BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

AUDIO EFFECT BASED ON LPC ANALYSIS OF VOICE

AUDIO EFEKT ZALOŽENÝ NA LPC ANALÝZE HLASU

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR — Jonáš Horčic

AUTOR PRÁCE

SUPERVISOR — prof. Dr. Ing. Jan Černocký

VEDOUCÍ PRÁCE

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Audio inženýrství**
specializace Zvuková technika
Ústav telekomunikací

Student: Jonáš Horčic ***ID:*** 246843
Ročník: 3 ***Akademický rok:*** 2024/25

NÁZEV TÉMATU:

Audio efekt založený na LPC analýze hlasu

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s technikami analýzy řečových signálů, soustřeďte se na LPC a detekci formantů. Prostudujte zvukové efekty využívající vlastností hlasu pro úpravu vstupních zvuků - Vocoder, Talkbox, diplomovou práci Tomáše Mészárose a další. Navrhněte zvukový efekt nebo sadu zvukových efektů s využitím analýzy řeči. Implementujte efekt(y) off-line (například v Pythonu) a pečlivě testujte. Navrhněte rozšíření efektu(ů), využívající např. modifikace formantů, analýzy základního tónu, harmonizace základního tónu, auto-tune, atd. Implementujte tato rozšíření, včetně zpracování v reálném čase jako softwarový plugin. Navrhněte, proveďte a vyhodnoťte uživatelské testy.

DOPORUČENÁ LITERATURA:

[1] DENG, L. and O'SHAUGHNESSY, D. Speech Processing: a dynamic and optimization-oriented approach. 1st ed. New York: Marcel Dekker, 2003. ISBN 08-247-4040-8

[2] REISS, Joshua D. a MCPHERSON, Andrew P. Audio effects: theory, implementation and application. Boca Raton: CRC Press, 2014. ISBN 978-1-4665-6028-4

Termín zadání: 10.2.2025 ***Termín odevzdání:*** 3.6.2025

Vedoucí práce: prof. Dr. Ing. Jan Černocký

doc. Ing. Jiří Schimmel, Ph.D.
předseda rady studijního programu

UPOZORNĚNÍ:

Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

Abstract

This thesis focuses on the development of a digital voice sound effect using linear predictive coding. It examines signal processing methods for analysing speech signals, with emphasis on key voice characteristics such as formants. Several extensions of the effect were implemented, including pitch shifting. The effect was specifically designed and optimised for real-time processing and implemented in C++ using the JUCE library. The final effect is distributed as a VST plugin featuring a graphical user interface. User testing was carried out to assess the quality of the effect.

Abstrakt

Tato práce se zaměřuje na vývoj digitálního zvukového hlasového efektu pomocí lineárního prediktivního kódování. Zabývá se metodami zpracování signálu pro analýzu řečových signálů se zaměřením na klíčové hlasové charakteristiky, jako jsou formanty. Byla implementována rozšíření efektu, včetně pitch shiftingu. Efekt byl navržen a optimalizován pro zpracování v reálném čase a byl implementován v jazyce C++ pomocí knihovny JUCE. Konečný efekt je distribuován jako VST plug-in s grafickým uživatelským rozhraním. Pro posouzení kvality efektu bylo provedeno uživatelské testování.

Keywords

Linear predictive coding, sound effect, vocoder, VST plug-in, DAW, real-time processing, JUCE

Klíčová slova

Lineární prediktivní kódování, zvukový efekt, vokodér, VST plug-in, DAW, zpracování v reálném čase, JUCE

Reference

HORČIC, Jonáš. *Audio effect based on LPC analysis of voice.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication. Supervisor prof. Dr. Ing. Jan Černocký,

Rozšířený abstrakt

Tato práce se zabývá digitálním zpracováním signálů pro použití v hudebních zvukových efektech. Zvukový efekt je zde definován jako systém pro zpracování signálů, který mění vstupní zvuk hudebně zajímavým způsobem. Přesněji se práce zaměřuje na zvukové efekty, které pracují s lidským hlasem.

Lidský hlas je časově proměnný, ale je možné ho rozdělit na dostatečně krátké rámce o délce 15 do 30 milisekund. Další důležitý poznatek je, že spektrum lidského hlasu má výrazné formanty. Na těchto dvou principech je založen syntezátor hlasu, který je předmětem této práce. Jedná se o kódování pomocí lineární predikce (LPC). Lineární predikce je metoda založená na autoregresi minulých vzorků. Z pojmu *kódování řeči* vzniklo slovo, které se obecně používá pro označení tohoto typu efektu: *vokodér*.

Lineární kódování je v podstatě interpolace spektrální amplitudové obálky. Provádí se aproximace a počítá se s určitým chybovým signálem zvaným *reziduum*. Primární použití LPC je pro kompresi řeči v telekomunikacích. Hlavní důvod použití LPC je ten, že umožňuje rozdělit řeč na zdroj hlasu (buzení) a funkci artikulačního ústrojí (koeficienty filtru).

Jako filtr, který představuje artikulační ústrojí, byl zvolen autoregresní filtr s nekonečnou impulzní odezvou (IIR). Při použití LPC pro vytvoření hudebního efektu nejde o nejvěrnější reprodukci. Naopak, jako buzení se použije zvuk hudebního nástroje nebo jiná nahrávka. Tím vznikne efekt, který se označuje jako *cross-synthesis*. Tento efekt je zvukem podobný historickému efektu talk-box. V úvodní kapitole 1.1 je popis talk-box efektu i jeho digitální podoby od T. Mészárose.

Kapitola o technickém návrhu efektu začíná blokovým schématem celého řetězce algoritmů. Jako hlavní efekt je implementována LPC křížová syntéza. Byla vyzkoušena různá rozšíření a samostatné efekty, z nichž byl proveden následný výběr. Nejsložitější rozšíření efektu je algoritmus schopný změnit výšku tónu hlasu se zachováním pozic formantů. Algoritmus pracuje ve frekvenční doméně. Jedná se o implementaci pitch shifting efektu z kolekce *DAFX* [17]. Zapojením více nezávislých bloků tohoto efektu vzniká další efekt, který se nazývá unison. Unison může sloužit i k obohacení hlasu před vstupem do LPC efektu. Na konci zpracování bylo přidáno řízení šířky stereo signálu pro omezení nejkrajnějších signálů a řízení míry směšování zpracovaného signálu s původním (Dry / wet).

Než bylo možné vytvořit implementaci schopnou zpracování v reálném čase, bylo potřeba lépe pochopit jednotlivé algoritmické části. Pro potřeby vývoje bylo kromě C++ využito programovacích jazyků Python a MATLAB. Jejich výhody i nevýhody jsou popsány v kapitole o nástrojích a knihovnách 3. Finální implementace je ovšem napsaná v C++.

Kapitola o implementaci se zabývá finalizací efektu. Byl srovnán výkon původní verze algoritmického řetězce a ekvivalentních algoritmů založených na rychlé Fourierově transformaci. Pro splnění podmínky zpracování v reálném čase byly zvoleny algoritmy s nižší komplexitou a optimalizované funkce z knihoven. Do diskuze byla zahrnuta i ideální velikost rámců signálu vzhledem ke kvalitě zvuku, odezvě a výkonu.

Pro ovládání parametrů efektu bylo vytvořeno grafické uživatelské rozhraní pomocí integrace WebView technologií v knihovně JUCE. Jedná se o vestavěný webový prohlížeč, který zobrazuje uživatelské rozhraní vytvořené v JavaScriptu. V závěru kapitoly byly popsány hlavní technické problémy a jejich řešení, které mohou sloužit jako užitečná reference.

Výstupem této práce je zcela digitální zvukový efekt vytvořený pro DAW produkční prostředí. Knihovna JUCE byla využita zejména pro kompilaci programu jako VST zásuvný modul, který je použitelný v různých DAW.

Hodnocení efektu v průběhu vývoje a v závěrečných fázích není možné provádět pouze měřením. Hudba a její vnímání jsou nevyhnutelně subjektivní. Objektivně lze posoudit

algoritmy, výkon, stabilitu a odezvu. Nejdůležitější jsou ale kvalita zvuku, možnosti použití a další subjektivní vlastnosti. Z tohoto důvodu efekt otestovalo několik recenzentů. Ověřili funkčnost efektu na různých systémech a přispěli do diskuze svými zkušenostmi a názory. Byl vytvořen dotazník pro zpětnou vazbu, který pokrývá nejdůležitější body. Obsahuje i otevřenou formu recenze. Výsledky jsou zhodnoceny v závěru (kapitola 7).

Prohlášení

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucího závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

. . . . . .
Jonáš Horčic
April 16, 2025

Poděkování

Děkuji prof. Janu Černockému za odbornou pomoc, cenné rady a materiály. Nejvíce pak za rozumný přístup a důvěru. Děkuji všem, kteří se účastnili na testování produktu a poskytli zpětnou vazbu.

Contents

1 Introduction **4**
- 1.1 Talk-box effect . . . 4
- 1.2 The goal of this thesis . . . 5
- 1.3 Frequently used terms . . . 6
- 1.4 Existing voice effects . . . 6

2 General theory **9**
- 2.1 Human voice . . . 9
- 2.2 Vocoder . . . 10
- 2.3 Core DSP . . . 11
- 2.4 Linear predictive coding . . . 13

3 Tools and practical considerations **17**
- 3.1 Programming languages . . . 17
- 3.2 VST plugins and host applications . . . 18
- 3.3 JUCE . . . 19
- 3.4 Build tools . . . 20
- 3.5 KFR framework . . . 21

4 Effect design **22**
- 4.1 Proposed design . . . 22
- 4.2 LPC effect design . . . 23
- 4.3 Data handling and processing . . . 23
- 4.4 Real-time processing . . . 25
- 4.5 Effect extensions . . . 26

5 Implementation **30**
- 5.1 Performance optimisation . . . 30
- 5.2 Frame collection . . . 34
- 5.3 Sample rates and frame sizes . . . 34
- 5.4 Loudness . . . 35
- 5.5 GUI . . . 35
- 5.6 Technical issues . . . 36

6 Testing and evaluation **39**
- 6.1 Distribution . . . 39
- 6.2 User feedback . . . 39

7 Conclusion **40**
7.1 Summary of work done 40
7.2 Feedback evaluation 40
7.3 Future work 41

Bibliography **42**

A User reviews **44**

B Submitted files **46**

C User manual **47**

List of Abbreviations

AU	Audio Unit
AAX	Avid Audio eXtension
CSS	Cascading Style Sheets
CPU	Central Processing Unit
DAW	Digital Audio Workstation
DnB	Drum and Bass
DSP	Digital Signal Processing
EDM	Electronic Dance Music
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FT	Fourier Transform
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IFFT	Inverse Fast Fourier Transform
IIR	Infinite Impulse Response
JUCE	Jules' Utility Class Extensions
LPC	Linear Predictive Coding
LLVM	Low Level Virtual Machine
MIDI	Musical Instrument Digital Interface
OLA	Overlap-Add
PSOLA	Pitch Synchronous Overlap-Add
RMS	Root Mean Square
SIMD	Single Instruction, Multiple Data
STFT	Short Time Fourier Transform
VST	Virtual Studio Technology

Chapter 1

Introduction

The objective of this thesis is to document the process and the knowledge gained throughout the development of a digital sound effect based on linear predictive coding. Here, a sound effect (Figure 1.1) is defined as a device that alters an input sound, for example from a musical instrument, to create a new output sound.

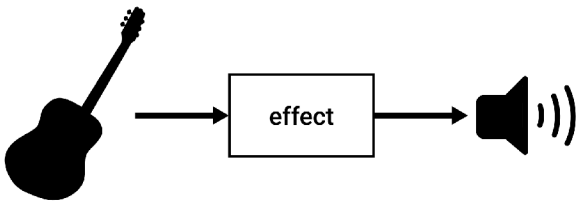

Figure 1.1: Simplified sound effect schematic.

1.1 Talk-box effect

Voice effects are specifically used to alter the characteristics of sounds based on the human voice. One of the first successful analogue effects is the `talk-box`. It is described by some as an effect that "makes instruments talk".

The original analog version of talk-box involves an instrument that generates an audio signal, which is sent to an amplifier and a speaker. The amplified sound travels through a tube that directs it into the singer's mouth. The instrument's sound acts as a replacement voice source, called *excitation*. The performer shapes the excitation using their vocal articulations. This setup is illustrated in figure 1.2:

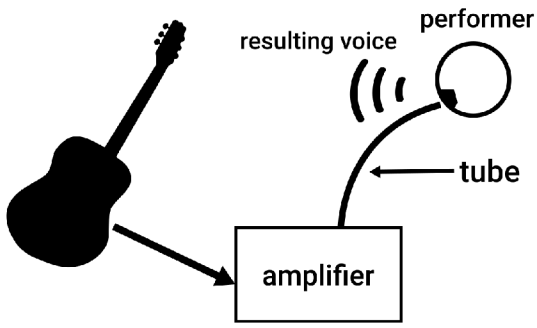

Figure 1.2: The original talk-box concept.

The performer usually plays a guitar or a keyboard synthesiser and has to synchronise the tones with articulation. A drawback of analogue talk-box is the tube leading to the performer's mouth, which is rather impractical, unsightly and unpleasant.

Decades since its introduction, numerous digital talk-box emulators have been created. They are most often referred to as `vocoders`. A vocoder is described in section 2.2. As part of his master's thesis *Speech Analysis for Processing of Musical Signals* [8], Tomáš Mészáros created a talk-box emulation that uses a hardware interface for a microphone and a musical instrument (Figure 1.3). All processing occurs digitally.

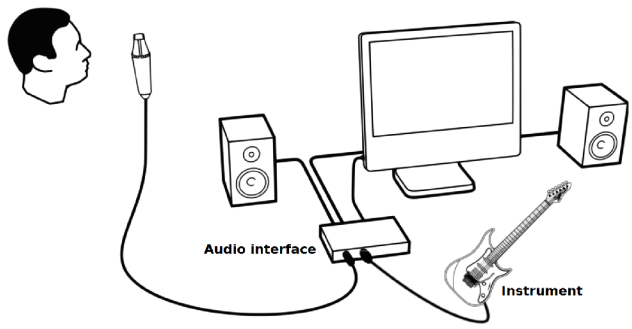

Figure 1.3: Hardware setup of Mészáros's talk-box effect. Reproduced from [8].

1.2 The goal of this thesis

The aim is to create a fully digital voice effect as a plugin for digital audio workstations (DAWs). The effect will be a vocoder based on LPC with real-time processing. This means that music producers should be able to load it into their DAW and use it with any combination of instruments, devices, and other effects. On top, the plan is to implement additional voice manipulation techniques.

Evaluating the effect during development and in the final stages is not possible in a full extent by measurements. Music and preferences are inherently subjective. The aspects that can be assessed objectively include algorithms' performance, stability, latency. The most important, however, are sound quality, versatility and other such qualities as perceived by users. For this reason, selected participants will review the final effect to verify its functionality and to share their perspectives.

1.3 Frequently used terms

To avoid confusion, it is important to define some terms frequently used throughout the thesis.

A vocoder effect requires two inputs – a voice signal and a *carrier signal*. Simply, the *carrier* signal is shaped by the effect to resemble voice. A common *carrier* signal is a guitar or a synthesiser sound. The equivalent of *carrier* is *excitation*. Depending on the context, either of these can be found in literature. *Excitation* will be used exclusively in the rest of the thesis.

Routing two or more signals into a plugin in DAW is possible with a *sidechain* function in a mixer console. Which signal is sidechained – the voice or excitation – is arbitrary. In our implementation, the voice signal is sidechained as shown in figure 1.4.

Figure 1.4: Routing the voice signal using *sidechain* (coloured blue) to channel 4. The effect resides on channel 4.

1.4 Existing voice effects

The selection of digital audio effects has been rapidly expanding in the last decades. Various plugin technologies enabled an easy integration with DAWs. This section lists some of the digital voice effects that can currently be considered the state of the art.

The `Little Alterboy by Soundtoys`[1] is described as a monophonic voice manipulator. Its knobs *pitch* and *formant* change the pitch of the voice and its formant spectrum. Simple and effective, the effect received a lot of attention from music producers. It uses a *vocoder* model (see section 2.2).

[1] https://soundtoys.com/product/little-alterboy/

Figure 1.5: Little Alterboy by Soundtoys.

`VocalSynth 2 by iZotope`[2], the successor of the original `VocalSynth`, has a wide array of voice and general sound effects organised in modules. It uses a sidechain or a MIDI input to emulate the talk box effect, among other custom vocoders. `VocalSynth 2` also has pitch correction and voicing controls.

`Harmony by Waves Audio`[3] seems to be among the most advanced commercial voice sound effects. Described as a real-time vocal harmoniser, `Harmony` is a fully-featured tool for changing notes, chords, and it can control up to 8 voices, each with a customisable pitch and stereo position. It has a user-friendly interface and features such as real-time pitch correction or automatic parameter control.

[2] https://izotope.com/en/products/vocalsynth.html
[3] https://waves.com/plugins/waves-harmony

Figure 1.6: VocalSynth 2 by iZotope.

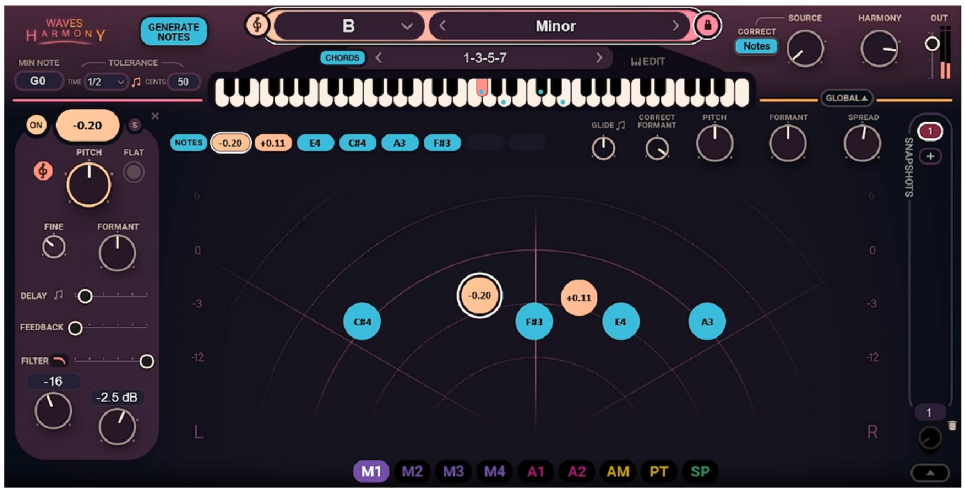

Figure 1.7: Harmony by Waves Audio.

Chapter 2

General theory

This theoretical chapter discusses only the most relevant topics to this thesis. Namely, the human voice, signal processing concepts and the theory behind LPC.

2.1 Human voice

Voice is in its essence made of sound pressure variations originating in the human vocal tract. Speech is made up of various segments. These in turn combine to create words, and words come together to form sentences. Voice has a transient nature. The changes in time carry the information.

Voice can be divided into short segments (frames) that are considered stationary. Practical analysis segment length for speech is widely cited to be between 15 and 30 milliseconds [13]. The length is a compromise between stationarity, resolution and latency.

We recognise two types of speech: voiced and unvoiced. Voiced speech is produced by the periodic vibrations of the vocal cords in the larynx. Intuitively, one can distinguish between voiced and unvoiced sounds by feeling the vibrations of one's throat while talking. The voice originates in the lungs, which push air through the trachea (Figure 2.1). Then, after passing through the larynx, the air conveys either a periodic or an aperiodic sound. The sound is further shaped by the vocal cavities and articulated by the lips and tongue.

An important concept is the separation of the speech source, or excitation, and the function of the vocal tract, which acts as a filter. For the objective of speech synthesis, it may seem like an oversimplification. Nonetheless, various strategies exploit the key characteristics of voice to create a faithful reproduction. Due to the shape of the vocal tract and the influence of the articulators, the voice exhibits prominent *formants* (Figure 2.2).

Formants are broad peaks in the frequency spectrum whose resonance frequencies barely change in the duration of one segment. They are important for intelligibility. Filtering an excitation source to replicate the formants of real speech is one step in creating a vocoder. The other requirement is time variation to form words and sentences.

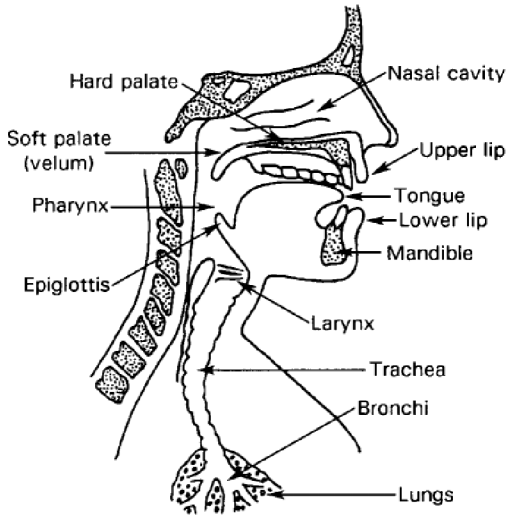

Figure 2.1: Diagrammatic cross-section of the human head, showing the vocal organs. Reproduced from [5].

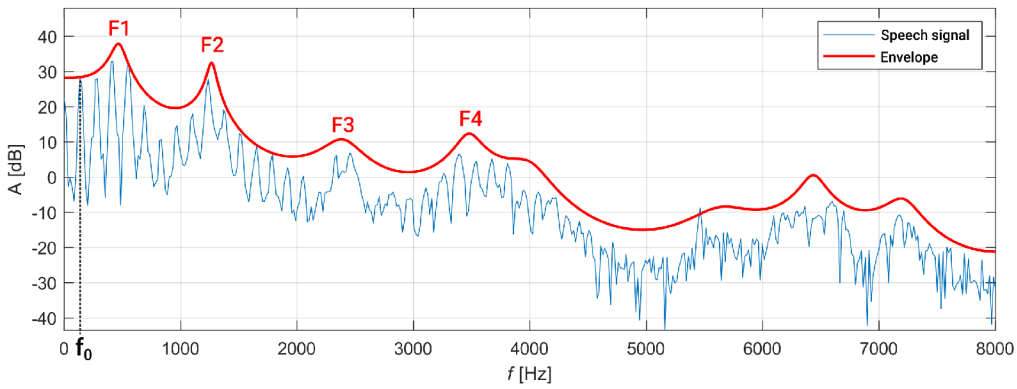

Figure 2.2: Frequency spectrum of a voice recording. Fundamental frequency f_0 and the first four formants are highlighted.

2.2 Vocoder

A vocoder can be described as a coding system that reproduces a voice based on its perceptual characteristics, with some margin of error. The term *vocoder* is a contraction of *voice coder*. Vocoders are not a new invention, yet the underlying principles are widely used in modern DSP applications, notably in telecommunications. Given the variety of vocoders and the difficulty in clearly distinguishing them, I will instead list the different analysis techniques they may use:

- a bank of bandpass filters,
- sinusoidal decomposition,
- spectral filters and resonators,

- linear predictive coding,
- waveform coding,
- neural networks.

Only linear predictive coding is applied in this thesis.

2.3 Core DSP

Convolution

Convolution is an essential DSP concept. For a discrete linear time-invariant (LTI) system with an impulse response $h[n]$, one can calculate its response $y[n]$ for an arbitrary input sequence using linear convolution.

$$y[n] = x[n] * h[n] = \sum_{i=-\infty}^{\infty} x[i]h[n-i]. \tag{2.1}$$

The equation is similar to the correlation theorem 2.3. In fact, the only difference is the direction the indices run. Convolution will be used in computation of residuals 2.4.

Autocorrelation

Autocorrelation is the case of cross-correlation when a signal is compared with a delayed version of *itself*. The use of autocorrelation in linear prediction is for its ability to suppress undesirable phase effects while preserving the harmonics and formants. For each lag k, there is one autocorrelation coefficient $R[k]$. In autocorrelation with normalisation, $R[0]$ is always equal to 1.

$$R[k] = \sum_{n=-\infty}^{\infty} x[n]x[k+n]. \tag{2.2}$$

More applicable for our use case is the Pearson's correlation coefficient [3]. It is also known as zero mean normalised cross-correlation (ZNCC). The coefficient values are between 1 and -1:

$$R[k] = \frac{\sum_{n=1}^{N-k}(x_n - \overline{x})(x_{n+k} - \overline{x})}{\sum_{n=1}^{N}(x_n - \overline{x})^2}, \tag{2.3}$$

where N denotes the total number of samples, $\overline{x}$ the mean of samples and k is the lag. Implementing (2.3) as an algorithm is feasible but quite inefficient; see section 5.1.

Autocorrelation function of a voiced segment typically has defined peaks and periodicity, called an impulse train. Autocorrelation function of an unvoiced segment is closer to white noise. From these peaks, we can infer the pitch period and pitch frequency (figure 2.3).

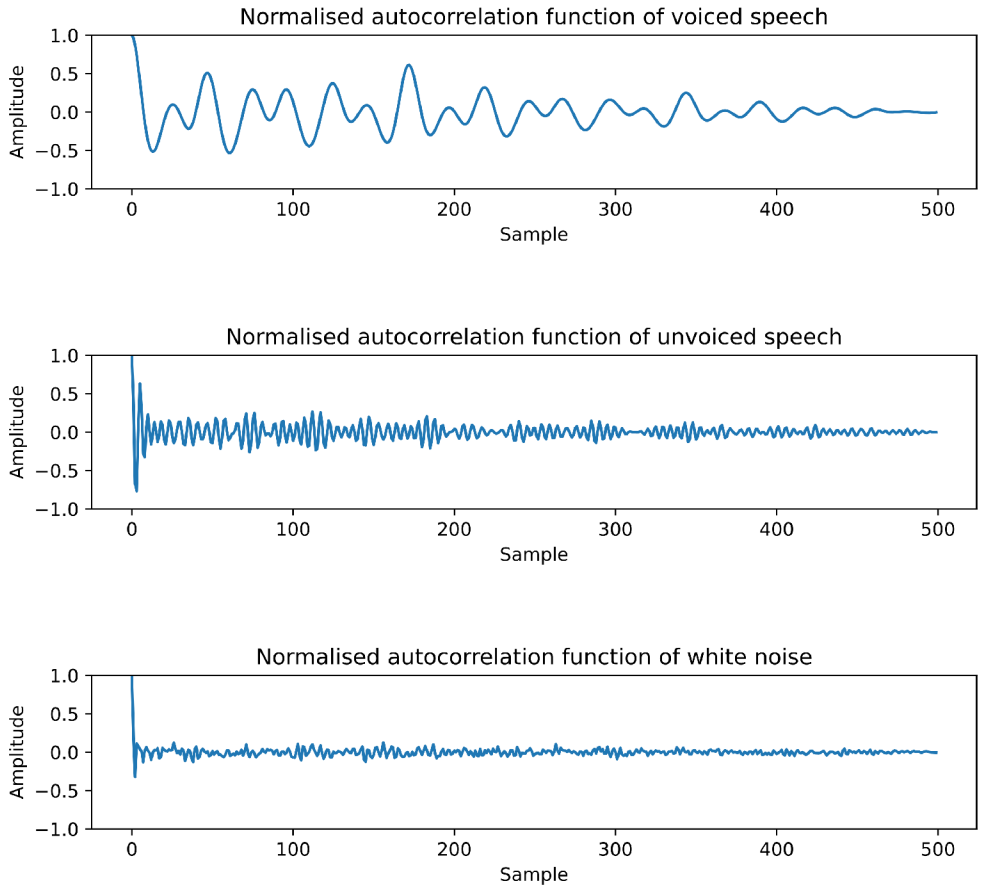

Figure 2.3: Normalised autocorrelation functions of voiced speech, unvoiced speech and white noise. Rectangular windowing was applied.

Fourier Transform

The Discrete Fourier Transform (DFT) algorithm is the cornerstone of DSP. Fourier transform transforms a discrete signal into the frequency domain. A frequency spectrum is a representation of a discrete signal decomposed into frequency bins described by complex numbers.

However, computing the DFT directly can be computationally expensive, especially for large data sizes. To address this, the Fast Fourier Transform (FFT) was developed as an efficient algorithm to compute the DFT. It works on the principle of divide and conquer, splitting a large operation into easily solved parts.

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-\mathrm{j}2\pi \frac{nk}{N}} \qquad k \in < 0, N-1 > \tag{2.4}$$

Let N denote the window size and $x[n]$ represent the input signal. The term $\frac{k}{N}$ corresponds to the normalised frequency associated with the k-th frequency component. Thus, $X[k]$ represents the frequency component at index k for a particular window of the input signal $x[n]$.

Convolution can be computed as a product of the FFT of two signals [1]. The DFT and FFT abbreviations are used interchangeably. I will use the term FFT.

2.4 Linear predictive coding

LPC relies on the linear combination of previous samples to predict the sample at current time:

$$\begin{aligned} s[n] &\approx \hat{s}[n] = -a_1 s[n-1] - a_2 s[n-2] - \ldots - a_p s[n-p] \\ \hat{s}[n] &= -\sum_{i=1}^{p} a_i s[n-i]. \end{aligned} \tag{2.5}$$

This way, although not perfectly, the original signal can be described using a significantly lower volume of data. The term *coding* can be used interchangeably with *compression*. Both refer to reducing the information needed to recreate the voice's spectral envelope.

The prediction is based on the previous samples up to the p-th past sample. The a_i are prediction coefficients and p is the model order. The model order is chosen depending on the context, analysis frame length and the sampling frequency.

Prediction error, also called the residual, is defined as the difference of the real and predicted samples. I denote it as $d[n]$, following Rabiner and Schafer [10], although it is frequently presented as $e[n]$:

$$d[n] = s[n] - \hat{s}[n] = s[n] + \sum_{i=1}^{p} a_i s[n-i]. \tag{2.6}$$

The equation can be simplified:

$$d[n] = \sum_{i=0}^{p} a_i s[n-i], \quad a_0 = 1. \tag{2.7}$$

Taking the z-transform of the prediction error (2.6), we obtain an output of a linear filter with a system function $A(z)$:

$$\begin{aligned} D(z) &= S(z) + \sum_{i=1}^{p} a_i S(z) z^{-i} \\ D(z) &= S(z) \left[1 + \sum_{i=1}^{p} a_i z^{-i} \right] = S(z)A(z).anoutputofafilter \end{aligned} \tag{2.8}$$

By isolating $S(z)$, the output component, we get the function $A(z)$:

$$A(z) = \frac{D(z)}{S(z)} = 1 + \sum_{i=1}^{p} a_i z^{-i}. \tag{2.9}$$

$A(z)$ is a function of the so-called whitening or analysis filter. It is the inverse of the intended speech synthesis filter. The equation can be rearranged:

$$S(z) = D(z) \frac{1}{A(z)}. \tag{2.10}$$

The speech signal $s[n]$ is an output of a filter with a transfer function $1/A(z)$ and the residual $d(n)$ as its input. The object of interest is the transfer function $H(z)$ of the synthesis filter:

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + \sum_{i=1}^{p} a_i z^{-i}}. \tag{2.11}$$

It describes an IIR filter with no zeroes and only poles. This type of filter is called an all-pole filter or an autoregressive (AR) model. It is a substitute for the human vocal tract. Given the correct coefficients, change in time, and an excitation signal, the all-pole filter is able to synthesise speech.

Prediction errors and residuals

The prediction coefficients a_i are the key information in linear prediction. They are acquired from the analysis of the original speech signal $s[n]$. One can think of LPC as an interpolation of the spectral envelope. The least squares method is used to compute a_i with minimal energy of the residuals (the sum of its squared samples). The route to this finding begins with the definition of the total squared error for the least squares method. All steps and reasoning made to achieve the *normal equation* (2.12) are thoroughly covered in [10]:

$$\sum_{i=1}^{p} a_i c_{ik} = c_{0k}, \quad 1 \leq i \leq p. \tag{2.12}$$

The equation has unknowns a_i and c_{ik}. First, coefficients c_{ik} needs to be computed for i in range $1 \leq i \leq p$ before it is possible to get the prediction coefficients a_i. Two viable methods to compute c_{ik} are the Covariance and Autocorrelation methods, which operate with a different range of samples.

Covariance and autocorrelation methods

The covariance method uses no explicit window and the number of samples is therefore theoretically unlimited. It is instead bounded by the range of error. In autocorrelation method, the samples $s[n]$ are windowed by a windowing function (see section 4.3). These differences carry implications for the computed values and filter stability. The autocorrelation method was chosen for its guaranteed filter stability [10] and because it leads to a Toeplitz matrix, which can be solved efficiently.

Levinson–Durbin recursion

The autocorrelation method was chosen to calculate c_{ik} of a speech segment. The issue at hand involves a set of p linear equations, known as the Yule-Walker equations, organised in the Toeplitz matrix:

$$\begin{bmatrix} r_0 & r_1 & r_2 & \cdots & r_{p-1} \\ r_1 & r_0 & r_1 & \cdots & r_{p-2} \\ r_2 & r_1 & r_0 & \cdots & r_{p-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{p-1} & r_{p-2} & r_{p-3} & \cdots & r_0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_p \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_p \end{bmatrix}, \tag{2.13}$$

where r_i are autocorrelation coefficients obtained from the autocorrelation (2.3) of a segment. Only prediction coefficients a remain to be calculated. Gaussian elimination has

an algorithm of complexity $O(n^3)$. The implications of complexity is discussed in detail in section 5.1. The Levinson–Durbin recursion exploits the symmetricity of a Toeplitz matrix to compute the coefficients with a complexity of $O(n^2)$. The result is a matrix containing the LPC coefficients a.

Excitation

The prediction coefficients define the filter (2.11) as a simplification of the vocal tract. The filter should recreate the spectral envelope by using resonances only. White noise with a constant power spectral density is a suitable excitation source. The all-pole IIR filter works quite well, as stated by O'Shaughnessy: "Physical signals tend to be characterised more by the presence of energy at certain frequencies rather than by its absence. That is, signals with spectral peaks are more common than ones with zeroes." [9].

I want to clarify the distinction between excitation and prediction error, which I respectively denoted as $e[n]$ and $d[n]$. While *excitation* generally refers to any input signal to a system, in LPC speech synthesis, it typically denotes the prediction error.

Spectral envelope

The question arises, how many prediction coefficients are considered sufficient? Just like with segment length, there is not one correct answer. The deciding criterion is the spectral resolution. A higher order increases the accuracy of the prediction model. But it comes at the cost of computation time, memory and transmission bandwidth [9]. Plots of a 2-second *a* vowel sound for different model orders were generated. White noise is used for excitation (Figure 2.4).

A low model order may produce unintelligible speech, while a high value creates exaggerated peaks corresponding to multiples of the fundamental frequency. Values in-between these extremes are acceptable. Ideally, the users of the effect should be able to set the model order within a reasonable range, for example $15 < p < 75$ at a sample rate 44.1 kHz.

Cross-synthesis

The basis of the sound effect is cross-synthesis, which is made possible thanks to the separation of excitation and spectral shaping filter. It is straightforward to use a different excitation other than white noise or simple impulse trains. One can supply a synthesiser, a musical instrument or any other sound as an excitation signal. Reminiscent of the talk-box effect (section 1.1), cross-synthesis can make "instruments talk". As the model works with a filter, the output sound depends heavily on the signals' harmonic contents. My plugin will allow the use of any excitation source.

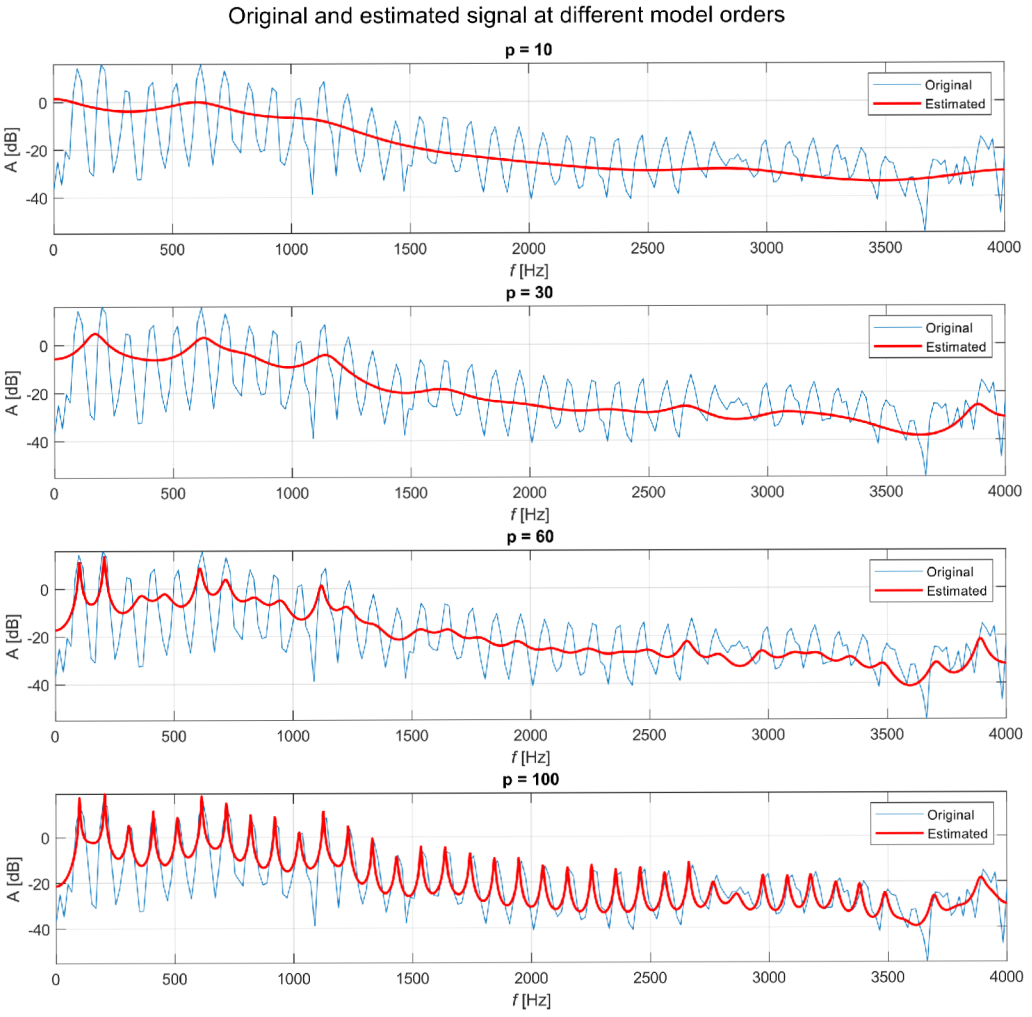

Figure 2.4: Magnitude frequency response of original and estimated signal, model orders of $p = 10, 30, 60, 100$ at a sampling frequency 8 kHz.

Chapter 3

Tools and practical considerations

This chapter focuses on the design process. The end goal is to develop an online implementation of the effect as a VST plugin in C++. However, I am free to use any tools at disposal. Each tool has its own strengths and shortcomings. For this reason, my workflow is split in different development environments.

3.1 Programming languages

MATLAB

I started working directly in C++, but after some time it became clear that it does not offer fast enough recompilation. Testing individual code pieces requires a lot of setup time. For learning about different algorithms, I rely on MATLAB by MathWorks. MATLAB is primarily a matrix calculator and a simulation toolbox. The signal processing extension has ready-made functions like cross-correlation and FFT. MATLAB has a forgiving engine and quick to write vector operations.

Its code analysis and refactoring options are limited. I still consider it more of a numeric computing tool than a general programming language. It does not have many third-party integration options and the proprietary functions are not extensible. Its use is strongly incentivised by example pieces of code in literature and in older web sources. MATLAB's usefulness declines with more specific requirements. The environment is convenient for prototyping, if one has access to a license.

Python

Python is a general and open source programming language. Its core is fairly limited, delegating the functionality to libraries obtained via package managers. Python is increasingly popular for data analysis, simulation and digital signal processing. Its official and community packages, the number of integration options and frameworks have surpassed MATLAB. Python may be used to create any sort of script, service or application. The similarity of syntax and convenience functions made working parallel with MATLAB an easy choice. All in all, while I am new to Python, the transfer of code is quick. The most trouble is caused by MATLAB indices starting at 1.

The Python packages I ultimately use are the `NumPy`, `SciPy` with math and science functions, `librosa` and `simpleaudio` for sound file operations and playback respectively, and the `matplotlib` for graphing functions. The dynamically typed design of Python and

its permissiveness, good documentation and profiling tools reduce the time from an idea to code.

C++

The main reason I work with C++ in this project is that it is widely used in audio programming. Specifically, for frameworks and standards associated with DAWs and drivers, C++ has been the de facto standard. The JUCE framework written in C++ appears to be the easiest option for creating an audio plugin. It came to my attention while reviewing projects by others that used this framework. JUCE is explored in more detail in the next section 3.3.

C++, when used correctly, imposes very little performance overhead. The language requires a compiler for compilation to machine code for the particular processor achitecture. The statically typed syntax and a lack of convenience functions results in more verbose code than Python. Compilation times and compatibility depend on the supplied compiler. The language offers a structured paradigm and low-level access to resources, but importing libraries and debugging can be time-consuming. Changing code in one place typically requires changing code elsewhere, for reasons such as the header-source design and cross imports. Given these factors, I primarily use C++ for the final implementation and optimisation. Still, I develop some of the features exclusively in C++.

3.2 VST plugins and host applications

VST

VST, standing for Virtual Studio Technology, is an open standard for audio plugin interfaces developed by Steinberg [14]. The latest major revision is the VST3. It is used to develop audio instruments, effects and other utilities packaged into a plugin for a host application or as standalone. Plugin architecture greatly simplifies development and integration with the various DAWs. Ensuring conformity to the specification, the system details and the sound processor code are kept separate.

According to the Steinberg VST API documentation [14], the VST3 consists of two primary components. A *processor* contains the audio processing logic. An *editor controller* is responsible for the parameter states and changes. The processor's inputs and outputs are defined by buses, each bus has a number of audio channels. VST presents a set of standardised functions and formats. The essential functions contain the logic for initialisation, audio processing and user interaction, parameter control and program behaviour definitions.

In addition to the audio processing engine, VST plugins often feature a graphical user interface (GUI) component. The GUI should provide intuitive controls for changing the plugin settings and adjusting the audio processing parameters. VST plugins may also accept MIDI input, for straightforward interaction with hardware like keyboards. MIDI is commonly used as the control medium for sound generators and instruments. In more complex plugins, presets enable users to save and recall parameter states or explore the plugin's capabilities using predefined configurations.

Digital audio workstations

A DAW is primarily a host application that manages communication between the operating system, drivers, and audio software. A DAW is expected to have audio mixing tools,

comprehensive input/output settings, import and export of audio files and extensibility options. Some of the most widely used DAWs are Ableton Live, FL Studio, Logic Pro, Pro Tools, Cubase and Reaper. Assuming an up to date version, nearly all DAWs support the VST plugin format. Alternatives to VST exist. The most prominent are *AU* developed for Apple systems and *AAX* made for Pro Tools. One can expect that a plugin available in both the VST and AU formats will be compatible with most users' systems.

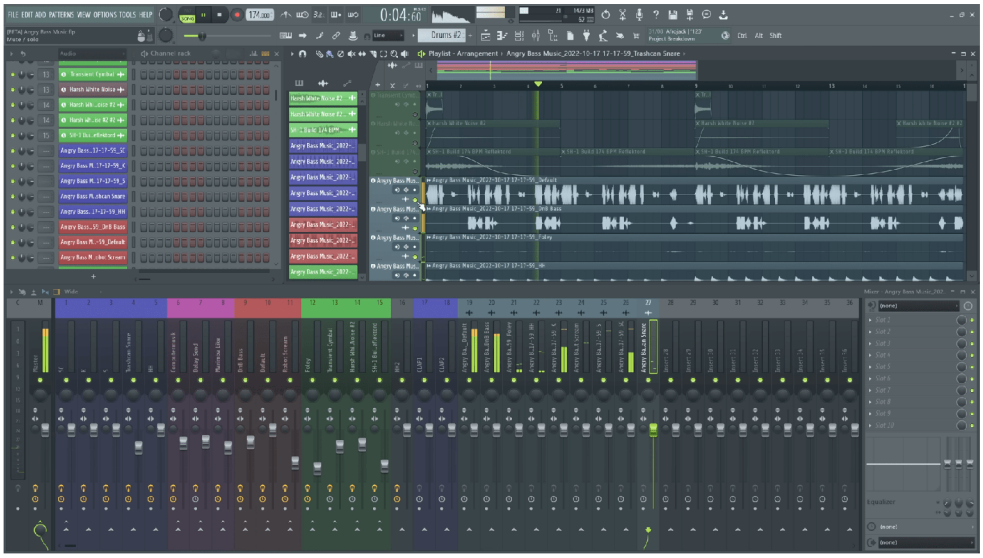

Figure 3.1: FL Studio DAW: the figure features a mixer, channel rack, and piano roll.

A DAW is a complex environment that bridges the hardware and software. The common plugins for DAWs are categorised as *generators* and *effects*. Generators are usually synthesisers, MIDI generators and samplers. The common *effects* include equalisers and filters, compressors, reverb, distortion. Our LPC vocoder also belongs to this category.

My primary testing DAW is FL Studio on Microsoft Windows. The effect was also verified to work in Ableton and in Reaper DAW thanks to the testers (see reviews in Appendix A).

3.3 JUCE

Jules' Utility Class Extensions[1] is an audio development framework written in C++. It is a stable framework in active development with many optional utilities. Although a paid license is needed for commercial use, for open source and academic development, a free GPLv3 license applies. JUCE makes it possible for a single codebase to work in all supported DAWs by exporting it as plugin formats described in section 3.2. Support for MIDI, graphical user interfaces, advanced graphics, and integration points significantly reduces development time compared to building a custom framework from scratch. JUCE lowers the entry barrier to start developing audio applications.

Another benefit of using JUCE is its active community and the abundance of tutorials and blog posts. It is important to note that while JUCE offers some constructs to assist with signal processing and computation, these are more suitable for learning and prototyping

[1] https://juce.com/

rather than for serious use. JUCE is a popular choice for the GUI options. My primary reasons for using JUCE are its export options, licensing terms, and the level of abstraction it provides.

The structure of JUCE templates reflect the plugin interface. At its most basic, the project includes just the *processor* and *editor* files. The *editor* handles the GUI and parameter controls. The *processor* contains directives that communicate the plugin characteristics to the host application. These include MIDI support, the number of programs, and the plugin lifecycle hooks[2]. The method `prepareToPlay` executes just before the plugin starts processing. It is the place where expensive resources are allocated, and some default values are set. A method named `processBlock` is the core of the audio processor's logic. It manages the audio buffer stream. The buffer is a wrapper for the input and output buses. A method `releaseResources` can be used to safely close resources and save states.

3.4 Build tools

This section is dedicated to C++ code compilation. C++ needs to be compiled for the target processor architecture. This means that it has less overhead than other languages that are separated by more layers of abstraction in between the code and the hardware. Among the array of compiler and toolchain technologies, I provide a brief description of the three most relevant ones.

Compilers

GCC, the GNU Compiler Collection, is a free and open source set of compilers for, but not limited to, C language family. GCC follows the UNIX principles and is used for most of Linux source code compilation.

MSVC, Microsoft Visual C++, is a proprietary compiler now part of the Visual Studio product suite. MSVC has a good "out of the box" support on Windows systems. A comparison table of C++ compilers on Wikipedia[3] states that MSVC fully supports the latest C++20 standard.

The third option is the LLVM project, which is a collection of compiler and toolchain technologies, available under Apache license. The LLVM official compiler for C and C++ languages `Clang` has fast compilation times and low memory usage compared to the alternatives. I chose Clang later during development because of its performance and requirements of one DSP library (see section 3.5).

CMake

CMake software is considered to be the standard for build automation, dependency management, testing, and distribution of C/C++ projects. A `CMakeLists.txt` file declares the sources, external libraries, software metadata, directories and compiler flags. Even though the JUCE framework is centered around Projucer, its custom project management system, I opted for CMake. Projucer is a convenient tool to quickly setup demo projects. If a project is to include more dependencies, CMake will in the end be the better option. For

[2] Lifecycle hooks include the `prepareToPlay`, `processBlock` and `releaseResources` functions. They are triggered at different points during the plugin's operation.

[3] https://en.wikipedia.org/wiki/List_of_compilers#C++_compilers

a beginner looking to use JUCE with CMake, I recommend the CMake template shipped with JUCE in the `examples` directory[4].

3.5 KFR framework

The audio processing algorithms that are described in chapter 5.1 require the Fourier Transform. There are many C++ libraries available that provide FFT functions. JUCE also comes with FFT and windowing functions useful for STFT (short time Fourier transform). JUCE's FFT functions are not as optimised as those found in specialised libraries, and integrating them with the rest of the code is not trivial. The following FFT libraries were considered:

- djfft[5]: header-only, GPU acceleration of multidimensional FFT
- FFTW[6]: used extensively, good performance, relatively old
- FFTW++[7]: built on FFTW3 with a simpler interface
- KISS FFT[8]: focused on simplicity
- KFR[9]: a DSP framework with more features
- Intel oneMKL[10]: open source interfaces for optimized computations
- PFFFT[11]: fast and lightweight FFT for slower hardware

The open source KFR framework was chosen, which offers more features than just FFT. It comes with a paid license for commercial applications. It has helpful documentation and examples. The FFT performance, based on the presented data, is very good.

Performance is also depends on the compiler. Clang is the recommended substitute for GCC and MSVC. The framework consists of several modules, imported optionally. It includes the functionality for computing 1D and multidimensional FFTs of arbitrary size and the real FFT and real inverse FFT. KFR is optimised for different processor architectures, which are automatically detected. Further optimisation is possible, for example by caching the FFT plan for repeated transformations [7].

KFR's `DSP` module enables FIR and IIR filters, windowing functions, sample rate conversion, loudness measurement. `IO` functions serve the task of audio file reading and writing in the formats WAV, FLAC, MP3 [7]. I find the smaller utilities and helper functions particularly useful, including those for buffers, mathematical and SIMD operations. The syntax is very close to simple C++ constructs, therefore the library is intuitive to use.

[4] https://github.com/juce-framework/JUCE/tree/master/examples
[5] https://github.com/jdupuy/dj_fft
[6] https://fftw.org/
[7] https://github.com/dealias/fftwpp
[8] https://github.com/mborgerding/kissfft
[9] https://kfrlib.com/
[10] https://github.com/oneapi-src/oneMKL
[11] https://bitbucket.org/jpommier/pffft/src

Chapter 4

Effect design

4.1 Proposed design

Figure 4.1 shows the proposed effect design. At its core is the LPC vocoder effect. The effect has two input signals, voice and excitation. Excitation is to be routed using the sidechain as explained in section 1.3 and is highlighted using blue color. There is a single output, with the resulting synthesised signal.

Preceding the `LPC` block are three `pitch shift` blocks in parallel. These are distinct effects that can change the pitch of the voice and add more pitched voices (see section 4.5). The rest of the blocks work to keep the signal stable and clear. The `match RMS block` is described in the following chapter 5.4. The `dry / wet` control adjusts the mix ratio of the original and processed voice.

The processing chain is designed so that all effects are optional. For example, the user can choose to only use `LPC` with no pitch change or vice versa. The `dry / wet` control can disable any mix-in of the processed signal. The parameters controllable by the user are listed in the user interface description (section 5.5).

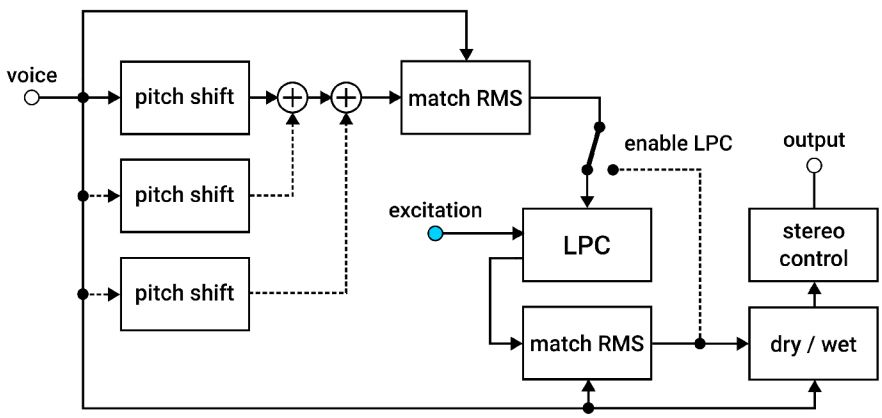

Figure 4.1: Block diagram of the complete effect chain, including effect extensions.

4.2 LPC effect design

The LPC vocoder effect, denoted in figure 4.1 as the `LPC` block, consists of four algorithmic components. First, the autocorrelation algorithm is applied to the input frames. The resulting autocorrelation coefficients are the input of the Levinson–Durbin recursion, which calculates the prediction coefficients.

An IIR filter is at the end of the chain. Provided with the residual and the prediction coefficients a, the filter's output will be the synthesised voice signal.

For the cross-synthesis effect, an excitation signal is processed in a different branch. This is necessary to obtain the residual signal from the excitation. Two signals therefore represent the effect's input; the voice and the excitation's residual. The residual signal can be obtained by convolving the excitation signal with its own prediction coefficients. Note that the convolution needs to produce the same number of samples as is the input.

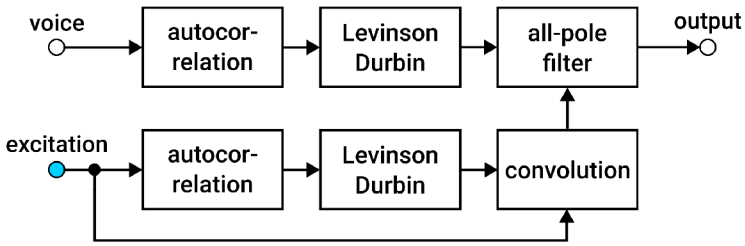

Figure 4.2: Block diagram of the LPC cross-synthesis effect. Audio frames are expected as the input.

This design is compatible with online processing. The higher-level logic outside this chain is assumed to have prepared audio frames for its input. The implementation details and algorithms are covered in chapter Implementation 5.

4.3 Data handling and processing

The tools used have been covered in chapter 3. The second half of this chapter discusses the design of the effect and key concepts in real-time processing. Some of the concepts are relevant to more areas of digital signal processing.

Audio data handling

An audio effect plugin typically processes an input signal to produce an altered output signal. A generator, also known as an instrument, might, for instance, take MIDI input and generate a new audio signal. In practice, plugins often feature multiple input and output buses, each with several channels. Data can be handled as audio files or as streams of audio blocks or individual samples. Offline processing occurs when dealing with input data of known boundaries, such as an audio recording. In contrast, online or real-time processing, must deliver the processed data within a very short time frame.

Real-time design is covered in dedicated section 4.4. For development purposes, it is advisable to start with prepared sound files. Using a sound recording of speech or music ensures the same input data for each run. I use WAV sound files with a sample rate of 44.1

kHz or higher. The processed sound can either be saved to a file or played back immediately after processing. The latter has the advantage of faster feedback.

Early in the design process, a decision had to be made between mono and stereo output. Some effects, such as the Little AlterBoy (section 1.4), are specifically designed for mono output. Changing a mono effect to stereo is usually as simple as processing each channel separately. For my effect, stereo processing was chosen for better sound quality – assuming the inputs are stereo.

Overlap-add

It was said previously that voice is a time-variant signal. For processing, speech has to be split into time segments short enough to align with the speed of change and long enough for resolution. From now on I will call these segments *frames*. The algorithms used in the LPC vocoder require such frames. One may observe that the frame-wise processing can introduce sudden transitions and unwanted artifacts where the frames meet. To address this issue, and also due to the possible spectral leakage of FFT, the frames are windowed and overlapped. This concept is called overlap-add. Each new frame begins within the preceding frame. The step, which is defined as a percentage of the frame's length, is called the hop size. The idea is illustrated in figure 4.3:

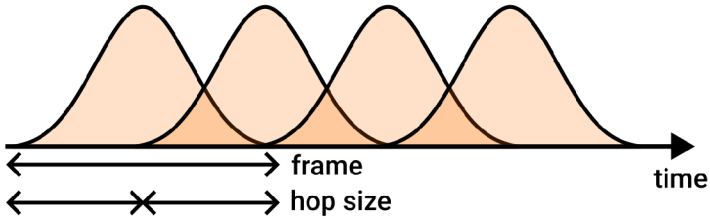

Figure 4.3: Overlap-add with 50 % overlap.

Each frame is first windowed by a windowing function. Then, the overlapping frames are added. A windowing function tapers off at the ends and has a maximum value in the center. Some windowing functions are: the Hann, Hamming, Blackman, Gaussian windows, which are all similar (Figure 4.4), and the more simplistic triangular, Bartlett, cosine windows.

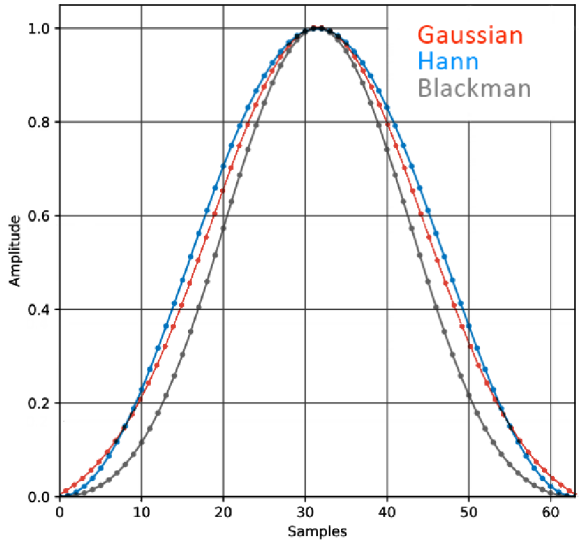

Figure 4.4: Windowing functions with normalised amplitude.

For OLA, the recommended functions are Hann and Hamming. The ideal overlap for these windowing functions is 50 % [2]. Windowing and overlap-add are also used in STFT to reduce the spectral leakage, in essence an aliasing. For my effect, Hann window was selected with a standard hop size equal to half of the frame length.

4.4 Real-time processing

To align better with the final design, the development was done with real-time use in mind. Although in my MATLAB and Python projects the data comes from a sound file, I choose to play the processed frames as soon as possible, each frame as it is finalised. In MATLAB the playback function `soundsc` plays the sound independently. After invoking the function, any processing continues without interruption.

In Python, the commonly used `sounddevice` package offers only a static instance. Subsequent calls to this instance make it reset and start playing the new sound. It is unusable for playing overlapping sounds. The package `simpleaudio` works with independent instances and supports stereo. It was possible to create a separate thread for each new frame to make the sounds overlap. This approach has another advantage of being able to calculate the time differences. If audio buffer underrun occurs, the program prints a warning, and if it finishes sooner (as it should) the main thread is paused for the remaining time of the frame.

In the first iterations, the programs were unable to process the frames fast enough even at a high CPU utilisation. Optimisation is an important topic reserved for the Implementation chapter 5. One may argue the frame-wise processing is not technically real-time. There is inherently a time period called latency, during which the algorithm waits to collect enough samples, and the additional time of processing. If we assume near-zero processing time, the latency is defined by the frame length at the given sample rate. Table 4.1 shows the minimum latencies with a frame size of 2048 samples at different sample rates. Unless solved otherwise, the effect will have latency in the tens of milliseconds, which might have a negative effect on live microphone use.

Table 4.1: Comparison table of sample rates, time and sample counts.

sample rate (kHz)	samples in 15 ms	2048 frame latency (ms)
8	120	256
22.05	331	93
44.1	662	46
48	720	43
96	1440	21

4.5 Effect extensions

The goals include designing extended functionality for the effect. It is logical to devise and integrate other effects that can be chained with or otherwise incorporated into the cross-synthesis LPC effect.

Pitch shifting

Shifting the pitch of a voice up or down may be achieved with various approaches. Simple example is playing a recording at a different sample rate. The result is a different perceived pitch, although the recording's duration will change. To retain the original duration, techniques like the PSOLA (pitch synchronous overlap-add) use overlapping frames. In PSOLA, the segments are moved closer together or further apart depending on the pitch shift factor. The frames are repeated to fill empty spaces, or eliminated to keep the original duration.

Each such technique based on OLA with interpolation is intended to address different phase and sound quality issues. Loss of phase coherence leads to transient smearing, noticeably softening the attack of sounds. Another studied issue called *phasiness* further degrades the sound quality, introducing a reverb-like effect. A phase not maintained across the frames can cause another alteration that makes the voice robot-like. A *robotisation* effect on the other hand may be intentional. Setting the phase in frequency domain to zero applies a fixed pitch onto a sound with virtually unchanged formant structure [11, 17].

Pitch shifters based on the FFT are more powerful. A discrete signal in the frequency domain comprises *frequency bins*. Pitch shifting in frequency spectrum means that the bins are shifted and wrapped around the edges. Reshuffling the array according to the shift factor before inverse transform demonstrates the basic principle.

A collection of audio effects published as *DAFX - Digital Audio Effects* [17] contains several examples of FFT-based pitch shifting in the chapter *Source-filter transformations*. The chosen algorithm achieves pitch shifting while preserving the original formants' positions. The output of the algorithm can be observed in figure 4.5. In essence, the formants are moved in the inverse direction of the spectrum's shift:

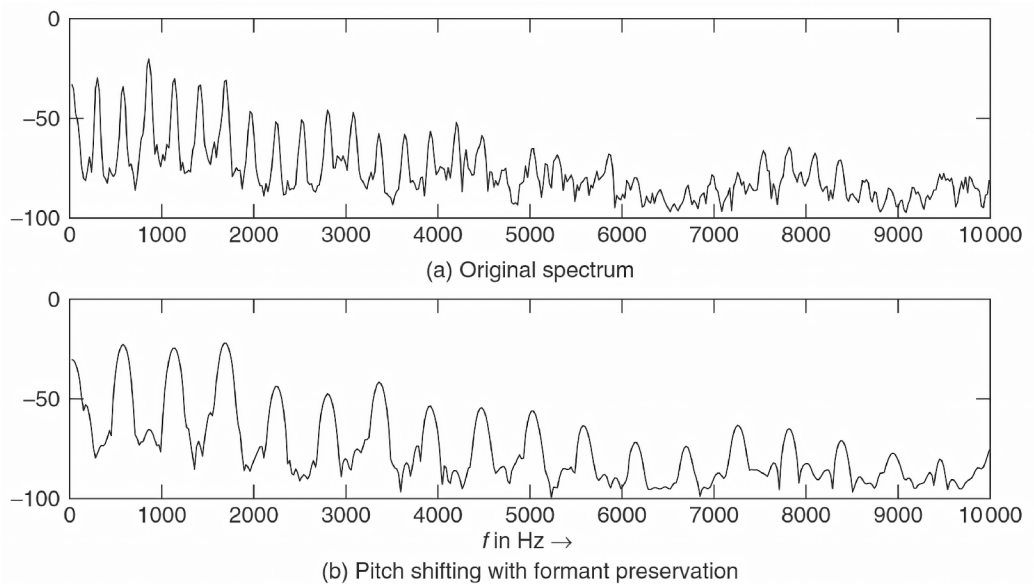

Figure 4.5: Example of pitch shifting with formant preservation. In spectrum (b), the pitch is increased by a factor of 2. Reproduced from [17].

The formants remain in approximately the same positions. The broadened resonance peaks contribute to the upward shift in pitch. This method effectively minimises phase-related issues, and the formant preservation helps to retain the intelligibility of speech.

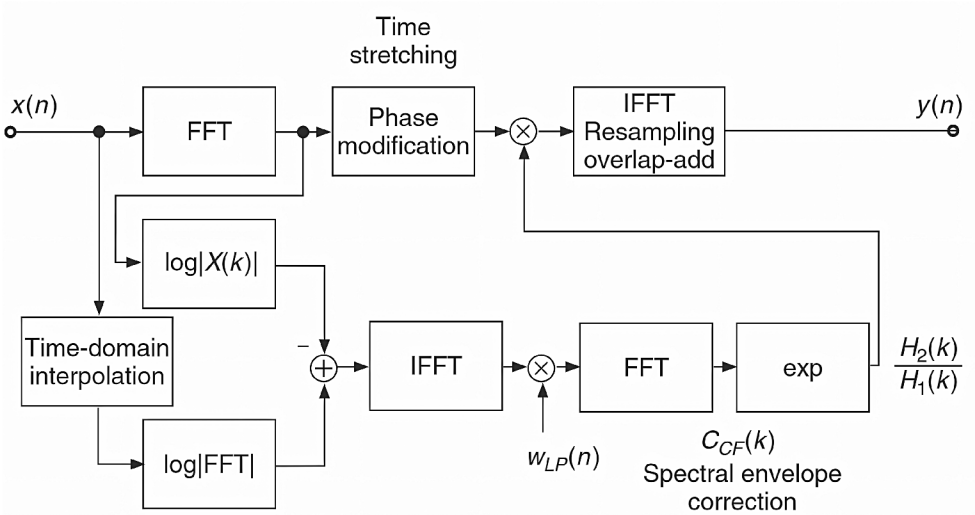

Figure 4.6: Block schematic of the pitch shifting algorithm. Reproduced from [17].

The block schematic 4.5 provides more details of the inner working of the algorithm. The $w_{LP}(n)$ is a "low-pass window". Although the principles remain, the code had to be rewritten and adjusted for use in real-time processing. Pseudocode of the algorithm 1 was recreated from the implementation:

```
Initialise parameters
Normalise input frame
for each grain of input frame do
    // grain size = 1/2 of frame size
    Window the grain
    Transform the grain using FFT
    Analyse phase information in frequency domain
    Retrieve original phase information
    Compute correction factor
    Apply correction to adjust for formant movement
    Reconstruct the processed grain using IFFT
    Interpolate grain and add to output
end
return output
```

Algorithm 1: Formant shift algorithm pseudocode

The algorithm was modified to support processing of short frames. The input frame size was kept consistent with the frame size used in the LPC effect. Pitch shifting is achieved by extracting grains from the input frame, windowing and applying interpolation, which is similar to OLA. Resampling is used to retain the original frame length.

The pitch shift factor and the overlap ratio are given by the ratio of the analysis and synthesis hop sizes. If these hop sizes are identical, no pitch shift occurs. Pitch shifting upwards is computationally more demanding, as it involves smaller analysis hop size and consequently a large number of overlapped grains.

Each grain is transformed into the frequency domain. Phase information of the unprocessed grain is obtained. The correction function computation is the most complex part of the algorithm and involves angle vectors. This correction function is used to move the formants in the inverse direction of the pitch shift.

Unison

An extension of the pitch shifting effect is a unison effect. Unison is achieved by having identical sound sources play "in unison" but with different effect parameters. This adds more perceptual layers to the voice signal. The three `pitch shift` blocks in figure 4.1 can create additional voices, each with its own pitch control. The voices are added to create a single signal again. The resulting voice signal can either be further processed by LPC or sent to playback.

Stereo to mono

In some instances, the LPC vocoder's output sound may lack focus and clarity. While processing stereo signals improves the perceived sound, it can be beneficial to limit the spatial characteristics. A technique called *stereo image widening* [15] was implemented. It adjusts the relative contribution of the `mid` and `sides`[1] of the stereo image. The controlling parameter is a factor ranging from 0 to 2. Values higher than 1 reduce the mid contribution, with a value of 2 fully suppressing a mono voice in a stereo track. The value was limited

[1] `Mid` is the mixed sound of left and right channels perceived in the middle of a stereo recording. `Sides` contribute to the width of the stereo image.

to values equal to or below 1. The user can control whether the output is stereo, mono or anything in-between.

```
// adjustable parameter
width = 0.8
// left and right channel data
L = in(:, 0)
R = in(:, 1)
// algorithm
side = 0.5 * (L - R)
mid = 0.5 * (L + R)
sideNew = width * side
midNew = (2 - width) * mid
newLeft = midNew + sideNew
newRight = midNew - sideNew
```

Algorithm 2: The algorithm of *stereo image widening* with an adjustable parameter *width* for controlling the stereo image. Reproduced from [15].

Chapter 5

Implementation

The chapter concerns mostly project-specific issues and the final effect.

5.1 Performance optimisation

All algorithms in this thesis were run exclusively on a CPU. A computer with the following components was used:

- CPU: Intel(R) Core(TM) i5-10300H @ 2.50 GHz.
- Memory: DDR4, 16 GB RAM

The reference stereo WAV files have a sample rate 44.1 kHz and 16 bit depth. The corresponding frame size is 2048 samples. In Python, FFT from the library `SciPy.fft` was used and in VST plugin written in C++, KFR's FFT was used.

Addressing performance issues is important to achieve stable real-time processing. In line with the principle of avoiding premature optimisation, the "root of all evil"[1], the discussion starts with algorithm complexity.

Algorithm complexity

A good metric to compare algorithm complexity is the *Big O notation*. $O(n)$ stands for *on the order of n*. Trying to optimise an algorithm makes little sense when algorithms of lower complexity achieving the same result exist. One example already mentioned is the Levinson–Durbin recursion exploiting the symmetricity of a Toeplitz matrix 2.4 as opposed to Gaussian elimination. The *Big O notation* has another benefit: "Doing an order-of-magnitude estimate of expected times in various parts of a program makes it easy, when you get real measurements of those times, to spot ones that differ substantially from your expectations" [12].

[1] https://geeksforgeeks.org/premature-optimization/

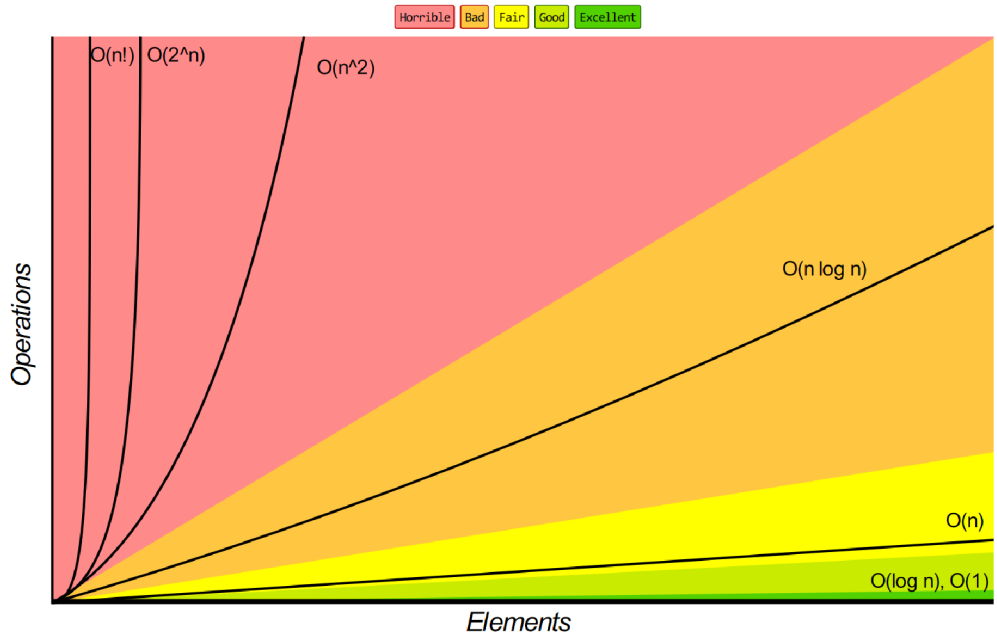

Figure 5.1: Big-O Complexity Chart visualises the worst-case algorithm complexity. Reproduced from [16].

For example, the Fast Fourier transform with complexity of a logarithm of base 2 is within the orange area, while a for-each loop computation typically has the complexity of $O(n^2)$, if not more.

FFT algorithms

Imperative style of code characterised by nested cycles severely underperforms in a real-time implementation. For the LPC vocoder code written imperatively, a rough estimate was made by iterating a counter in the processing chain for a frame of 1000 samples and the model order of 50. The number of cycle iterations per a single frame, assuming stereo, is around half a million. The algorithms were replaced with FFT-based solutions.

Autocorrelation is the most reused part of the chain. The Wiener-Khinchin theorem works with the fact that the PSD (power spectral density) and the autocorrelation of a signal are a Fourier transform pair [13]. Autocorrelation function $R(i)$ is the inverse transform $IFFT$ of a product of signal's FT and conjugate FT, $F^*(f)$ [13]. $S(f)$ is an interim value:

$$\begin{aligned} F(f) &= FFT[x(t)] \\ S(f) &= F(f)F^*(f) \\ R(i) &= IFFT(S(f)). \end{aligned} \tag{5.1}$$

The filter was the next block in need of replacement. An IIR filter is technically the inverse of convolution. Again, using FFT, it was implemented as a division of the FT of an excitation signal $c(t)$ and the FT of the prediction coefficients $a(i)$:

$$
\begin{aligned}
F(f) &= FFT[c(t)] \\
A(f) &= FFT[a(i)] \\
S(f) &= F(f)/A(f) \\
y(t) &= IFFT(S(f)).
\end{aligned}
\tag{5.2}
$$

Due to working with a stable all-pole filter, the division is not an issue. The last performance hindrance was the computation of residuals $r(t)$. A convolution of an excitation signal and its prediction coefficients can be computed using FFT as well. The only difference compared to the IIR filter is that the division is replaced by a product:

$$
\begin{aligned}
F(f) &= FFT[c(t)] \\
A(f) &= FFT[a(i)] \\
S(f) &= F(f)A(f) \\
r(t) &= IFFT(S(f)).
\end{aligned}
\tag{5.3}
$$

As a result, the processing complexity was reduced by several orders of magnitude as seen in Table 5.1. Reliable data was measured using a Python profiler. The figure shows the computation times before and after, clearly highlighting the importance of how to approach optimisation.

Table 5.1: Python profiler data of method run times of original (before) and FFT algorithms (after).

function	call count	before		after		own time per call **(ms)**	
		time (s)	own time (s)	time (s)	own time (s)	before	after
main	1	68.8	0	3.52	0		
autocorrelation	124	30.5	30.5	0.53	0.11	246	0.8
Levinson	124	49.1	18.6	0.95	0.39	150	3.2
IIR filter	62	16.4	16.4	0.06	0.02	265	0.3

The data includes both the total run time of methods and the actual computing time. For the measurements, thread pausing was disabled. Thus, the raw processing that previously took 68.8 s now takes only 3.5 s. Much of the improvement can be attributed to the FFT algorithms.

Streamlining C++

In the C++ codebase, in-place transform functions were used to speed up operations such as computing an exponential of a complex vector – used in phase computations in the pitch shift effect. Some of the SIMD operations from the KFR library were used for scalar operations. Passing references of vectors as method arguments help avoid the costs of data copying and it also makes the code more reusable (listing 5.2). Most data structures, namely vectors and windowing functions, are declared in headers for reusability and efficient resource allocation. Where necessary, vector data transfer is made with the use of pointers.

```
void mulVectorWith(univector<float>& vec1, const univector<float>& vec2) {
    std::transform(
        vec1.begin(), vec1.end(),
        vec2.begin(), vec1.begin(),
        std::multiplies<>()
    );
}
```

Listing 5.1: In-place scalar multiplication of two vectors using KFR's `univector` wrapper, and references as function arguments.

Multithreading

Multithreaded processor architectures are ubiquitous and present a potential to significantly speed up computation. As worker threads can work concurrently only on parallelisable tasks, the ratio of parallel to serial-only code is a defining factor. Amdahl's law [4] describes the maximum possible speed-up factor. The figure shows utilisation curves. Utilisation is the speedup divided by the number of processors.

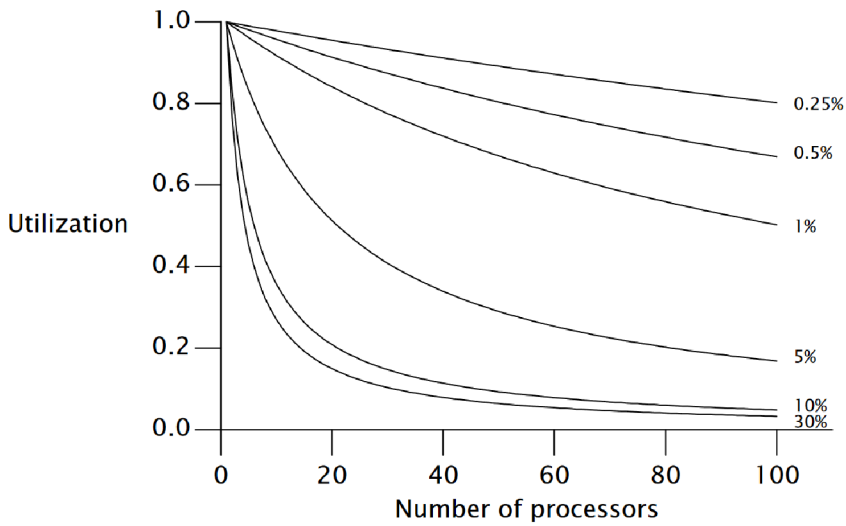

Figure 5.2: Maximum utilisation under Amdahl's law. Reproduced from [4].

The ratio of serial-only code heavily impacts the possible utilisation. Considering that my real-time DSP code is mostly serial, only smaller components could be effectively parallelised. Moreover, any shared resources would have to be properly synchronised with locking mechanisms to guarantee thread safety. Thread safety is crucial to prevent race conditions and deadlocks [4].

A degree of multithreading was implemented in an attempt to fix an unrelated issue. The code delegated left and right channel processing to two threads. After finding the root cause of the issue and with reference to Amdahl's law, the code was reverted to serial execution. The threading added complexity without real performance gains.

5.2 Frame collection

JUCE provides template code with a structure roughly corresponding to the VST specification. The method responsible for processing is called `processBlock`, as shown in listing 5.2:

```
void AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
juce::MidiBuffer& midiMessages) {
    for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
        ...
```

Listing 5.2: JUCE `processBlock` method's body example.

The loop iterates samples in the audio buffer. Write pointers reference the input buses and their channels. Frame processing requires a mechanism for sample collection and the processed frame has to be returned, again on a sample-by-sample basis. Moreover, the overlap-add of two frames has to be taken into account.

Minimum latency is specified by the frame size at a given sample rate (see section 4.4). When the plugin is instantiated, the first frame of samples has to be collected before any meaningful data can be returned. Until this data is available, the plugin may return zeroes or the input signal. I opted for the latter.

The implementation may use a ring buffer, FIFO (first in, first out) buffer or a basic buffer with indices. For overlap-add, at least two of these buffers will collect samples. With overlap, the speed of signal changes can be captured more effectively even though the latency remains similar.

My implementation uses two plain buffers with continuously updated indices. The logic is in the method `float LPCeffect::sendSample(...)` in the source code (see Appendix B).

5.3 Sample rates and frame sizes

Another important decision involves choosing the frame size. The options are limited by the fact that FFT is most efficient with sizes that are powers of two. Considering a sample rate of 44.1 kHz, the usable frame sizes are 512, 1024, 2048, and possibly 4096 samples, at least without the use of padding. Testing showed that a frame size of 2048 samples offers the best resolution and granularity at 44.1 kHz. Latencies of the 2048 sample window at different sample rates are shown in table 4.1.

DAWs can be set to different sample rates. Various DAWs can also come with different defaults. Using a constant frame size across different sample rates can introduce artifacts or, in the worst case, render the effect unusable. For the best user experience, it is important to make the effect sample rate-agnostic[2] Sample rate value can be obtained via the JUCE function `double AudioProcessor::getSampleRate()`. The value is propagated in constructors of the effect instances. Three different cases are recognised depending on the sample rate. The primary frame size can be 1024, 2048 or 4096 samples. Likewise, the pitch shift effect has three possible states of frame size and synthesis hop size (grain size is half the frame size). The effect was tested at all sample rates available in the FL studio DAW. As a result, the effect supports sample rates in the range of 22.5 – 96 kHz. Considering the most common sample rates are 44.1 kHz and 48 kHz [6], it is expected to perform well.

[2] A sample rate-agnostic system can handle any of the common sample rates.

5.4 Loudness

The `match RMS` blocks in figure 4.1 are necessary to keep the signal levels stable. In section 2.4 on linear predictive coding, we assumed a gain factor of 1. Libraries in MATLAB and Python use normalisation or dynamic control for sound playback[3]. In low-level C++, sample values must be kept within some limits. I opted for the most reliable and practical method: computing the energies of the signal to be corrected and of the reference signal as:

$$\mathrm{RMS}(\mathbf{x}) = \sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}} \tag{5.4}$$

Based on relative sizes of these powers, a correction factor is the ratio of the energies. The `input` signal's samples are multiplied by the factor so that its energy matches the energy of reference `RMS(ref)`:

$$\mathrm{input} = \mathrm{input} \cdot \frac{\min(\mathrm{RMS(ref)}, \mathrm{RMS(input)})}{\max(\mathrm{RMS(ref)}, \mathrm{RMS(input)})} \tag{5.5}$$

This correction is applied each frame. The reference is the original sidechained voice signal. In practice, both signals *input* and *ref* are a voice signal.

5.5 GUI

JUCE gives a lot of freedom regarding the graphical user interface (GUI) options. First, it provides tools for creating a native, cross-platform GUI. Second, there is the option to use an OpenGL rendering engine for advanced 2D or even 3D graphics. Lastly, `JUCE 8` introduced improved support for WebView, an embeddable browser technology used in progressive web apps and wrappers. This technology was selected.

When developing an interface for a browser rendering engine, the developer can use various JavaScript frameworks or simple HTML and CSS. Working separately from the backend on a local server with hot reloads greatly improves the development experience. However, there was one caveat: JUCE 8 was released just months[4] before the time of writing this thesis. The documentation was fairly limited. Fortunately, I was able to set up the framework correctly. After replacing the legacy WebView with WebView2 (on Windows), the plugin window can render a website just like a browser does.

The design was created iteratively in a vector graphics editor and the components were exported to the JavaScript frontend code. The built static files are inserted in a designated directory in C++. The crucial communication between GUI and C++, such as exchanging values of interactive controls, requires specific listeners in C++. All in all, modern web development and low-level C++ audio code can cooperate remarkably well. The design is inspired by a neumorphism style with minimal distractions. A lot of attention went to the controls, so that they have the feel of a native VST plugin controls.

[3] https://mathworks.com/help/matlab/ref/soundsc.html
[4] https://juce.com/blog/juce-8-feature-overview-webview-uis/

Figure 5.3: Effect's GUI during the review phase.

The effect chaining was made so that the controls can work independently and in any combination. The user can change any of the listed parameters:

- cross-synthesis on/off
- LPC model order
- stereo width
- dry/wet
- primary voice pitch shift
- a second (added) pitch shifted voice
- a third (added) pitch shifted voice

5.6 Technical issues

Issues in C++ codebase that cost the most time are listed below. Solutions are also stated, which could help others, should they encounter similar issues. The solutions refer to method names in the source code – see Appendix B.

Conditional checks in audio processing

Any conditional checks in `processblock` (located in `PluginProcessor.cpp`) usually distort the sound and should be avoided. Avoided should be also `float` number comparisons. Before trying to find performance issues, which can also manifest as stuttering audio, it is recommended to check for hard conditionals.

Audio bus overwriting

If pointers are used to access the audio buses, an issue can appear that I like to call *audio bus hijacking* – overwriting all previous data in the buses.

```
void processBlock (juce::AudioBuffer<float>& buffer,
juce::MidiBuffer& midiMessages) {
    auto mainInput = getBusBuffer(buffer, true, 0);
    auto* channelL = mainInput.getWritePointer(0);
    auto* channelR = mainInput.getWritePointer(1);
}
```

Listing 5.3: Acquiring write pointers to the main audio bus.

If the plugin never turns to an inactive state, even when it receives no input from DAW, it will continue overwriting the samples in the buses. This is very obvious in a DAW, where all other channels appear to have a disabled output. This happened because zeroes were written to the main bus when the plugin received no input. A quick solution in `processBlock` counts the sum of the frame's samples to determine whether to write any values. A sum close to zero bypasses the plugin.

JUCE WebView constructor ordering

An undocumented requirement in JUCE caused a large setback. An issue caused the WebView 5.5 window to crash after a second reload with memory access error code. Eventually, I localised the issue, invoked stack traces and reported the find. The error resulted from an incorrect order of code in the constructor of `PluginEditor.cpp`.

The lifetime of `WebSliderParameterAttachment` should be nested inside the lifetime of the `WebBrowserComponent` as shown in code listing 5.4. If the order is reversed, memory access errors may happen. The `relay` and `attachment` components are used as listeners of parameter changes originating from the GUI.

```
// PluginEditor.h
juce::WebSliderRelay modelOrderRelay;
juce::WebBrowserComponent webView;
juce::WebSliderParameterAttachment modelOrderSliderAttachment;
// PluginEditor.cpp
TheAudioProcessorEditor(TheAudioProcessor &p)
    : AudioProcessorEditor(&p), processorRef(p),
        modelOrderRelay{
            webView, "modelOrder"
        },
        webView{
            ...
        },
        modelOrderSliderAttachment{
            *processorRef.treeState.getParameter("modelOrder"),
            modelOrderRelay, nullptr
        }
{
    addAndMakeVisible(webView);
}
```

Listing 5.4: The correct constructor order for JUCE WebView with `relay` and `attachment` components.

Chapter 6

Testing and evaluation

6.1 Distribution

The effect was compiled using JUCE to the VST plugin format. Due to the different binaries' requirements and incompatibilities, only Windows-compatible plugin was available to testers. The effect was tested to work on Windows in FL Studio and Ableton DAWs. Setup steps for the DAWs were provided to testers in a manual and are also available in Appendix C.

6.2 User feedback

A simple survey was created to gather feedback from the reviewers. Four questions with a rating on a scale aim to assess the quality of key aspects:

Rate on a scale from 1 (worst) to 5 (best):

1. How good is the overall sound quality of the effect?
2. How unique and original is the effect?
3. How stable and performant is the effect in DAW?
4. How likely would you use the effect in music production or performance?

An open review form was the other part of the question form (see Appendix A). In total, 5 reviewers tested the effect. The summary is shown in Table 6.1.

Table 6.1: User ratings of key aspects

Question	**1**	**2**	**3**	**4**	**5**	**Average**
1. Sound quality	0	0	0	2	3	4.6
2. Originality	0	0	1	2	2	4.2
3. Stability and performance	0	0	1	4	0	3.8
4. Practicality	0	0	1	3	1	4

Chapter 7

Conclusion

7.1 Summary of work done

The goal of the thesis was to get acquainted with DSP techniques for speech signal analysis with a focus on LPC. The important voice characteristics like formants were examined and visualised.

Existing voice sound effects were mentioned, like the talk-box effect (section 1.1). The algorithms used in an LPC vocoder and its internal structure were discussed in detail. An effect design was proposed, including extensions like pitch shifting with formant preservation in chapter 4.

The effects were first implemented offline. They were tested and improved before an online implementation was written in C++. The real-time processing performance was analysed, improved and verified. DSP libraries and frameworks were integrated, namely JUCE (section 3.3) and KFR (section 3.5).

The proposed design was successfully implemented and built as a VST plugin with an embedded graphical user interface. All notable technical issues were documented for future reference.

7.2 Feedback evaluation

The quantitative ratings across the four key criteria (Sound quality, Originality, Stability and performance, Practicality) summarised in Table 6.1 show that the effect performs well. *Sound quality* received the highest average score, while *Stability and performance* had the lowest.

User feedback, as shown in the open review section (see Appendix A), highlights that users were impressed by the sound quality and the versatility of the sound effect. For example, Jan Dedík mentioned: *"I was amazed by how many outcomes I can do using these effects and various sound files."* The simple controls and intuitive user interface also received positive remarks.

There are areas for improvement, especially regarding the compatibility with other DAWs and plugins. Lukáš Novotný pointed out: *"The plugin could be better protected against incompatible I/O bus dimensions in REAPER."* Users also suggested a number of minor improvements.

Overall, the feedback is positive and constructive. The effect performs well in key areas like sound quality, but the integration with other DAWs is one of the weaknesses. Once

the issues are addressed, the effect will be published for use in music production and live performance.

7.3 Future work

Short-term work on the effect will involve addressing the reported bugs and issues. I may add new features and menus to the user interface, such as presets. The effect is planned to be published under an open source license. I will also develop better documentation in the process.

The possible long-term prospects include establishing a build pipeline on virtual machines to enable native distribution for MacOS and Linux. More advanced algorithms could be implemented to possibly reduce the effect's latency to only several milliseconds.

Bibliography

[1] COOLEY, J. W.; LEWIS, P. A. W. and WELCH, P. D. The Fast Fourier Transform and Its Applications. *IEEE TRANSACTIONS ON EDUCATION*, March 1969, vol. 12, no. 1. Available at: https://ieeexplore.ieee.org/document/4320436.

[2] DENG, L. and O'SHAUGHNESSY, D. *Speech Processing: a dynamic and optimization-oriented approach*. 1st ed. New York: Marcel Dekker, 2003. ISBN 08-247-4040-8.

[3] DOWNEY, A. B. *Think DSP: Digital Signal Processing in Python*. 1st ed. O'Reilly Media, Inc., 2016. ISBN 9781491938454. Available at: https://greenteapress.com/wp/think-dsp/.

[4] GOETZ, B. *Java concurrency in practice*. 1st ed. Upper Saddle River: Addison-Wesley, 2006. ISBN 03-213-4960-1.

[5] HOLMES, J. and HOLMES, W. *Speech synthesis and recognition*. 2nd ed. London: Taylor & Francis, 2001. ISBN 07-484-0856-8.

[6] iZotope, Inc.. Digital audio basics: audio sample rate and bit depth, February 2025. Available at: https://izotope.com/en/learn/digital-audio-basics-sample-rate-and-bit-depth.html.

[7] KFR Company. *KFR 6 framework documentation*. February 2025. Available at: https://kfrlib.com/docs/.

[8] MÉSZÁROS, T. *Speech Analysis for Processing of Musical Signals*. 2015. Master's thesis. Brno University of Technology - FIT.

[9] O'SHAUGHNESSY, D. Linear predictive coding. *IEEE Potentials*, February 1988, vol. 7, no. 1. Available at: https://ieeexplore.ieee.org/document/1890.

[10] RABINER, L. R. and SCHAFER, R. W. *Introduction to Digital Speech Processing*. Illustratedth ed. Now Publishers Inc., 2007. ISBN 978-1-60198-070-0.

[11] REISS, J. D. and MCPHERSON, A. P. *Audio effects: theory, implementation and application*. 1st ed. Boca Raton: CRC Press, 2014. ISBN 978-1-4665-6028-4.

[12] SITES, R. L. *Understanding Software Dynamics*. 1st ed. Addison-Wesley Professional, 2021. ISBN 9781491938454.

[13] STEIN, J. Y. *Digital signal processing: a computer science perspective*. 1st ed. New York: Wiley, 2000. ISBN 04-712-9546-9.

[14] Steinberg Media Technologies. *VST 3 API documentation*. February 2025. Available at: https://steinbergmedia.github.io/vst3_dev_portal/.

[15] TARR, E. *Hack audio: an introduction to computer programming and digital signal processing in MATLAB*. 1st ed. Routledge, 2018. Audio Engineering Society Presents. ISBN 9781138497542.

[16] The Big O Cheat Sheet. *Big-O Algorithm Complexity reference chart*. February 2025. Available at: https://bigocheatsheet.com/.

[17] ZÖLZER, U. *DAFX: Digital Audio Effects*. 2nd ed. Wiley, 2011. ISBN 9780470665992.

Appendix A

User reviews

The answers and names were included with the permission of testers. The answers have not been edited.

"The VST plugin LPC-vocoder was tested in the REAPER DAW using both the supplied test suite and own material. Based on its name, the primary function should be a vocoder. This particular plugin combines signals such that, when a voice or singing serves as the modulation signal and a carrier signal with a rich frequency spectrum is selected, the result is an intelligible output, which I consider desirable and pleasant sounding. A minor drawback could be the occasional artifacts, which may be caused by the chosen algorithm but can be reduced through filtering individual signals.

Additionally, the plugin offers the ability to alter pitch and add two additional voices at varying pitches. This feature broadens the plugin's range of options, and its sound quality is satisfactory when in use. Each voice could benefit from not only pitch settings but also gain control or an option to snap to a key and scale.

The plugin could be better protected against incompatible I/O bus dimensions in REAPER. Overall, this is a rich-sounding vocoder effect enhanced with pitch-shifting capabilities.

Lukáš Novotný
amateur guitarist

"The program works very well. At first, I was a little confused about how it should work, but after a while, it became super easy to use. All the effects work perfectly without any issues. It's really fun to distort your voice and create some cool effects that I or anyone else can use in their songs or sound projects.

One small issue I noticed is that there should be a guide or a small info button explaining when the effects are active. As I mentioned, I was initially confused about why the effects weren't changing the sound at all (LPC needs to be turned on, and DRY/WET needs to be applied).

Overall, I find this program both usable and beneficial for sound producers. It's easy to use, modern-looking, and intuitive."

Matúš Jacko
DnB, Pop and EDM enjoyer

"I tried this kind of sound effect for the first time in my life. As someone who doesn't have experience with such things, I liked the effect. Also, I had quite some fun. It's interesting how the effect can decompose human voice to make it completely unrecognizable. As an organist, it would be interesting to try it with some organ recording. I think that this effect has real use in the music industry."

Dominik Máca
pianist, organist, drummer, member of JIP band

"The effect's sound quality is very good but it depends heavily on the base sound and the voice quality. Compared to other vocoders I have heard, the quality is one of the best. It also has the additional pitch shifters. I like the simple UI. Other VST effects often have uninspiring UIs. It must have been a lot of work to create the effect.

The installation is standard but the setup a little more involved due to the sidechain. Performance is good even with my poor CPU. Using the effect with microphone made me laugh very hard. It is just good. To conclude, this is a fun and quite original VST effect."

Adam Bičiště
telecommunications student with DAW experience

"I have experience with DJing, but I tried something new in production software (FL Studio) and made some simple edits that I could use during my set. The biggest upside was the simplicity of the real-time voice adjustments. The controls are easy enough so that it doesn't require any background knowledge of the technical side.

Apparently, it is possible to connect my hardware mixer but we didn't try. I was amazed by how many outcomes I can do using these effects and various sound files."

Jan Dedík
up-and-coming DJ

Appendix B

Submitted files

`/compiled` – `.vst3` compiled binary for Windows.
`/programCode` – C++ source code.
`/soundExamples` – Example sounds produced by the plugin.

Source code compilation

The C++ code contains classes related to JUCE and classes with the processing logic. Every `.cpp` file has corresponding header file. The JUCE-related classes are `PluginProcessor.cpp`, `PluginEditor.cpp`. The processing classes are `LPCeffect.cpp` and `ShiftEffect.cpp`. Folder `public` contains JavaScript files for the GUI.

Dependencies:
- JUCE 8.0.1[1]
- KFR 6.0.2[2]
- Microsoft.Web.WebView2 1.0.X [3]

Compilers and toolchains:
- CMake 3.29.3[4]
- Clang from LLVM 19.1.0[5]
- Ninja 1.12.1[6]

First, `CMakeLists.txt` should be adjusted to one's system and file locations. After CMake initialisation and build, several JUCE configurations will appear. Use `VST3` to build the plugin.

[1] https://github.com/juce-framework/JUCE
[2] https://github.com/kfrlib/kfr
[3] https://developer.microsoft.com/en-us/microsoft-edge/webview2
[4] https://cmake.org/
[5] https://github.com/llvm/llvm-project
[6] https://github.com/ninja-build/ninja

Appendix C

User manual

This is a brief manual with steps to set up the plugin in FL Studio DAW or Ableton DAW on Windows.

FL Studio

1. Copy the `.vst3` file to the `/Program Files/Common Files/VST3/` folder.
2. In FL Studio, scan for new plugins by clicking `More plugins` → `Manage plugins` → `Find more plugins`.
3. Select the `LPC-vocoder` effect. Insert it into a mixer channel (see example in section 1.3).
4. Open `Plugin settings` in its top bar and then `Sound settings` → `Processing`.
5. Set `Sidechain` to input 1.

Ableton

1. Copy the `.vst3` file to the `/Program Files/Common Files/VST3/` folder.
2. In Ableton, open `Plug-Ins` and select the `LPC-vocoder` effect.
3. Open `Arrangement View`.
4. Insert the effect to a channel similar to the example in section 1.3.
5. In the plugin window, set `Sidechain` to a voice track.