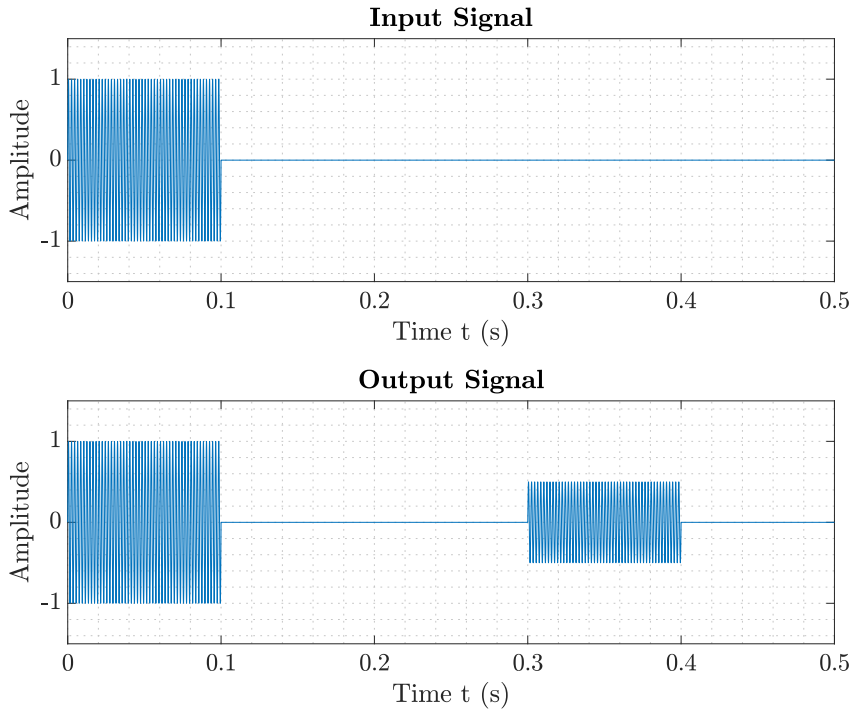


Figure 4.3: Echo filter Eq. 4.3 with a 500 Hz, 0.1 s sine wave input.

4.1.2 Frequency Domain: Feedforward Comb Filter (FFCF)

The echo filter is better known as a Feedforward Comb Filter (FFCF), as the magnitude frequency response of the filter resembles an upside-down comb. As the delayed signal is added to the original signal, constructive or destructive interference occurs depending on the phase difference of the signals. Different frequencies react differently to the FFCF depending on the relationship between the period of the signal and the filter delay constant D (from Eq. 4.1). We can work out what frequencies are affected, and how exactly, as follows. The FFCF can be said to have a **fundamental frequency** F_D of [10]:

$$F_D = F_s/D_{\text{samples}} = 1/D_{\text{seconds}} \text{ (Hz)} \tag{4.4}$$

When an input signal is a frequency of a multiple of F_D, the delayed signal will be in phase, causing constructive interference – there will be **peaks** in the frequency response. When the input signal frequency is halfway between multiples of F_D, the delayed signal will be out of phase, and there will be deconstructive interference - **troughs** in the frequency response.

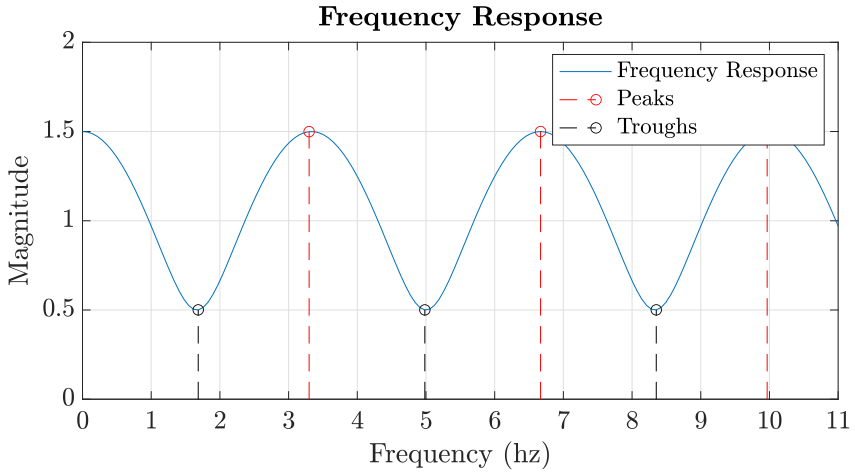


Figure 4.4: Frequency response of FFCF 4.3, up to 11 Hz.

Example

We will use Eq. 4.3 again as our example. The fundamental frequency F_D of the FFCF in Eq. 4.3 is 44100/13230 (or 1/0.3) = 3.33... Hz, meaning there will be peaks every 3.33 Hz in the frequency response of the filter. The troughs will occur in between the peaks:

Peaks (Hz): 3.33, 6.67, 10, ...

Troughs (Hz): 1.67, 5, 8.33, ...

These can be seen in the frequency response calculated in MATLAB (Figure 4.4).

The magnitude of the peaks and troughs can be calculated by considering the transfer function of the filter [10] (a and D have the same meaning as in Eq. 4.1):

$$H(z) = 1 + az^{-D} \tag{4.5}$$

To get the frequency response of the filter we can set $z = e^{j\omega}$, where ω is the normalised radian frequency (see *Normalised Radian Frequency* in Appendix A for details):

$$H(e^{j\omega}) = 1 + ae^{-j\omega D} \tag{4.6}$$

($H(e^{j\omega})$ is often given the alias $H(\omega)$ - this will be used for the remainder of this report.)

Now we have the frequency response of the filter, we can find the frequencies that will produce the peaks and the troughs. The fundamental frequency of the filter in radians/sample is (from

Eq. 4.4):

$$\omega_D = 2\pi/D \text{ (radians/sample)}$$

As mentioned before, the peaks will happen at multiples of the fundamental frequency, i.e. when:

$$\omega = k\omega_D = \frac{2\pi k}{D} \qquad\qquad k = 0, 1, 2, \ldots$$

Therefore, the magnitude of the frequency response at the peaks will be:

$$\begin{aligned} H(\omega) &= 1 + ae^{-j\omega D} \\ &= 1 + ae^{-j\times 2\pi k/D\times D} \\ &= 1 + ae^{j2\pi k} \\ &= 1 + a \end{aligned}$$

Plugging in the value of $a = 0.5$ from the example filter from Eq. 4.3:

$$H(\omega) = 1 + a = 1.5$$

The peaks of the frequency response are 1.5; this can be seen in Figure 4.4.

The troughs will happen in between multiples of the fundamental frequency:

$$\omega = \frac{(2k+1)\pi}{D} \qquad\qquad k = 0, 1, 2, \ldots$$

Thus, the magnitude of the frequency response at the troughs will be:

$$\begin{aligned} H(\omega) &= 1 + ae^{-j\omega D} \\ &= 1 + ae^{-j\times \frac{(2k+1)\pi}{D}\times D} \\ &= 1 + ae^{j\pi(2k+1)} \\ &= 1 + ae^{j\pi} \\ &= 1 - a \end{aligned}$$

Plugging in the value of $a = 0.5$ from the example filter from Eq. 4.3:

$$H(\omega) = 1 - a = 0.5$$

The troughs of the frequency response should be 0.5; this too can be seen in Figure 4.4.

The pole-zero plot of an arbitrary FFCF can be seen in Fig 4.5. As can be seen from the plot, the filter has no zeros - hence, it is *feedforward* (if there were zeroes instead of poles, it would be feed*back*).

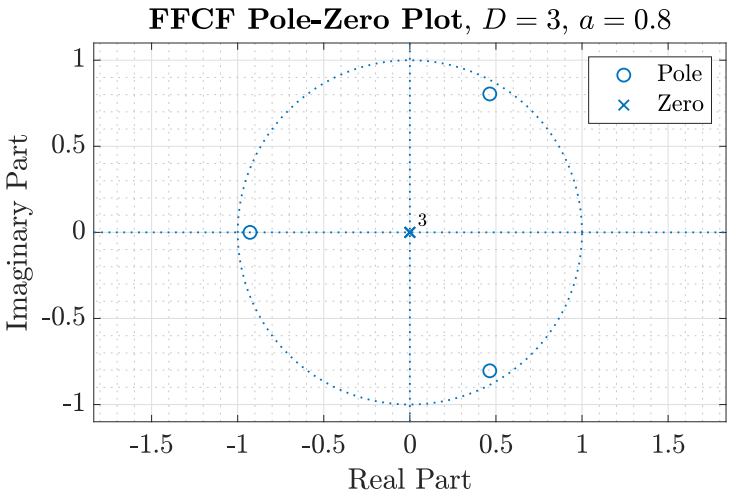


Figure 4.5: FFCF pole-zero plot.

4.2 Implementation: VST Plugin

The effect was implemented as a VST plugin using JUCE, and can be found in the Public Folder (Appendix B). To store the delayed signal, a delay line class was created using a circular buffer. The finished plugin can be seen in Figure 4.6. The "Delay Gain" parameter controls a and ranges between 0 and 1. The "Delay Length (ms)" parameter sets D (by converting the milliseconds into a quantity of samples), within a range of 0-400 ms. The Delay Length slider is "skewed", meaning that when the slider is in the middle position the value returned is 100, instead of half of the range (200). The distance required to move the slider to increase the value is much larger for smaller delay lengths, so the user gets better precision, and smaller for larger delay lengths. Both sliders also allow for number input.

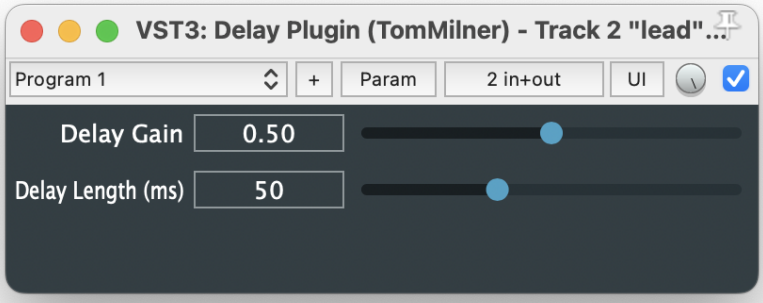


Figure 4.6: Echo effect VST plugin (confusingly referred to as a "Delay" effect in musical circles, hence the different name)

4.3 Demo Recordings

Demo recordings were made with the plugin in the REAPER DAW (Figure 4.7). Recordings of the plugin in use can be found in the Public Folder.

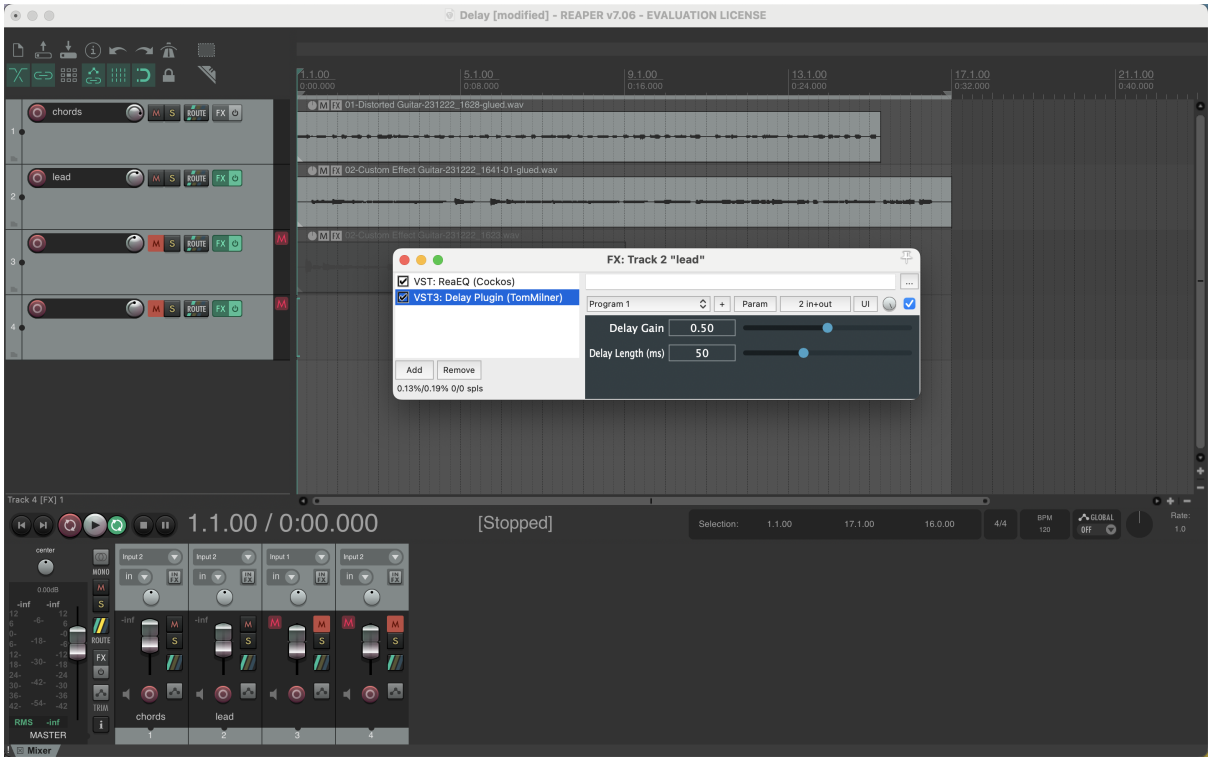


Figure 4.7: Echo VST plugin in use in the REAPER DAW.

4.4 Reflection

Although echo is a simple effect to implement, the MATLAB analysis taught the author a lot of low-level DSP theory about comb filters, impulse responses, transfer functions, and frequency responses. The produced plugin is easy to use and works well in real time. In the future, the plugin could benefit from additional echo-based effects like flanging or chorus modulation (Appendix C details how these work), but for this project, the plugin is sufficient.

Chapter 5

Distortion Effect

The next effect we will undertake is probably the most well-known, and the most distinctive, of the selection: **distortion**. It is a classic effect, used in some form in most genres of Western music - in particular, "*rock 'n' roll*".

5.1 Analysis

5.1.1 Time Domain: Square Wave Approximations

Distortion was originally implemented through hardware clipping; a signal would be amplified so its peak value would surpass a threshold, and any part of the signal above the threshold would be flattened out, or "clipped" [11]. This has the effect of making the signal tend towards a *square wave* shape which will result in *harmonics* being added to the signal, which we will explore in Section 5.1.2.

In his book *"DAFX: Digital Audio Effects, Second Edition"* [11], Udo Zölzer provides useful formulae for creating distortion effects through square wave approximations. The formula chosen to base the distortion implementation off was the following [11]:

$$f(x) = \text{sign}(x)(1 - e^{-|x|})$$

This function gives a sine-wave approximation that approaches a square wave at high powers of e (high values of x). Importantly, we cannot use the same analysis techniques for distortion as we did for echo, as distortion is a non-linear effect; echo was a linear time-invariant system,

whilst this is not.

The variables gain g and range r were added to allow fine-tuning of the "sharpness" of the square wave corners; a higher value of the product rg produces a squarer wave (as is seen in Figure 5.1).

$$f(x) = \text{sign}(x)(1 - e^{-r \times g \times |x|}) \tag{5.1}$$

$$g = \text{Gain, } 0...1 \text{ (continuous)}$$

$$r = \text{Range, } 0...100 \text{ (discrete)}$$

r and g are intended to be used as follows: the user sets the range r as the maximum acceptable distortion. They can then use the gain g to set the gain within that range, so they can fine-tune the distortion to get the perfect tone.

Figure 5.1 shows how tweaking r and g can produce a range of different tones, from sine wave approximations to square waves.

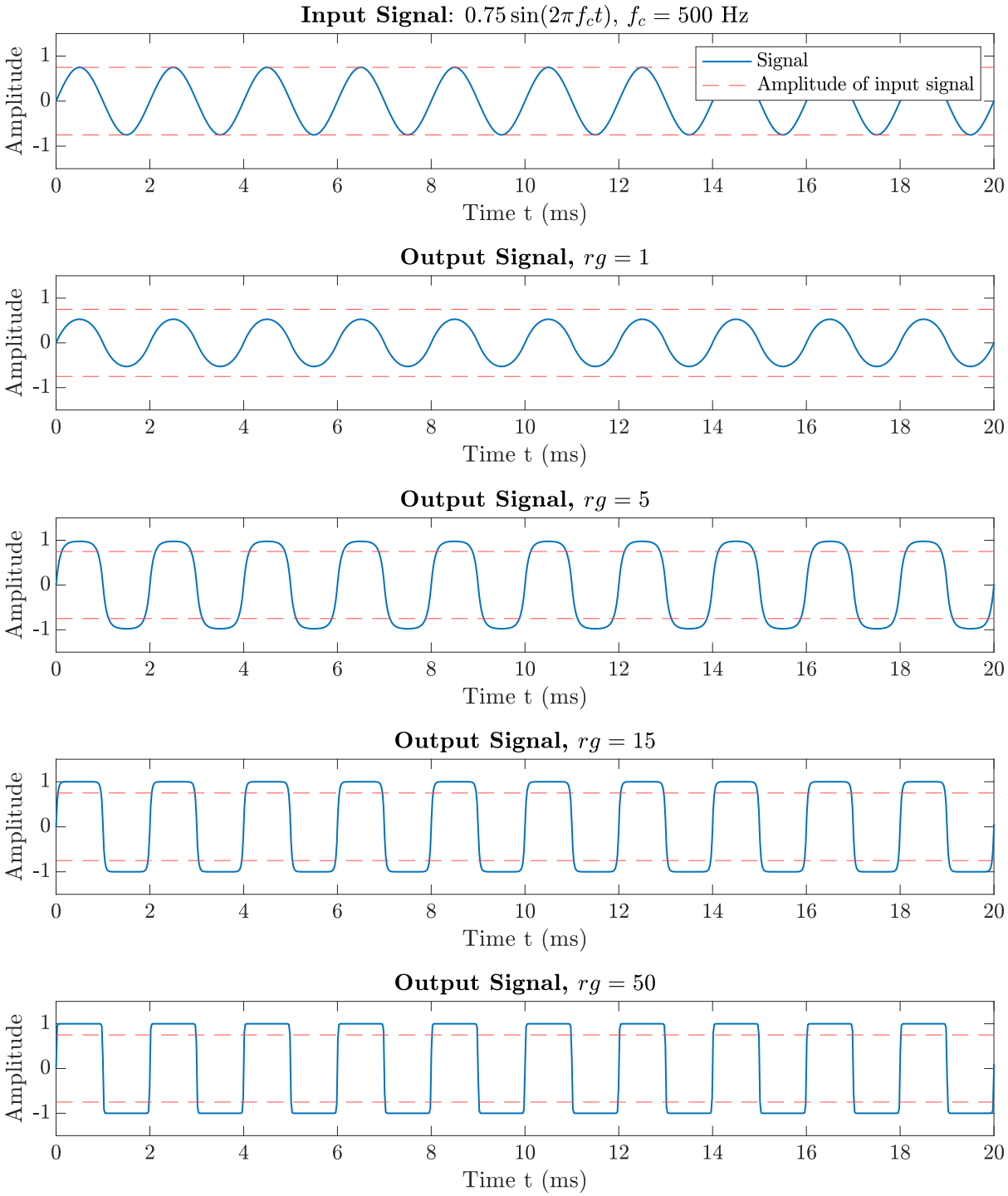


Figure 5.1: The sine-wave approximation from Eq. 5.1 turning into a square wave approximation as $r \times g$ increases.

Figure 5.1 also shows how the amplitude of the output signal varies with rg; a side effect of this filter is that it changes the volume of the signal.

This can be remedied by passing the output signal through a standard linear amplifier so the

user can manually adjust the output signal gain to their desired level [1] (as is done later for the VST plugin implementation in Section 5.2).

5.1.2 Frequency Domain: Harmonics

As mentioned earlier, the square wave approximation adds *harmonics* to the signal. Harmonics are frequency components added to a signal that are multiples of the signal's original frequency. Square waves only add odd harmonics, which are odd multiples of the original frequencies (i.e, if the original frequency is f_c, the odd harmonics are $3f_c$, $5f_c$, $7f_c$, etc).

These harmonics give the signal a grittier, more powerful sound; Figure 5.2 helps us see why. As the value of rg (from Eq. 5.1) increases, the energy of the odd harmonics increases, which can be seen as peaks in the plot.

To the listener, the harmonics are heard as extra notes; instead of hearing just the note being played, the listener hears *multiple notes at the same time*. When the guitarist does play multiple notes at the same time (*chords*) through a distortion effect, each of these multiple notes will produce their own harmonics, which will interact with each other to build upon the notes of the original chord. A common (and fascinating) example of this phenomenon is in *power chords* - see Appendix D.

5.2 Implementation: VST Plugin

As with the echo effect VST plugin (Section 4.2), the distortion effect was implemented using JUCE and tested in the REAPER DAW, and can be found in the Public Folder.

[1] This gain adjustment could be done automatically with automatic gain control (AGC), but that is outside the scope of this project.

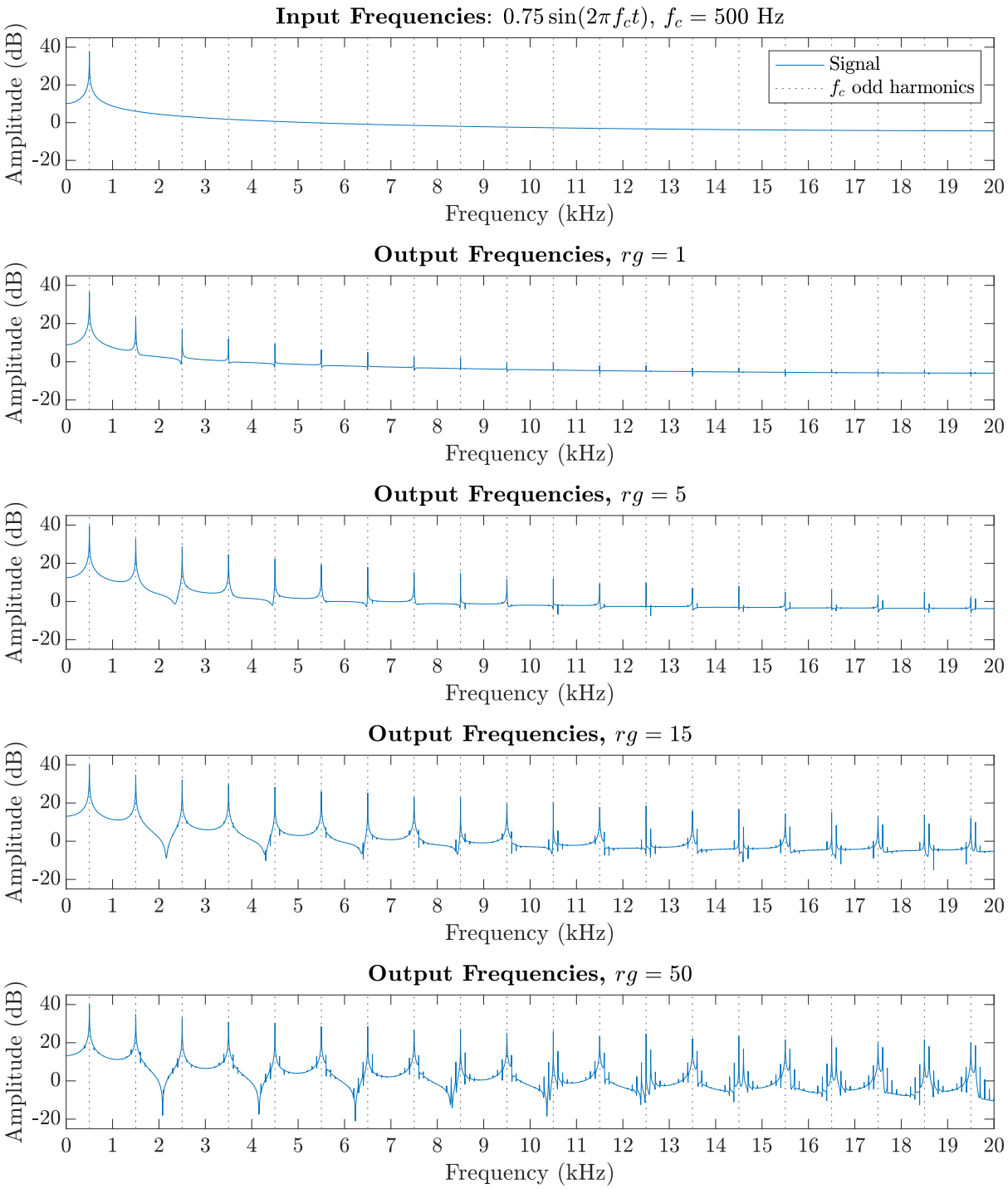


Figure 5.2: As the rg coefficient in Eq. 5.1 increases, the energy of the odd harmonics increases - these are the peaks in the signal. The vertical dotted lines highlight odd multiples of the frequency of the original signal ($f_c = 500$ Hz).

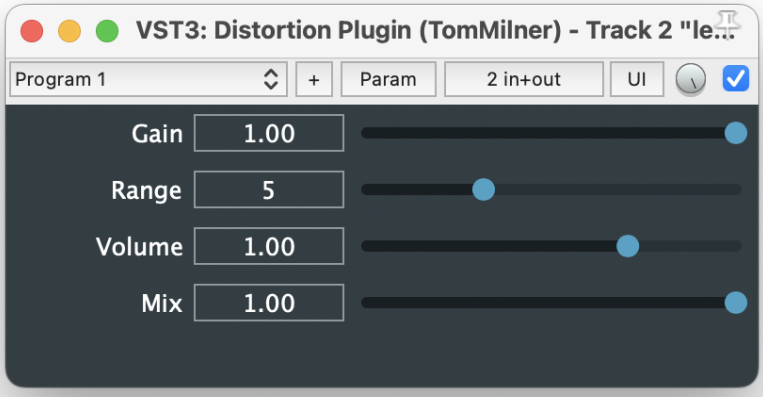


Figure 5.3: The distortion effect implemented as a VST Plugin.

The *Gain* and *Range* parameters are the g and r parameters mentioned earlier (Eq. 5.1).

`Mix` allows the player to mix the input signal with the distorted signal; at 0.5, the output consists of equal parts input and distorted signals.

`Volume` allows the player to adjust the overall volume of the output signal - this is the linear amplification method mentioned at the end of Section 5.1.1.

5.3 Demo Recordings

Demo recordings of the VST plugin can be found in the Public Folder. The `Volume` parameter was used to adjust the output gain to reduce the audio signal clipping, and the `Mix` parameter was left on 1, so the reader can hear just the distortion effect.

5.4 Reflection

Whilst the plugin does implement distortion, it sounds somewhat ear-piercing; granted, the author is not the best guitarist, but the plugin could do with some work before it is ready to sound as good as commercial distortion plugins.

One way of improving the tone could be to run the output through a lowpass filter, to slightly attenuate some of the higher, more ear-piercing frequencies produced. However, the plugin has

satisfied the requirements detailed in the Project Objectives section (Section 1.2) so no further improvements were made.

Chapter 6

Tremolo Effect

The third effect of the project was tremolo. This is a simple effect, but it gives an incredibly distinctive sound. Informally, tremolo works by oscillating the volume of a signal at a rapid speed to give a "shimmering" effect. A very intense usage of tremolo can be heard at the start of the song "How Soon Is Now", by "The Smiths", and a more subtle, classic, tremolo can be heard on the guitar playing the main riff in "Born on the Bayou", by "Creedance Clearwater".

6.1 Analysis

The tremolo works by modulating the amplitude of the input signal with a Low Frequency Osillator (LFO) called the **modulation signal**. This alters the volume of the signal over time.

The modulation signal is as follows [12]:

$$y(t) = (1 - D) + D \times \sin^2(\pi \times t \times f_m) \tag{6.1}$$

$$D = \text{ Modulation Depth, 0..1}$$

$$f_m = \text{ Modulation Frequency (Hz)}$$

The modulation depth D controls how deep the modulation signal can "cut" into the input signal. A high value of D (close to 1) will allow the modulation signal to silence the input signal completely; a low value will not.

The modulation frequency f_m controls how fast the modulation signal oscillates. A higher modulation frequency results in a "choppier", "shimmering" sound, whilst a low modulation

frequency will create a slower "pulsating" sound.

Common values for f_m are between 4-7 Hz [13]. When $f_m < 20$ Hz the tremolo is heard in the time domain, and when $20 < f_m < 70$ Hz, auditory roughness is introduced to the signal [13]. Interestingly, f_m's above 70 Hz can start to be heard as extra frequency components in the output signal.

Figure 6.1, generated in MATLAB, shows a visual representation of tremolo with two different modulation signals.

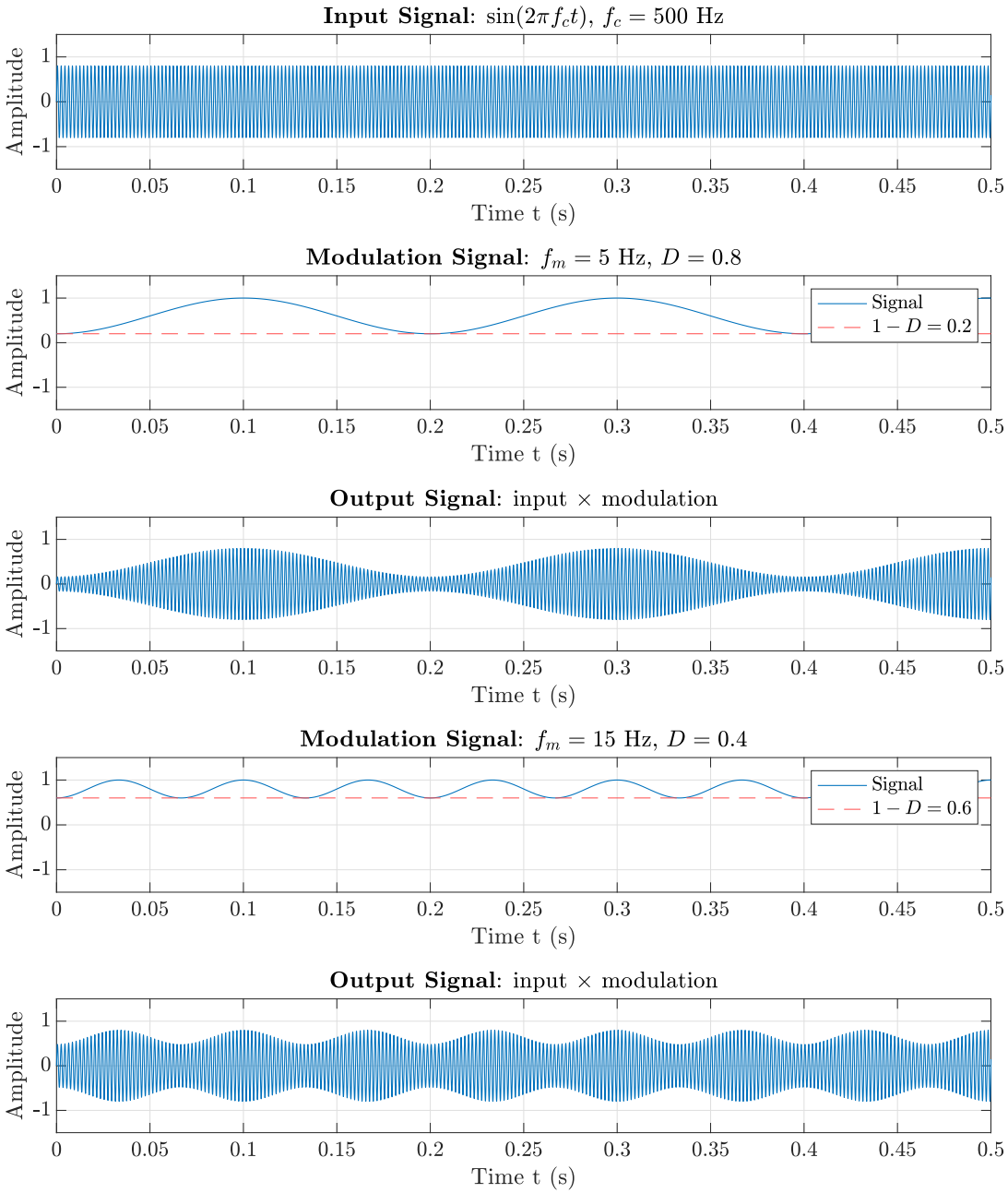


Figure 6.1: Two examples of tremolo modulation.

6.2 Implementation: VST Plugin

The VST plugin for tremolo was more technical to implement than expected. To ensure the implementation was efficient enough for real-time usage, a wavetable was used to cache values for the $\sin^2(\pi t f_m)$ modulation signal calculations (from Eq. 6.1). This prevented the same values from being calculated multiple times, but came at a loss of accuracy to the result; only a discrete set of values can be cached, meaning any values of t not in the cache will be rounded to the nearest value or interpolated (as will be discussed in Section 6.2.2). Thus, this produces an approximation of $\sin^2(t)$.

6.2.1 Wavetable Approximation of $\sin^2(t)$

The wavetable approximation works as follows. $\sin^2(t)$ is a periodic function with period $0...\pi$. When the plugin starts, the wavetable module computes the values of $\sin^2(t)$ over the domain $0...\pi$ and stores the results in a lookup table (this is the wavetable). As we are using the $\sin^2(t)$ function to produce a wave, we will only ever need the result of the *next* value of t; there will never be any arbitrary calls to the function, only calls for sequential values at constant intervals. Therefore, we can implement a function `getNextValue()` to get us the next value from the $\sin^2(t)$ wave. Figure 6.2 represents this visually.

The accuracy of the approximation can be determined by the size of the lookup table, N; the larger the table, the more values of t we can store, and the more accurate our approximation of $\sin^2(t)$ is. Figure 6.2 shows a wavetable of size $N = 32$. This waveform has distinct, noticeable steps in its output, and thus does not give the most accurate approximation of $\sin^2(t)$. The VST plugin uses $N = 128$, which gives a much smoother approximation, and was good enough for this project. A comparison of this wavetable versus the real function can be seen in Figure 6.3.

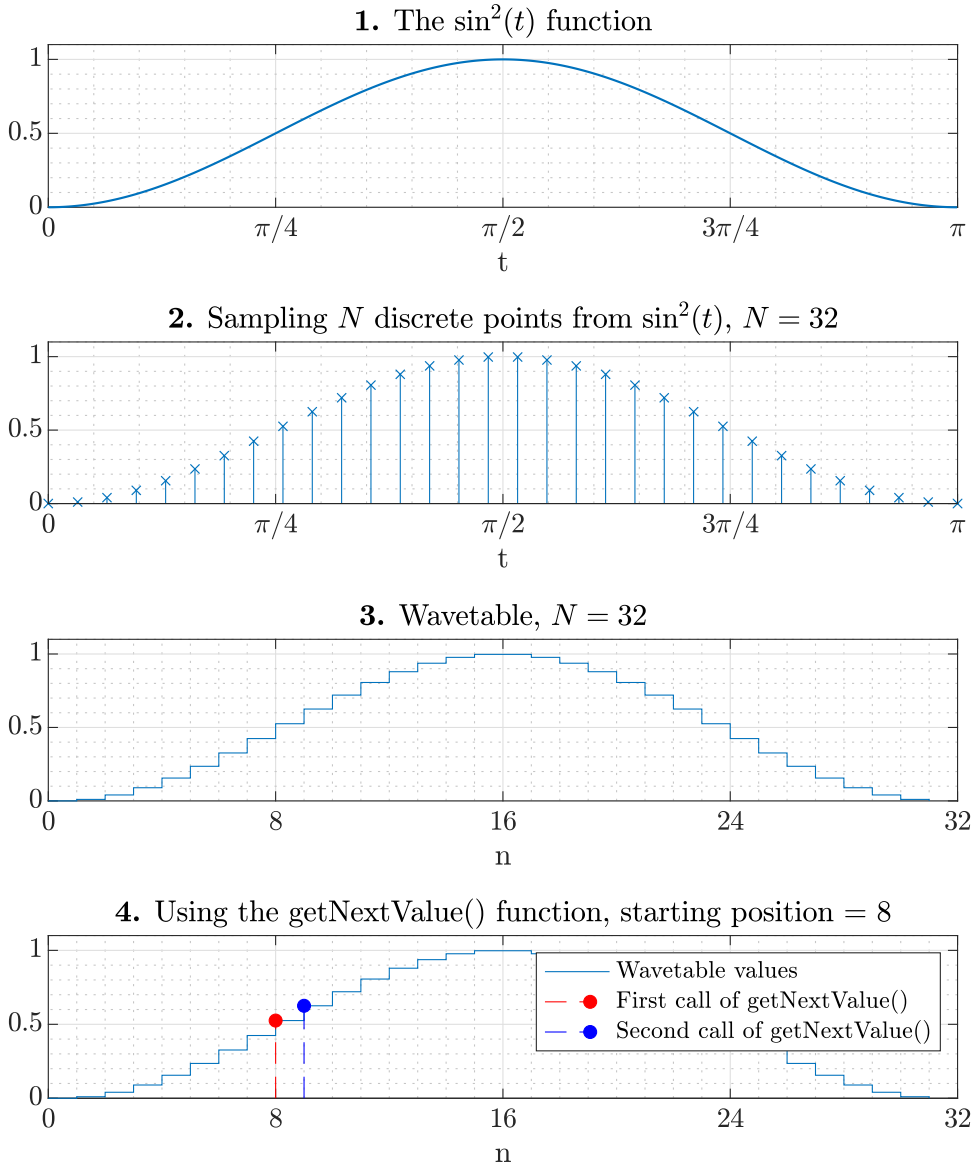


Figure 6.2: Using a wavetable to generate values of $\sin^2(t)$.

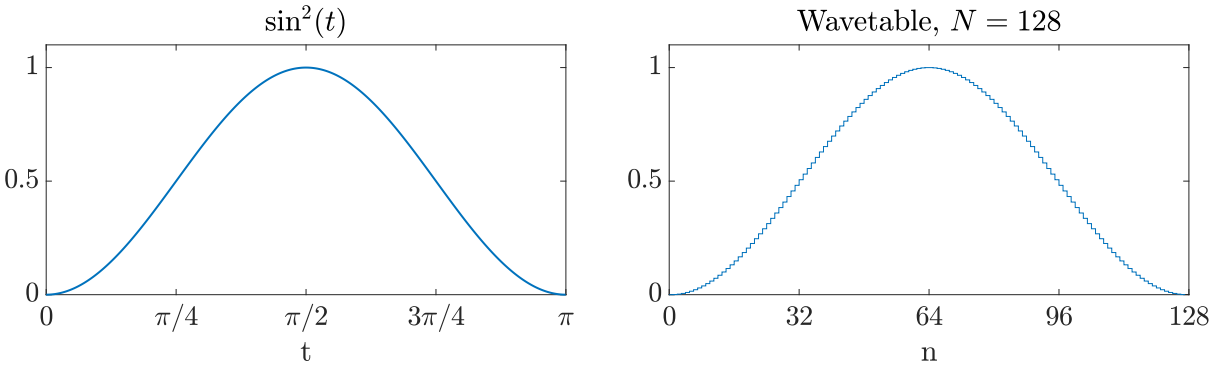


Figure 6.3: The wavetable used in the tremolo VST plugin implementation.

6.2.2 Using the Wavetable for Arbitrary Frequencies

Once the wavetable was set up it could be used for creating outputs of $\sin^2(t)$ at any arbitrary modulation frequency f_m (from Eq. 6.1), i.e.:

$$y(t) = \sin^2(\pi f_m t), \quad f_m \in \mathbb{R}+$$

This was how the LFO modulation signal was produced.

To get a 1 Hz signal from the wavetable we must cycle through all N points (or *indices*) in the wavetable over the course of 1 second. In 1 second, the VST produces F_s samples (F_s being the sampling frequency of the VST). Therefore, in F_s samples we must cycle through all N indices in the wavetable to produce a 1 Hz signal. $N \neq F_s$, so we need to work out the number of indices to advance by, Δ_i, per sample, to reach 1 Hz. This is:

$$\Delta_\text{i} = N/F_s$$

$$\Delta_\text{i} = \text{Number of indices to advance by.}$$

$$N = \text{Size of the wavetable.}$$

$$F_s = \text{Sampling rate.}$$

To produce the waveform at the target frequency, f_m, we can scale Δ_i accordingly:

$$\Delta_\text{f} = f_m \times \Delta_\text{i}$$

It is likely the value of Δ_f is not an integer. In this case, a result is linearly interpolated from

it is neighbouring values in the table.

6.2.3 Final Plugin

The final plugin implementation was as follows:

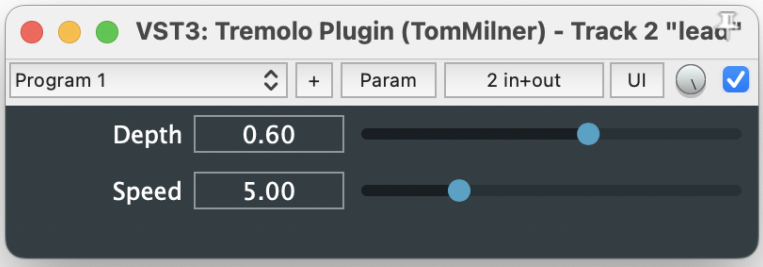


Figure 6.4: The final tremolo VST plugin implementation.

The `Depth` parameter allows the user to change D, and the `Speed` parameter allows the user to change f_m (both from Eq. 6.1)[1]. MATLAB and VST code can be found in the Public Folder.

6.3 Demo Recordings

Recordings of the MATLAB program and VST plugin can be found in Public Folder.

6.4 Reflection

Whilst the MATLAB analysis of the tremolo effect was fairly trivial, the VST plugin was harder (and more enjoyable) to implement than expected due to the wavetable synthesis method explained earlier.

The plugin is a success; it sounds good and is easy to use. Further improvements could be to introduce different shapes for the modulation signal, e.g. triangle or square waves, alongside the $\sin^2(t)$ shape.

[1] f_m is given the name *Speed* in the plugin because it is common terminology for the frequency of a tremolo.

Chapter 7

Reverb Effect

As mentioned in the Introduction (Chapter 1), the reverb effect emulates the sound of the guitar in a large echoing room and is an incredibly popular effect that is used across multiple genres. It works by simulating the natural reflections of sound in a physical space to create a sense of depth and space in an audio recording.

Some examples of songs that are *drenched* in reverb are "Heaven or Las Vegas" by the "Cocteau Twins", "When the Sun Hits" by "Slowdive", and "Subterranean Homesick Alien" by "Radiohead".

7.1 Analysis

Reverb effects processors come in different forms; many physical and electronic contraptions have been devised to try and simulate the dynamics of sound echoing around a room (a particularly interesting method out of the scope of this project is *plate reverb* - see [14]).

As this is a DSP project, we explore some software approaches to reverb, before creating our implementation in MATLAB.

There are two main approaches for implementing DSP reverberation:

1. Convolutional Reverb
2. Algorithmic Reverb

Convolutional Reverb

This approach "attempts to simulate exactly the propagation of sound from the source to the listener in a given room" [15]. A user supplies the reverb effects unit with a digital audio signal and an impulse response recording of an environment. The reverb is produced by convolving the digital audio signal with the impulse response making the audio sound like it is playing in the environment.

According to Zölzer, convolutional reverb is the "most faithful" [16] reverberation method available. It is, however, "computationally expensive and inflexible", and there is "no easy way to achieve real-time parametric control of the perceptual characteristics of the resulting reverberation" [15]. Thus, convolutional reverberation was not chosen for this project.

Algorithmic Reverb

The other approach to artificial reverb is to algorithmically model room acoustics to produce reverberation-like effects. The paper "Natural Sounding Artificial Reverberation", by M. R. Schroeder [17] details one of the first algorithmic implementations of reverb. The implementation uses two types of digital filters: *feedback comb filters* and *allpass filters*.

7.1.1 Feedback Comb Filters (FBCFs)

A Feedback Comb Filter (FBCF) has a frequency response resembling a comb, similar to Feedforward Comb Filters in Chapter 4, but using zeros as opposed to poles. Equation 7.1 shows the transfer function of a simple FBCF.[1]

$$H(z) = \frac{z^{-m}}{1 - gz^{-m}} \tag{7.1}$$

$$g = \text{ Gain}$$

$$m = \text{ Delay (samples)}$$

The variables used to represent the *gain* and *delay* terms have changed since Eq. 4.5 to better match the corresponding literature; where gain was a in Eq. 4.5, it is now g, and the delay constant D is now m.

[1] Simpler FBCF designs exist such as $H(z) = \frac{1}{1-gz^{-m}}$, but we use the design used by Schroeder to stay consistent with the literature.

The frequency response of the filter is Eq. 7.2.

$$H(e^{j\omega}) = \frac{e^{-j\omega m}}{1 - ge^{-j\omega m}} \tag{7.2}$$

There are **peaks** in the frequency response every F_D Hz, where $F_D = F_s/m$ (just like Eq. 4.4, F_s = sampling rate). Between the peaks, there are **troughs**.

The magnitudes of the peaks and troughs are shown in Eq. 7.3. The calculations for these magnitudes are almost identical to that of the FFCF (Eq. 4.1.2), so have been omitted.

$$\text{Magnitude of the Peaks:} \quad \frac{1}{1-g} \tag{7.3}$$

$$\text{Magnitude of the Troughs:} \quad \frac{1}{1+g} \tag{7.4}$$

Example

We can explore an example filter with a sample rate $F_s = 44100$, delay $m = 5$ and gain $g = 0.5$ (Eq. 7.5).

$$H(z) = \frac{z^{-5}}{1 - 0.5z^{-5}} \tag{7.5}$$

$$\text{Fundamental Frequency:} \quad F_D = \frac{F_s}{m} = \frac{44100}{5}$$
$$= 8.82 \text{ kHz}$$

$$\text{Frequencies of Peaks:} \quad kF_d, \quad k = 0, 1, 2, ..$$
$$\longrightarrow 0, 8.82, 17.64, ... \text{ kHz}$$

$$\text{Frequencies of Troughs:} \quad \frac{(2k+1)F_d}{2}, \quad k = 0, 1, 2, ..$$
$$\longrightarrow 4.41, 13.23, ... \text{ kHz}$$

$$\text{Magnitude of the Peaks:} \quad \frac{1}{1-a} = \frac{1}{1-0.5} = 2$$

$$\text{Magnitude of the Troughs:} \quad \frac{1}{1+a} = \frac{1}{1+0.5} = 0.667$$

Figure 7.1 shows the frequency response of the filter, where the peaks and troughs can be seen at the frequencies and magnitudes just calculated. Figure 7.2 shows the pole-zero plot of the filter, and Figure 7.3 shows the impulse response of the filter.

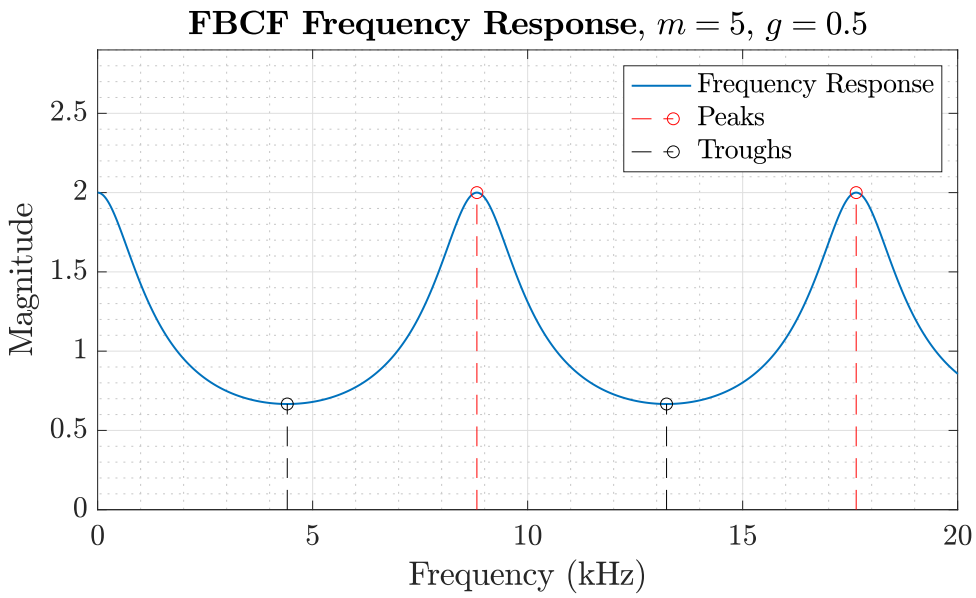


Figure 7.1: FBCF Eq. 7.5 frequency response.

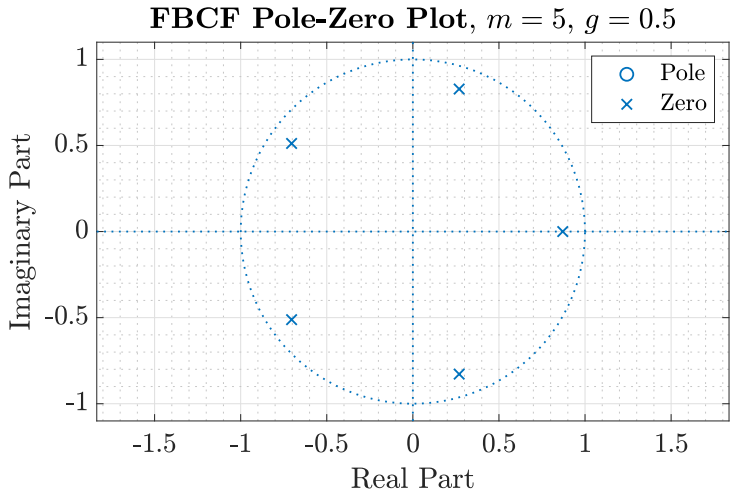


Figure 7.2: FBCF Eq. 7.5 pole-zero plot.

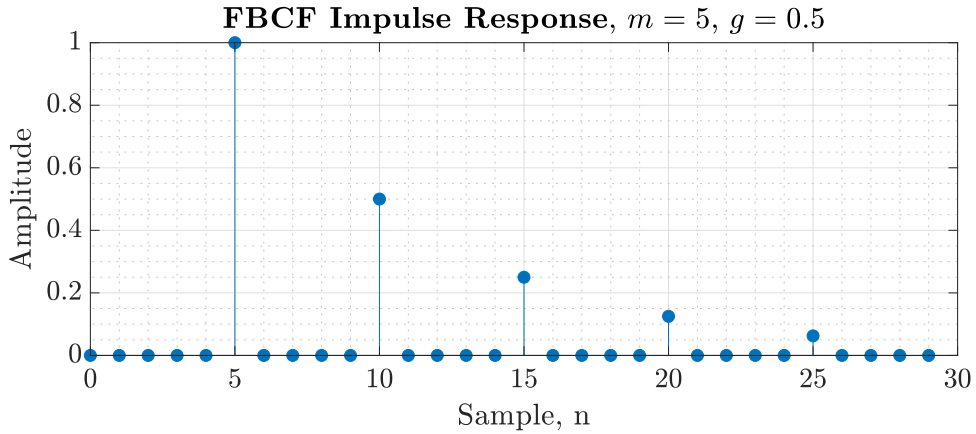


Figure 7.3: FBCF impulse response.

A signal flow diagram of the FBCF can be seen later in Figure 7.6 in Section 7.1.3.

7.1.2 Allpass Filters

Allpass filters are filters that have a completely flat frequency response - they let all frequencies pass at unity gain (Figure 7.4).

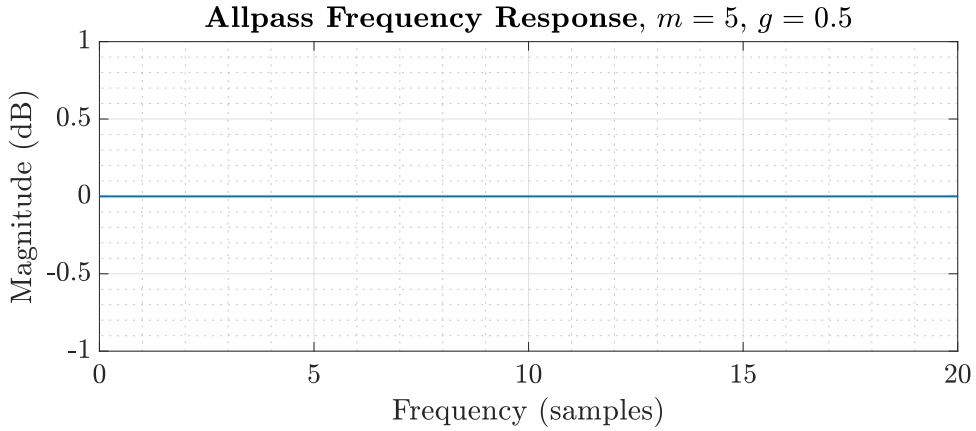


Figure 7.4: Allpass filter frequency response.

This comes from the transfer function shown in Equation 7.6 [18].

$$H(z) = \frac{1 - gz^m}{z^m - g} = \frac{z^{-m} - g}{1 - gz^{-m}} \tag{7.6}$$

Which gives the frequency response (setting $z = e^{j\omega}$):

$$\begin{aligned} H(e^{j\omega}) &= \frac{e^{-j\omega m} - g}{1 - ge^{-j\omega m}} \\ H(e^{j\omega}) &= e^{-j\omega m}\frac{1 - ge^{+j\omega m}}{1 - ge^{-j\omega m}} \\ |H(e^{j\omega})| &= 1 \times 1 = 1 \end{aligned} \tag{7.7}$$

Eq. 7.7 shows that the frequency response is unity: the first term ($e^{-j\omega m}$) in the product has unity magnitude, and the second term is a quotient of complex conjugates ($\frac{1-ge^{+j\omega m}}{1-ge^{-j\omega m}}$), which also has unity magnitude.

The poles and zeros of the filter are mirror images of each other, as can be seen in Figure 7.5.

A signal flow diagram of the allpass filter can be seen later in Figure 7.6 in Section 7.1.3.

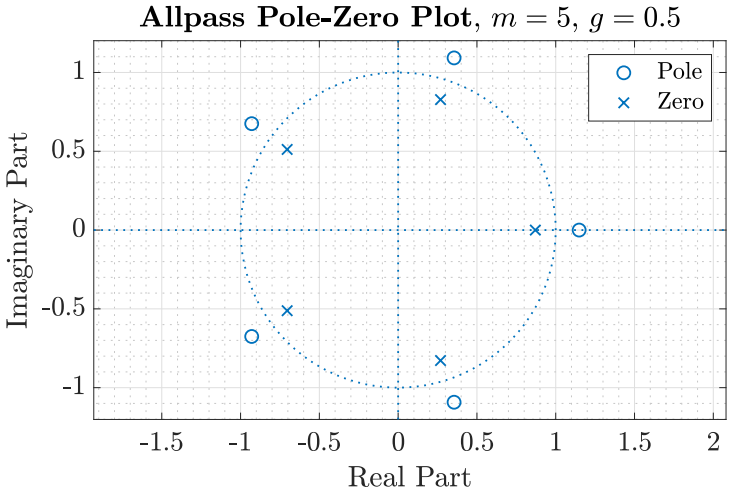


Figure 7.5: Allpass filter pole-zero plot.

7.1.3 How the Schroeder Reverb Works

This project uses William Gardner's Schroeder reverb implementation, shown in Figure 7.6.

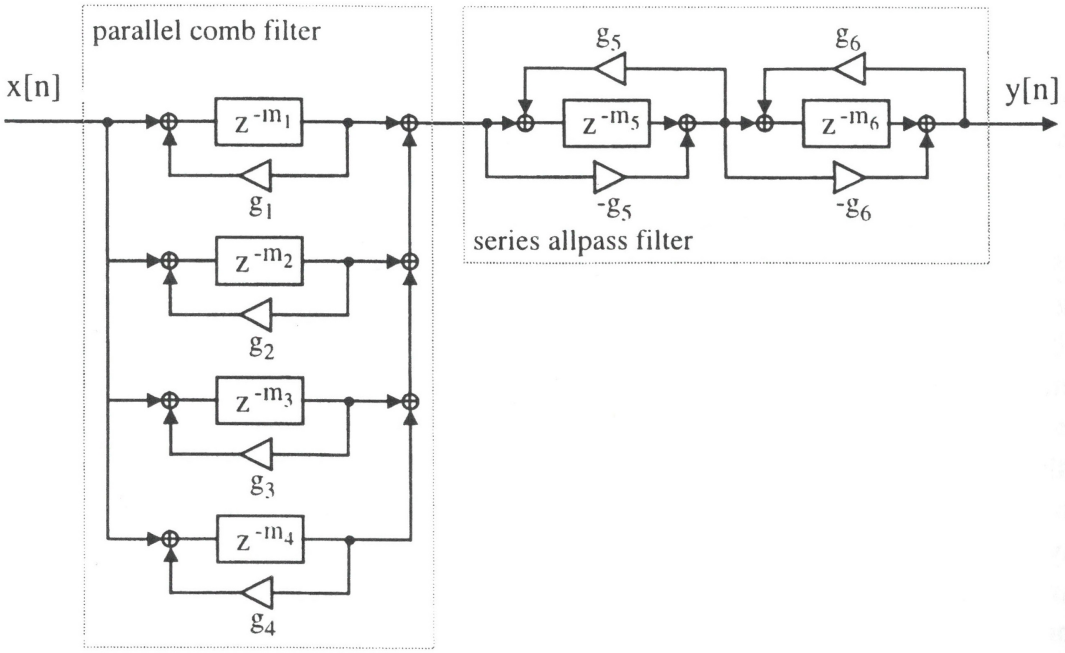


Figure 7.6: Gardner's version of the Schroeder reverb [15].

There are two sections to the reverb: the parallel comb filter section, and the series allpass filter section.

Parallel Comb Filter Section

The role of the parallel comb filter bank is to approximately simulate the frequency response of a room with a reverberation time (T_r) of one-second [17]. The reverberation time, T_r, is the time taken for the reverberation sound levels to decay by 60 dB [17]. A room with $T_r = 1$ s has roughly 15 large frequency response peaks every 100Hz [17]. An FBCF with a delay time of 40 ms produces 4 response peaks per 100Hz. Thus, putting 3 to 4 FBCFs in parallel, with incommensurate delays [2], will produce 12 - 16 peaks every 100Hz - effectively simulating the frequency response of the room. If the delays are not incommensurate, the frequency responses will superimpose on each other, and the response will result in an uneven comb.

Cascaded Allpass Filter Section

The right-hand side of the diagram consists of two cascaded allpass filters to increase the *echo density* of the reverb [17]. Echo density is the "number of echos per second at the output [...] for a single pulse at the input". As discussed in Section 7.1.2, the allpass filters do not modify

[2]Their ratio can "not be expressable as a ratio of whole numbers" [19].

the frequency components of the signal at all - the frequency response of the bank of comb filters is preserved.

7.2 Implementation: Gardner's Schroeder Reverb, in MATLAB

Gardner suggests that the delays m_1 to m_4 of the parallel comb filters are set such that the ratio of the largest to the smallest is 1.5; "a range of 30 to 40ms"[15] is suggested.

Gains g_1 to g_4 are set according to Eq. 7.8 [15], which ensures the level of the reverb is -60 dB by the end of T_r seconds.

$$g_i = 10^{-3m_iT/T_r} \tag{7.8}$$

$$i = 1, 2, 3, 4$$

$$g_i = \text{Gain } i$$

$$m_i = \text{Delay } i$$

$$T_r = \text{Reverberation Time (seconds)}$$

$$T = 1/F_s = \text{Sampling Period (seconds)}$$

The allpass delays m_5 and m_6 are recommended to be 5 and 1.7 ms, and the allpass gains g_5 and g_6 are recommended to be 0.7 ms [15].

The algorithm was implemented in MATLAB (and can be found in the Public Folder), and demo recordings were made using two input signals: a sine wave burst, and a guitar sample.

7.3 Demo Recordings

Demo recordings of both the sine wave burst and guitar sample inputs can be found in the public folder (Appendix B), in the `Effect Demos/Reverb/Audio/MATLAB` folder.

`Effect Demos/Reverb/Audio/DAW Recordings` contains recordings of the guitar sample being used with a backing track.

7.4 Reflection

7.4.1 Sine Wave Burst

Figure 7.7 shows the effects of multiple reverberation times on a 200 ms sine wave input. It can be seen that Gardner's formula for gain values (Eq. 7.8) based on T_r is effective; the amplitude of the signal decays until reaching ≈ -60 dB at T_r. The figure also shows that larger values of T_r have a much smoother decay; the decay of signal `(1)` has an incredibly sharp (unnatural sounding) cutoff, whilst signals `(3)` and `(4)` are smoother and sound more natural.

Figure 7.8 shows that *early reverberations* are larger for smaller values of T_r, and Figure 7.9 shows that *late reverberations* are smaller for larger values of T_r. The lower values of T_r simulate a small room with close-together walls (and fast soundwave reflections), whilst higher values of T_r simulate a larger room, with slower reflections.

It can be heard from the recordings that the reverb for impulsive signals does not sound natural; distinct echoes can be heard in the output, producing a "fluttering" effect. This is a well-known flaw with the Schroeder algorithm [15], and is fixed by Moorer's reverberator implementation [20]. These fixes are out of the scope of this project.

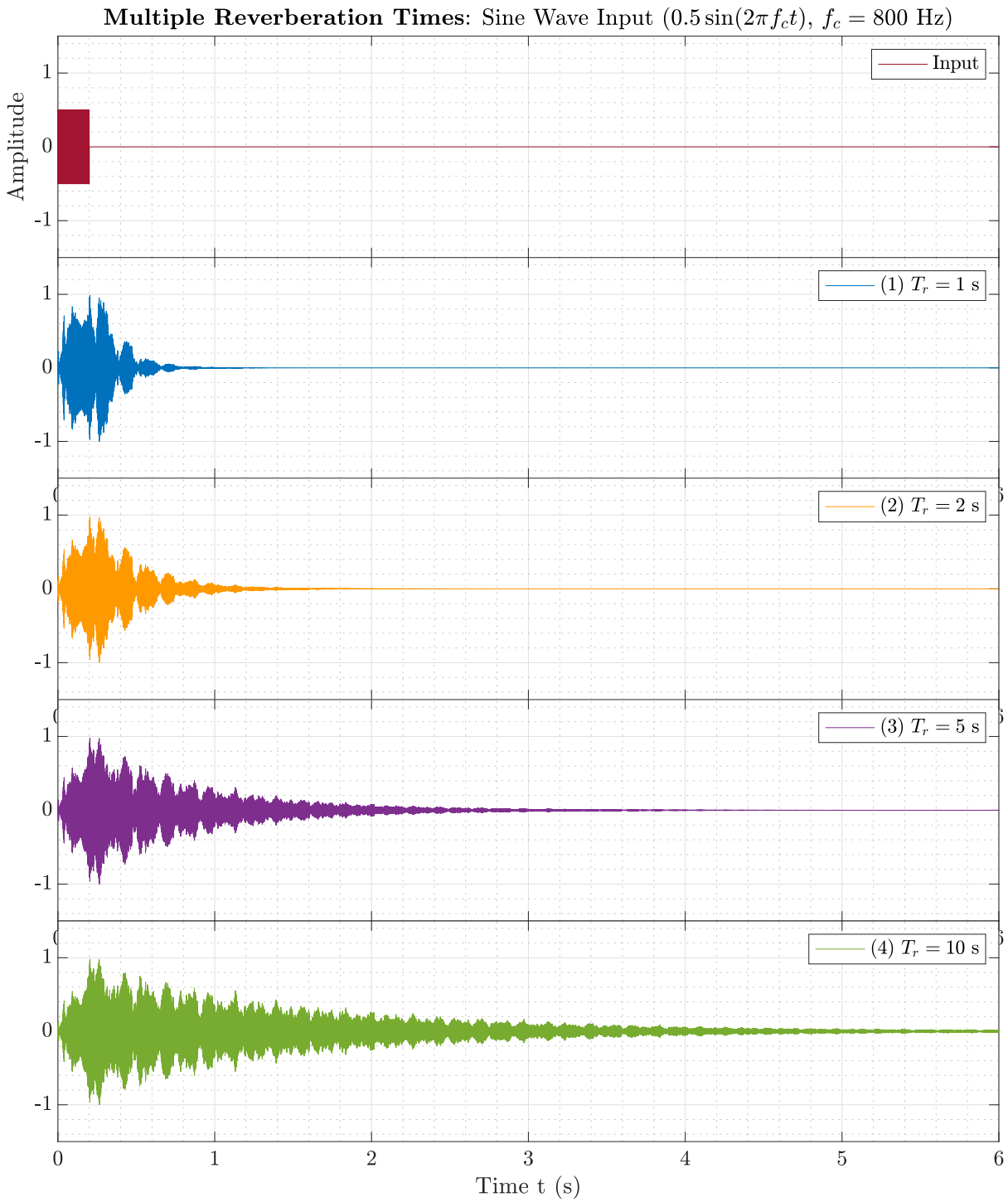


Figure 7.7: Reverb on a sine wave burst.

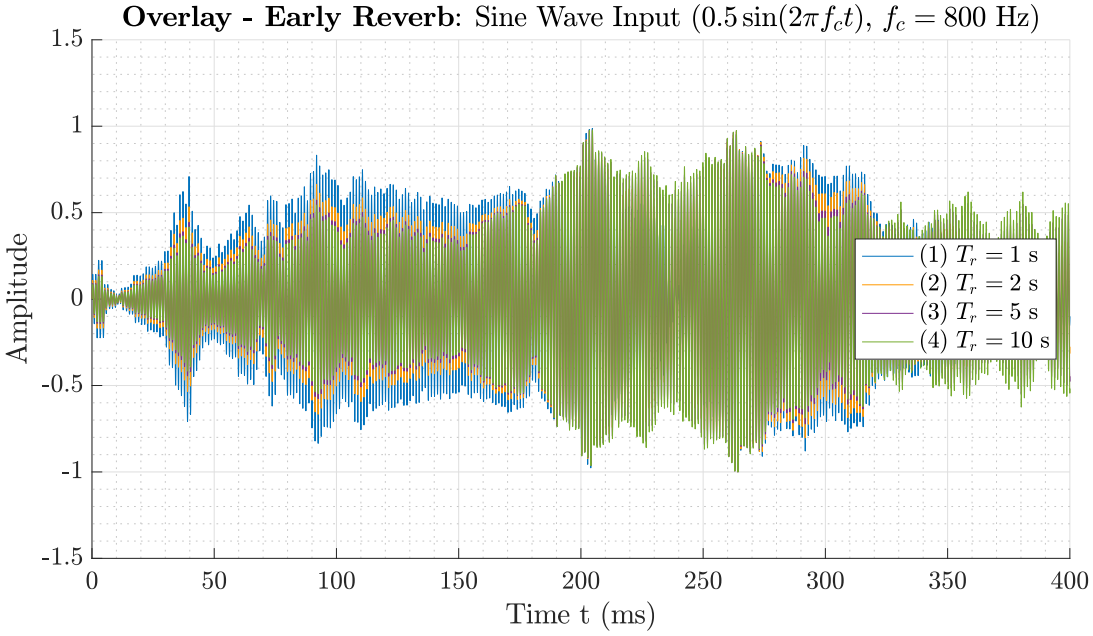


Figure 7.8: Early reverberations, sine wave input.

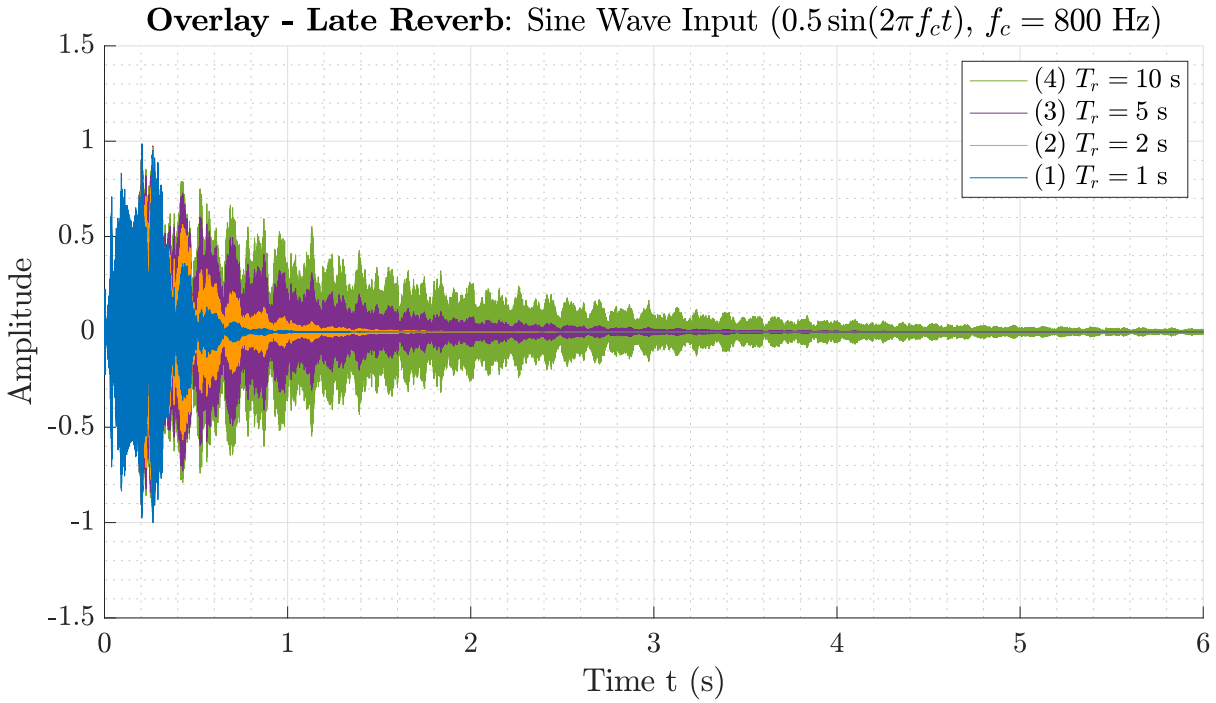


Figure 7.9: Late reverberations, sine wave input.

7.4.2 Guitar Sample

Figures 7.10, 7.11, and 7.12 show the reverb plots for a recorded guitar signal input. From Figure 7.10 it can be seen that higher values of T_r "smear" out the signal - rapid changes in volume are less distinct (especially visible at $t = 7.5$ s). Overall, the reverb implementation

adds a sense of depth and space to the guitar input and is considered a success.

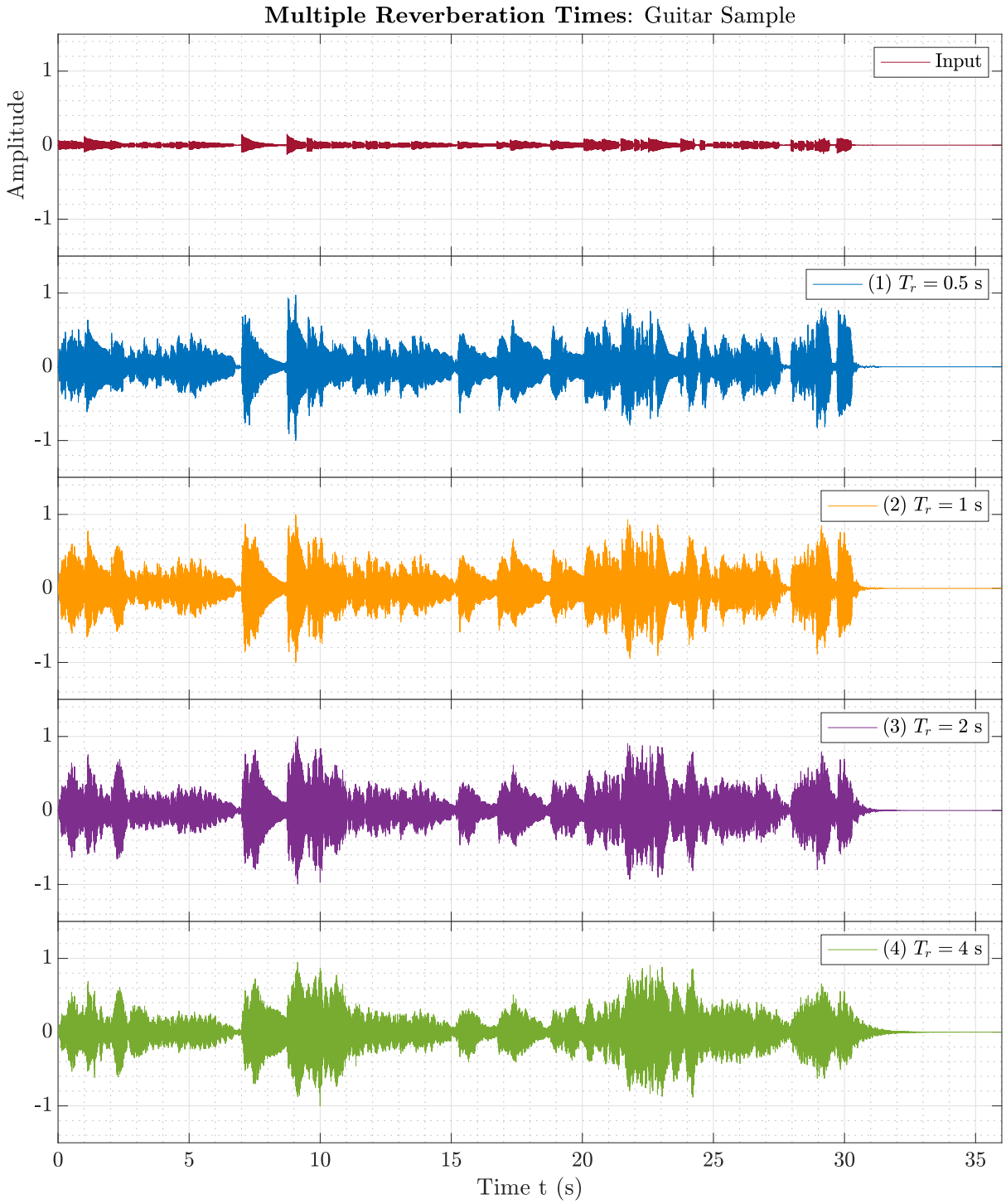


Figure 7.10: Reverb on a guitar sample.

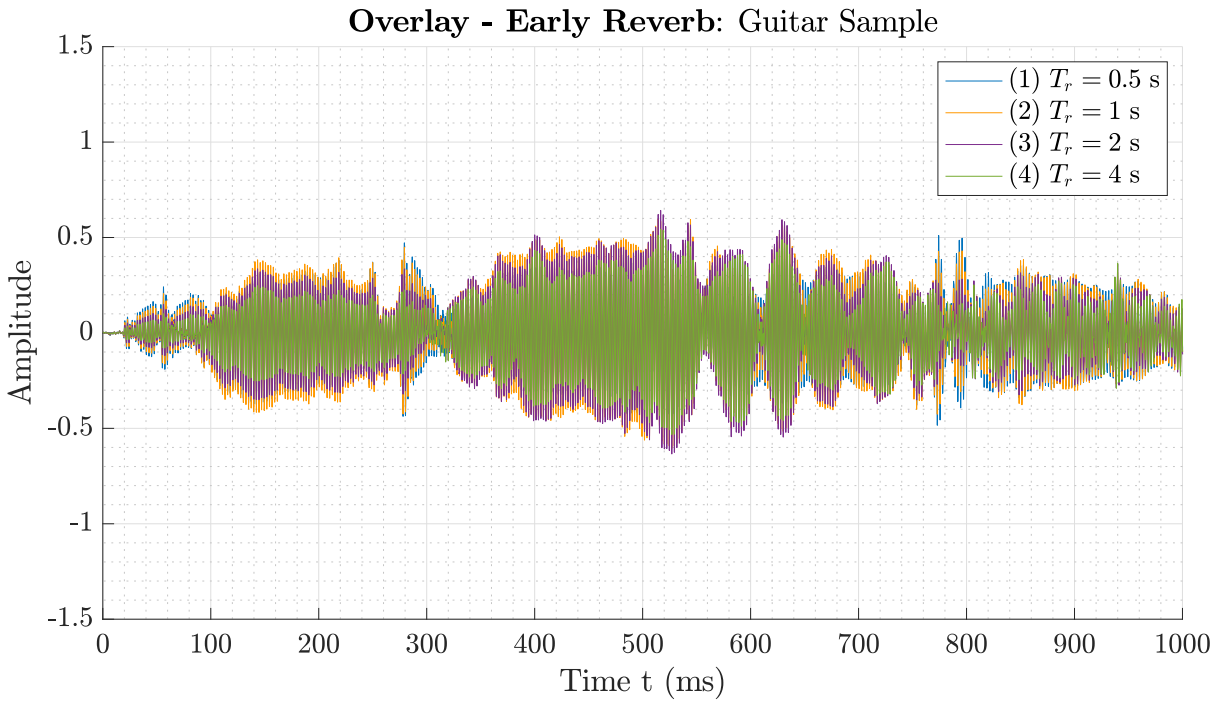


Figure 7.11: Early reverberations, guitar sample input.

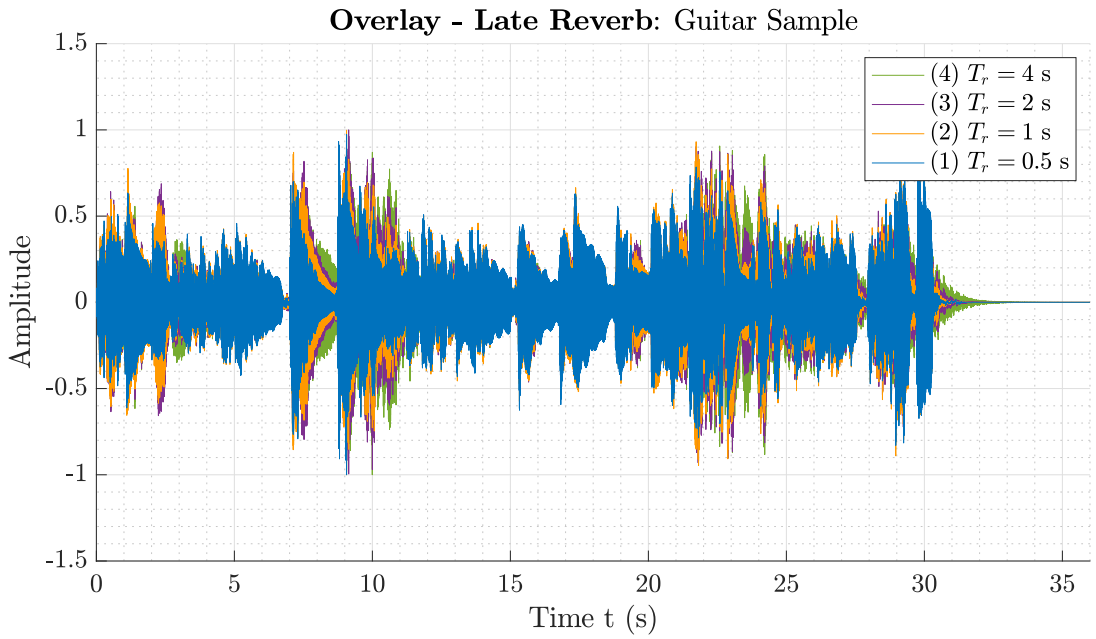


Figure 7.12: Late reverberations, guitar sample input.

Chapter 8

Vocoder Effect

A vocoder (technically, a *channel vocoder*) is a device that blends two audio signals to make one sound like the other. It's used heavily by bands like "Daft Punk", who use it to create robotic-sounding voices for their songs (one example is "Harder, Better, Faster, Stronger"[21]).

8.1 Analysis

Figure 8.1 is a recreation of the channel vocoder found in the DAFX book [22].

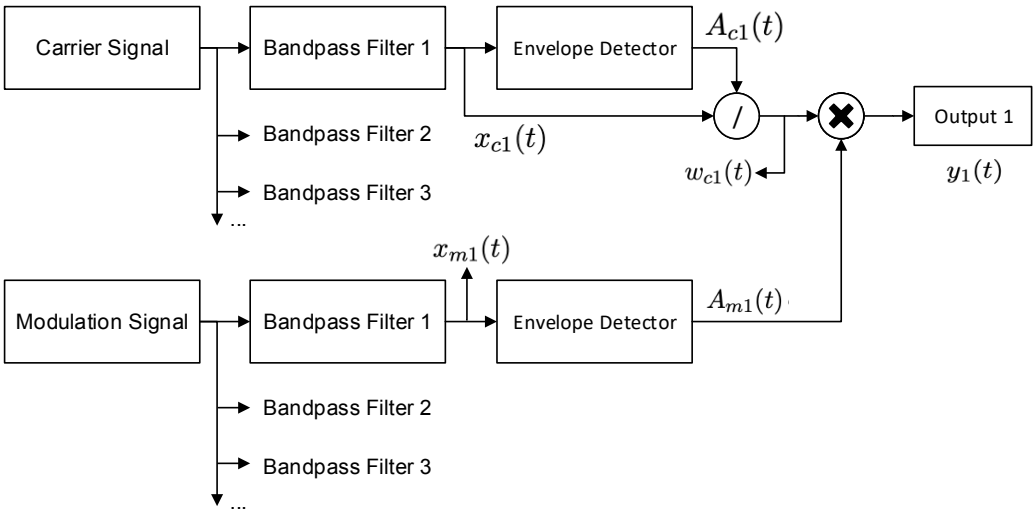


Figure 8.1: Vocoder Signal Diagram [22].

The **carrier** signal is usually a "buzzing" sound (i.e. a squarewave or sawwave), and the **modulation** signal is usually a voice input.

Each signal is split into N different frequency bands using bandpass filters. The signals produced by the bandpass filters are as follows:

$$x_{ci}(t) = A_{ci}(t)\cos(2\pi f_i t + \varphi_{ci}(t)) \tag{8.1}$$

$$x_{mi}(t) = A_{mi}(t)\cos(2\pi f_i t + \varphi_{mi}(t)) \tag{8.2}$$

$$i = 1, 2, 3, ..., N$$

$x_{ci}(t) =$ Band i of the carrier signal, produced by bandpass filter i.

$x_{mi}(t) =$ Band i of the modulation signal, produced by bandpass filter i.

$A_{ci}(t), A_{mi}(t) =$ Envelopes of the carrier $x_{ci}(t)$ and modulation $x_{mi}(t)$.

$\varphi_{ci}(t), \varphi_{mi}(t) =$ Phases of $x_{ci}(t)$ and $x_{mi}(t)$.

$f_i =$ Center frequency of band i.

In each band, the carrier and modulation signal envelopes are isolated using envelope detectors. The carrier signal is then divided by its envelope, removing its amplitude information, and leaving just the frequency and phase components ($w_{ci}(t)$).

$$w_{ci}(t) = \frac{x_{ci}(t)}{A_{ci}(t)} = \frac{A_{ci}(t)\cos(2\pi f_i t + \varphi_{ci}(t))}{A_{ci}(t)} = \cos(2\pi f_i t + \varphi_{ci}(t))$$

(To prevent division by zero errors a marginal offset $\epsilon = 0.001$ is added to $A_{ci}(t)$). The resulting signal is multiplied by the modulator's envelope to produce the output signal for that band.

$$\begin{aligned} y_i(t) &= A_{mi}(t) \times w_{ci}(t) \\ &= A_{mi}(t)\cos(2\pi f_i t + \varphi_{ci}(t)) \end{aligned}$$

The output signals of every band are summed (not shown in the diagram) to produce the vocoded output signal.

$$y_{\text{out}}(t) = \sum_{i=1,2,3,...}^{N} y_i(t)$$

8.2 Implementation: Channel Vocoder in MATLAB

The vocoder was implemented in MATLAB, based on the channel vocoder found in [22]. The following sections detail the implementation of specific parts of the vocoder.

8.2.1 Calculating N

To calculate N (the number of frequency bands required), we need the size of each frequency band.

Smaller bands will result in a more accurate output signal due to the higher frequency resolution, but will lead to more bands in total and thus more processing time.

The compromise reached was making each band a third of an octave. An octave is defined as all frequencies between an arbitrary frequency f, and its double, $2 \times f$. Thus, a third of an octave is $2^{1/3}$.

We can work out how many thirds of an octave lie between two frequencies, and thus our value of N, using the following formula:

$$N = \left\lfloor \log\left(\frac{f_{\max}}{f_{\min}}\right) \div \log(2^{1/3}) \right\rfloor$$

$$f_{\max} = \text{The highest frequency input to the vocoder.}$$

$$f_{\min} = \text{The lowest frequency input to the vocoder.}$$

The value is floored (instead of ceilinged) so we are not wasting processing on a band of frequencies mostly outside of the target range.

The MATLAB implementation uses the full range of human hearing [23]:

$$f_{\min} = 20 \text{ Hz}, \quad f_{\max} = 20 \text{ kHz}$$

Thus, $N = 29$.

8.2.2 Bandpass Filters: Chebyshev Filters

Chebyshev Type 1 filters were used for the bandpass filters. These are IIR filters that introduce a ripple of 3 dB in the passband [22]. As they are common, well-established filters, they will

not be analysed in this report.

8.2.3 Envelope Detection: Square-Law Envelope Detector

The square law envelope detector was used for envelope detection and can be seen in Figure 8.2.

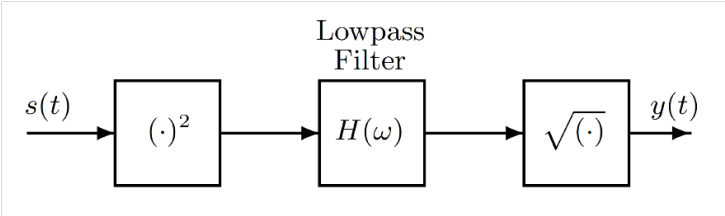


Figure 8.2: Square-Law Envelope Detector [24].

We can see how the envelope detector works by passing in a simple signal $s(t)$ with an envelope $A(t)$.

$$s(t) = A(t)\cos(2\pi f_0 t)$$

$$A(t) = 2 + \sin(2\pi f_1 t)$$

$$f_0 = 20 \text{ Hz}, \quad f_1 = 2 \text{ Hz}$$

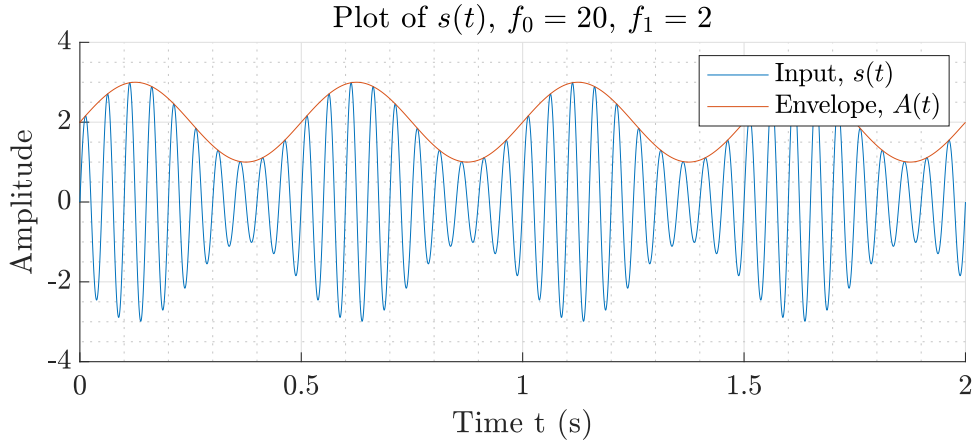


Figure 8.3: Envelope Detector example input.

The purpose of the envelope detector is to extract $A(t)$.

The first stage, $(.)^2$, squares the input signal.

$$\begin{aligned} s^2(t) &= A^2(t)\cos^2(2\pi f_0 t) \\ &= A^2(t) \times \frac{1-\cos(2\pi(2f_0)t)}{2} \\ &= 0.5A^2(t) - 0.5A^2(t)\cos(2\pi(2f_0)t) \end{aligned} \tag{8.3}$$

As Eq. 8.3 shows, after squaring, the $0.5A^2(t)$ term is isolated - this is almost the envelope of the original signal. The second term $0.5A^2(t)\cos(2\pi(2f_0)t)$ is now twice the frequency of the original signal $s(t)$.

To get just the envelope signal $A(t)$ from Eq. 8.3, a lowpass filter can be used to remove the second term - this is $H(\omega)$ in Figure 8.2. The gain factor of 0.5 can be ignored, as in practice the signal gain is dominated by the LPF (as seen in Section 8.2.4).

$$\begin{aligned} \text{LPF Output} &= \cancel{0.5}A^2(t) - \cancel{0.5A^2(t)\cos(2\pi(2f_0)t)} \\ &= A^2(t) \end{aligned}$$

Finally, the LPF output can be square-rooted to produce the original envelope of the signal:

$$y(t) = A(t)$$

Frequency Limitations

There is a limit to the frequencies of envelopes this filter can detect. If the envelope signal $A(t)$ has bandwidth W, then $A^2(t)$ will have bandwidth $2W$ [1]. The second term in the equation (Eq. 8.3) is centred around $2f_0$, and will also have a bandwidth $2W$ (as it is a product of $A^2(t)$). Therefore, the spectrum of the first and second terms will overlap if $2W \geq 2f_0 - 2W$. This can be simplified to $2W \geq f_0$, which reveals the limits of this method: to ensure detection, the frequency of the carrier must be more than twice the bandwidth of the envelope signal ($2W < f_0$).

[1] Squaring in the time domain convolves the signal with itself in the frequency domain, doubling its bandwidth.

8.2.4 Lowpass Filter (LPF)

The envelope detector uses a lowpass filter; the implementation used was a 2nd order IIR filter. The linear difference equation of the LPF is as follows:

$$y(n) = x(n) + 2ry(n-1) - r^2 y(n-2), \quad r = 0.99$$

This produces the transfer function (and frequency response):

$$H(z) = \frac{z^2}{(z-r)^2} \implies |H(e^{j\omega})| = \left| \frac{e^{2j\omega}}{(e^{j\omega}-r)^2} \right|, \quad 0 \leq |\omega| \leq \pi$$

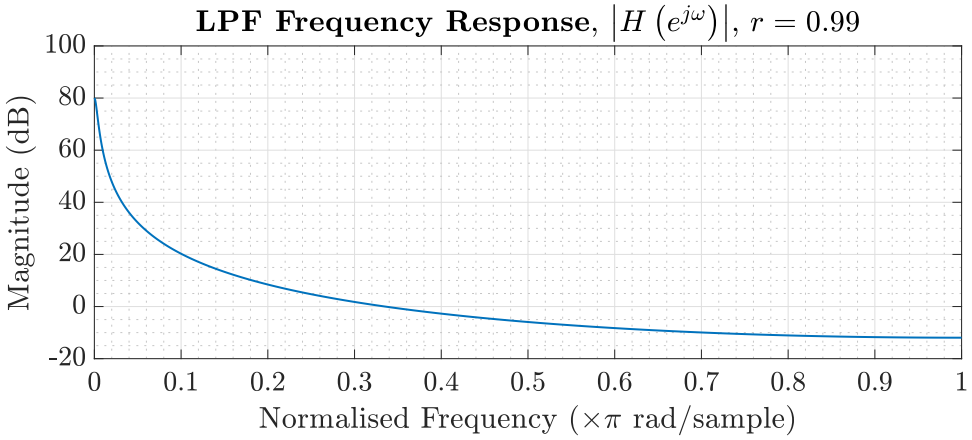


Figure 8.4: LPF Frequency Response

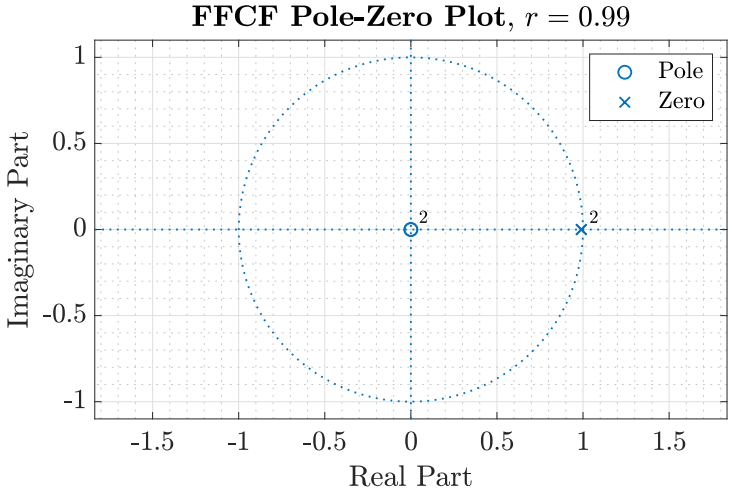


Figure 8.5: LPF Pole-Zero Diagram

The frequency response of the transfer function (Figure 8.4) shows there can be as much as 80 dB gain to signals passing through the LPF - this is the dominating gain mentioned at the end

of 8.2.3.

8.2.5 Envelope Signal Gain Cancellation

As shown, the LPF adds gain to the envelope detector's output signal. This means gain will be added to the signal envelopes $A_{ci}(t)$ and $A_{mi}(t)$.

Whilst this seems like it could be a problem, it is already taken care of; $A_{mi}(t)$ and $A_{mi}(t)$ will have a similar gain (K) from the LPF, and so when they are divided out, the gain is approximately cancelled. This can be seen from the vocoder equation, rewritten:

$$y_i(t) = KA_{mi}(t)\frac{x_{ci}(t)}{KA_{ci}(t)} = x_{ci}(t)\frac{KA_{mi}(t)}{KA_{ci}(t)} = x_{ci}(t)\frac{A_{mi}(t)}{A_{ci}(t)} = A_{mi}(t)\cos(2\pi f_i t + \varphi_{ci}(t))$$

$$K = \text{Gain added by LPF.}$$

8.2.6 Frequency Band Plots

Once the vocoder was built, plots were made of the modulation, carrier, and output signals (Figure 8.6) to verify it was working.

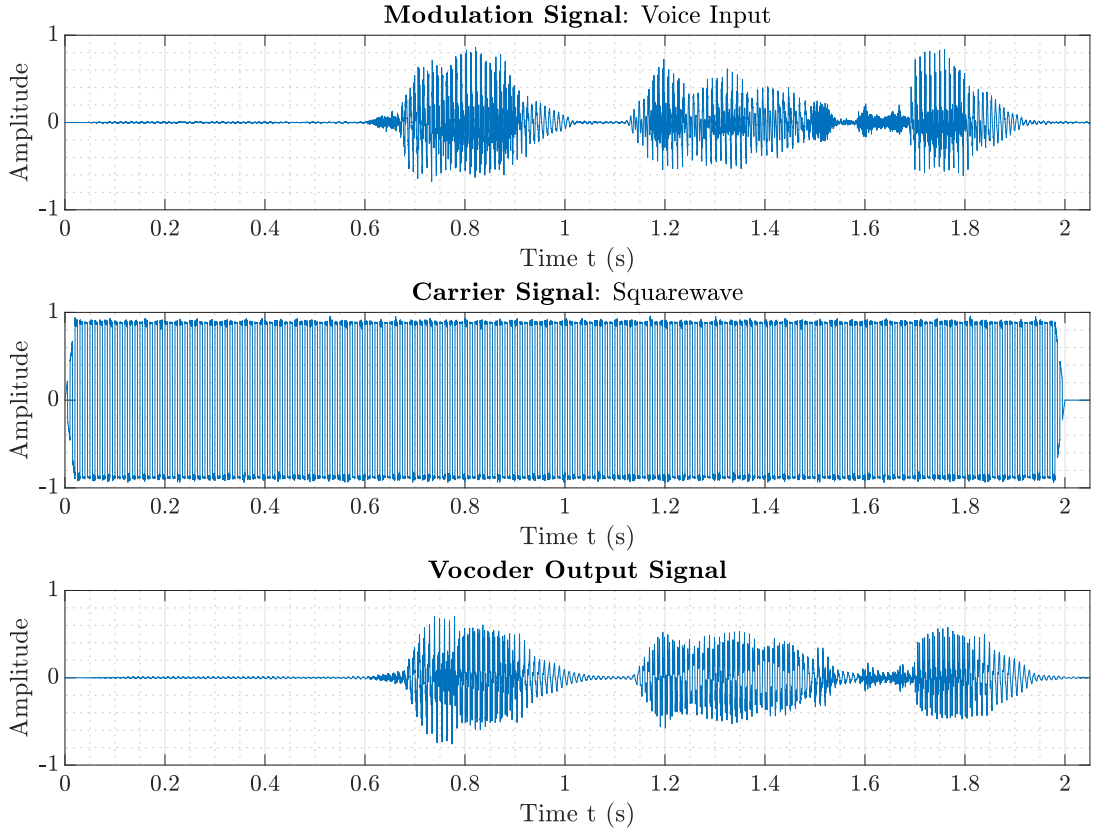


Figure 8.6: Vocoder Input and Output Signals

The modulation signal was a voice recording, and the carrier signal was a squarewave. It was hard to verify the vocoder was working from input and output plots alone, so plots were made for each of the frequency bands. Two arbitrary bands, 8 and 18, are shown in Figures 8.7 and 8.8, and plots for other bands can be found in the Public Folder (Appendix B), in the `Vocoder/Example 1 - Squarewave/Plots` directory, in `.pdf` and `.fig` format.

The frequency band plot starts by showing the **bandpass filtered input signals** $x_{ci}(t)$ and $x_{mi}(t)$. It then shows the **envelopes** of those signals, calculated using the square-law envelope detector detailed in Section 8.2.3. In this plot, it can be seen that a large amount of gain is applied to both envelopes - this is the gain from the LPF, discussed in Section 8.2.4. The plot then shows the **gain cancelled envelope**, which is the quotient $\frac{A_{mi}(t)}{A_{ci}(t)}$ mentioned in Section 8.2.5. Finally, the plot shows the output signal for that band, $y_i(t)$.

Once the output signal for each band has been produced, they are summed together to produce the final output signal.

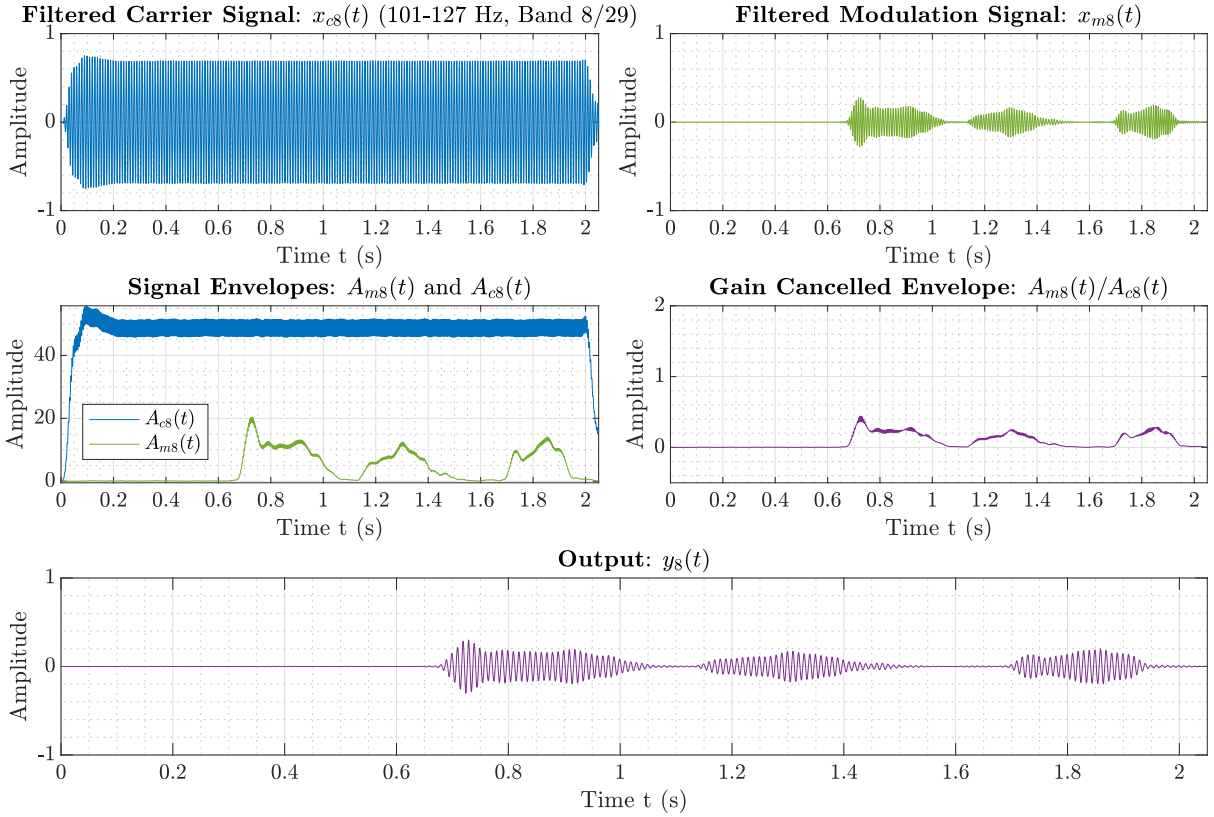


Figure 8.7: Vocoder frequency band 8.

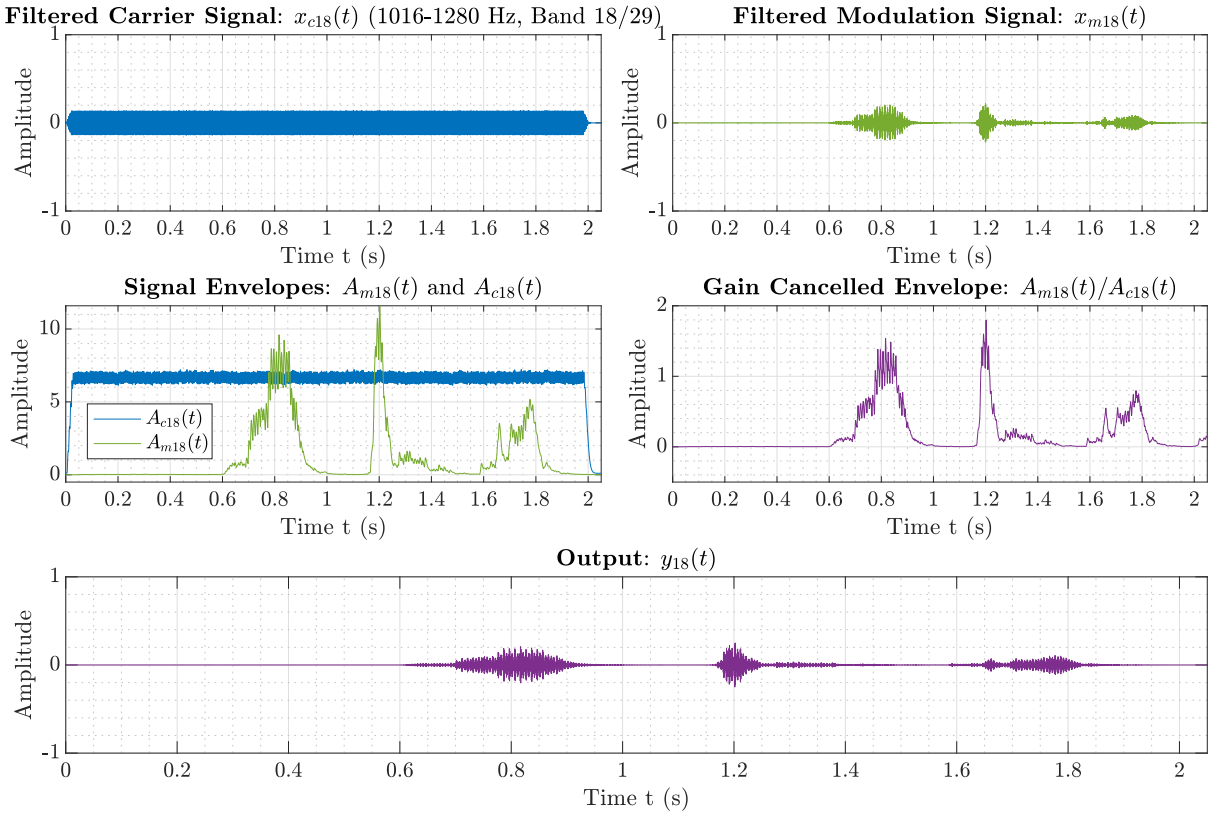


Figure 8.8: Vocoder frequency band 18.

8.3 Demo Recordings

Example recordings and plots of the vocoder can be found in the public folder (Appendix B), in the `Vocoder` directory. There are two examples: a simple example with a squarewave carrier signal, and a more complex example attempting to recreate the main riff of "Harder, Better, Faster, Stronger", by Daft Punk [2], mentioned earlier.

8.4 Reflection

Figures 8.7 and 8.8 show the vocoder implementation successfully extracts the envelope of the modulation signal and applies it to the carrier signal, producing a signal that looks like a combination of both.

By listening to the example recordings the vocoder can be considered a success; for both examples, the output recording sounds like a blend of the carrier and modulator input signals. Whilst the Daft Punk example does not sound exactly like the song, it's close, and is good enough for this project.

Further improvements could be made to improve the internal mechanisms of the vocoder - one such improvement could be to change the envelope detector to use the Hilbert transform; this would remove the frequency limitations of the square-law detector [24].

[2] This starts at 2:01 in the song [21].

Chapter 9

Conclusion

This project taught the author the value of active learning; for some of the effects (particularly reverb), there seemed an overwhelming amount of information to absorb and digest on the topic. The author started by trying to research all the different algorithms available for reverb and made little progress on the effect; it was only once a MATLAB implementation of the effect was made that the author began to gain an intuition for the algorithms.

The DSP theory proved to be more complicated than expected. As such a lot of time was spent with the project supervisor, who was incredibly helpful in explaining the more complicated theory.

The extension objective of creating a physical effects unit with the Bela was not reached, but this is something the author intends to progress outside of this project.

The VST plugins for Echo and Tremolo sound good, and the author considers them complete and usable for real-life playing. The Distortion plugin still needs some tweaking; whilst the effect has been effectively implemented, the plugin sounds far off industry alternatives. The Reverb MATLAB implementation successfully adds a sense of space to the signal and sounded better than the author expected. The Vocoder implementation was the most technically impressive and interesting-sounding effect of the project. Whilst it could be quicker and the output could be improved, the author is happy with the result.

Overall, the project was a success; the author has developed a comprehensive understanding of the DSP theory of each effect in the project, and effective VST or MATLAB implementations have been made for each of the effects.

Bibliography

[1] Bela, "Bela Homepage," 2024. URL: https://bela.io/.

[2] JUCE, "JUCE Home Page," 2024. URL: https://juce.com.

[3] REAPER, "REAPER Homepage," 2024. URL: https://www.reaper.fm.

[4] steinberg, ""Our Technologies" page," 2024. URL: https://www.steinberg.net/technology/.

[5] AKG by HARMAN, "AKG K52 Product Page," 2024. URL: https://uk.akg.com/headphones/K52.html.

[6] Focusrite, "Refurbished scarlett solo audio interface," 2024. URL: https://focusrite.com/products/scarlett-solo-refurbished.

[7] steinberg, "VST SDK Homepage," 2024. URL: https://steinbergmedia.github.io/vst3_doc/vstsdk/index.html.

[8] JUCE, "Tutorials Page," 2024. URL: https://juce.com/learn/tutorials/.

[9] The Audio Programmer, ""JUCE Framework Tutorials" YouTube Series," 2024. URL: https://www.youtube.com/@TheAudioProgrammer.

[10] S. J. Orfanidis, *"Delays, Echoes and Comb Filters," in Introduction to Signal Processing.* Prentice-Hall, 1996.

[11] U. Zölzer, *DAFX: Digital Audio Effects, Second Edition: Chapter 5.3.3 Overdrive, Distortion and Fuzz.* John Wiley & Sons, 2011.

[12] C. Floisand, "Coding some tremolo," 2012. URL: https://christianfloisand.wordpress.com/2012/04/18/coding-some-tremolo/.

[13] U. Zölzer, *DAFX: Digital Audio Effects, Second Edition.* John Wiley & Sons, 2011.

[14] S. Costello, "The physics and psychophysics of plates," 2015. URL: https://valhalladsp.com/2015/11/08/the-physics-and-psychophysics-of-plates/.

[15] W. Gardner, *Applications of Digital Signal Processing to Audio and Acoustics, Chapter 3 - Reverberation Algorithms.* Kluwer Academic Publishers, 1998.

[16] U. Zölzer, *DAFX: Digital Audio Effects, Second Edition: Chapter 5.6 Reverberation.* John Wiley & Sons, 2011.

[17] M. R. Schroeder, "Natural sounding artificial reverberation," *Journal of the Audio Engineering Society*, 1962.

[18] J. O. Smith, ""Schroeder Reverberators" in Physical Audio Signal Processing," 2010. URL: https://ccrma.stanford.edu/ jos/pasp/Schroeder_Reverberators.html.

[19] E. W. Weisstein, "Incommensurate," 2024. URL: https://mathworld.wolfram.com/Incommensurate.html.

[20] James A. Moorer, "About This Reverberation Business," *Computer Music Journal*, 1979.

[21] Daft Punk, "Harder, Better, Faster, Stronger," 2001. (Song from the "Discovery" Album).

[22] U. Zölzer, *DAFX: Digital Audio Effects, Second Edition: Chapter 8.3.1 Vocoding or cross-synthesis.* John Wiley & Sons, 2011.

[23] Purves D, Augustine GJ, Fitzpatrick D, et al, *Neuroscience. 2nd edition.* Sinauer Associates, Inc., 2001.

[24] Dr. Steven A. Tretter, "Demodulating an AM Signal by Envelope Detection," 2018. URL: https://user.eng.umd.edu/t̃retter/commlab/c6713slides/ch5.pdf.

[25] Native Instruments, "What is a flanger? how to use it in music production," 2023. URL: https://blog.native-instruments.com/what-is-a-flanger/.

[26] H. Bode, "History of electronic sound modification," *JAES Volume 32, Issue 10 pp. 730-739*, October 1984.

Appendix A

Normalised Radian Frequency

The normalised radian frequency ω is a measure of frequency relative to a known, constant value, measured in radians per second.

In this report, normalised radian frequencies are calculated relative to the sampling rate (F_s) of the signal.

$$f \text{ (Hz)} \longrightarrow \omega = \frac{2\pi f}{F_s} \text{ (rad/s)} \tag{A.1}$$

When $f = F_s$, $\omega = 2\pi$. Thus, the Nyquist rate of any signal sample rate can be represented as π, making the mathematics simpler and easier.

Appendix B

Public Folder

The *Public Folder* contains all the files produced by this project.

Volume Warning: The audio recordings have not been volume-matched, so please be wary of the volume - especially with the distortion effect.

To access the (OneDrive) public folder use the following link:

Public Folder Link: `https://bit.ly/tm-public-folder`

Figure B.1: Public Folder QR Code.

Or, if GitHub is preferred:

GitHub Link: https://github.com/tom-milner/DSP-Algorithms-for-Audio-Effects

There are subdirectories for each of the effects in the project containing audio files, MATLAB code and plots, and VST Plugin C++ code for the reader's further interest.

The folder structure is as follows:

1. **Demo Effects**

   Contains quick demos in the form of audio recordings and figures of all the effects in the project.

2. **Code**

   Contains code for implementations of each of the effects, aswell as all the code used to generate plots and demo recordings.

Appendix C

Echo Based Effects

By tweaking the delay and gain (D and a) the echo filter can produce many different effects, some of which can be seen in Table C.1 (reproduced from "*DAFX: Digital Audio Effects*"[13]).

Delay Range (ms) (Typ.)	**Modulation (Typ.)**	**Effect Name**
0...15	-	Resonator
0...20	Sinusoidal	Flanging
10...25	Random	Chorus
25...50	-	Slapback
> 50	-	Echo

Table C.1: The different effects caused by different delay lengths in the echo filter.

The middle column of Table C.1 introduces the interesting idea of modulating the delay length D with a time-varying signal, changing D over time. This would vary the frequency response of the filter over time, producing some curious effects. *Flanging* and *Chorus* both rely on this idea.

Flanging recreates the *swooshing* sound created by a jet engine flying overhead [25]; a noticeable usage of it is at the end of the song "Itchycoo Park" by the band "Small Faces" [26].

Chorusing reproduces "the effect of a group of musicians playing the same piece simultaneously. The musicians are more or less synchronized with each other, except for small variations in their strength and timing" [10]. A famous example of the chorus effect is in the intro of "Paradise City" by "Guns and Roses".

Appendix D

Harmonics of Power Chords

A particular example of distortion enhancing the sound of a chord is the *power chord*. A power chord is a simple chord made of two particular notes (using their music theory names): the *root*, and the *fifth*. The music theory meaning of these names is unimportant, but what is important is that the ratio of the frequency of the root, f_r, to the frequency of the fifth, f_5, is very close to $3:2$, i.e:

$$f_5 \approx \frac{3}{2} \times f_r$$

When the power chord is played through distortion, harmonics are produced for both of these notes. As both f_r and f_5 are multiples of $f_r/2$, the odd harmonics produced will also all be multiples of $f_r/2$.

$$f_r = 2 \times \frac{f_r}{2} \qquad\qquad f_5 = 3 \times \frac{f_r}{2}$$

Thus, when the listener hears the distorted power chord, what they hear sounds like *harmonics* of $f_r/2$, tricking their ear into hearing $f_r/2$; even though it was never played! This can be seen in Figure D.1. A peak in the signal can be seen at $f_r/2$, and the vertical dotted blue line shows the repeating odd harmonics of $f_r/2$.

Audio samples of different levels of distortion applied to power chords can be found in the Public Folder (Appendix B).

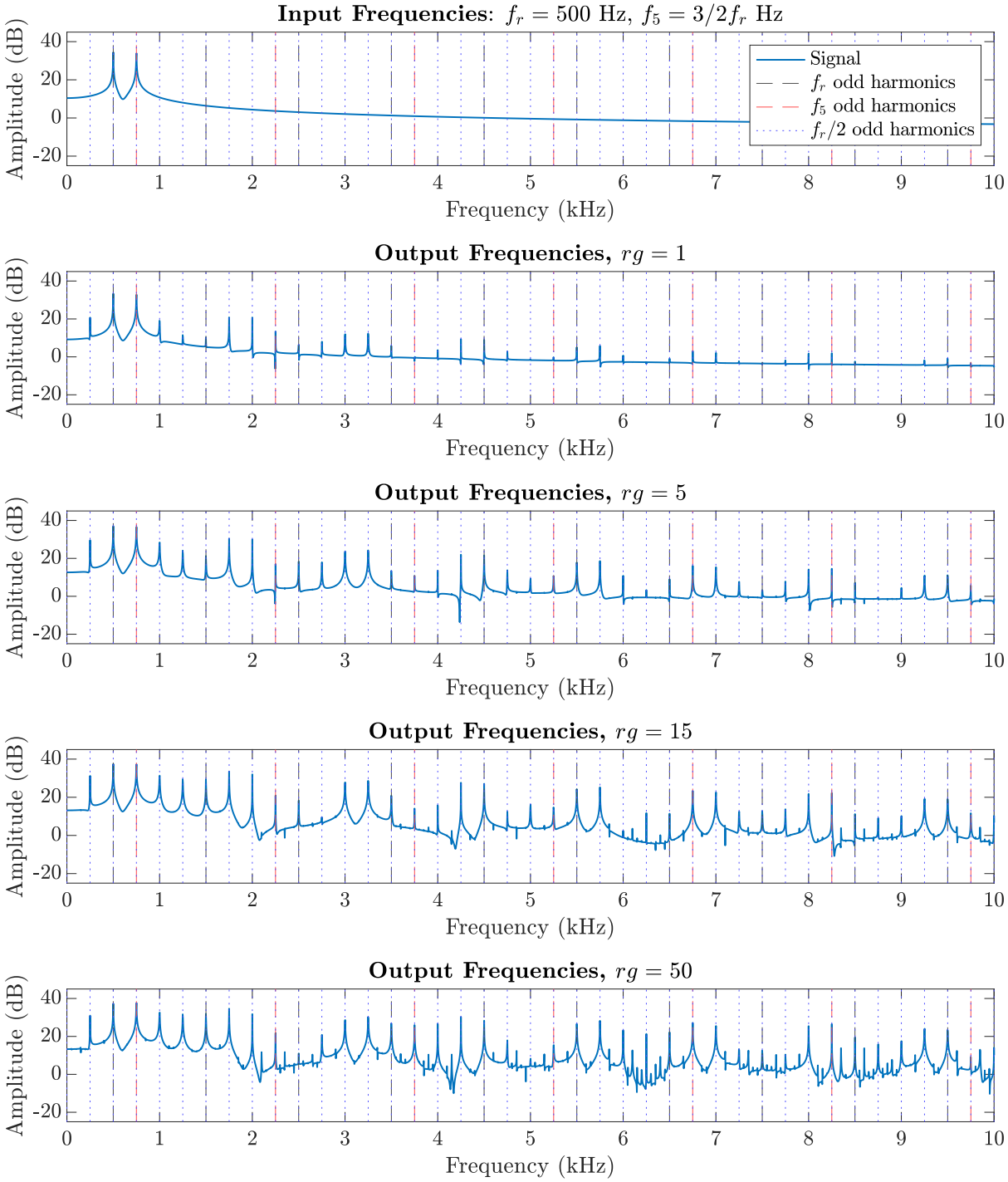


Figure D.1: The energies of the harmonics increase just as in Figure 5.2, but a new fundamental frequency is heard by the listener: $f_r/2$.