HOCHSCHULE MITTWEIDA
University of Applied Sciences

BACHELORARBEIT

Herr
Kendy Drechsler

Entwicklung eines Audio-Plugins zur adaptiven frequenzselektiven Kompression in Max/MSP

2025

Fakultät: Medien

BACHELORARBEIT

Entwicklung eines Audio-Plugins zur adaptiven frequenzselektiven Kompression in Max/MSP

Autor:
Herr Kendy Drechsler

Studiengang:
Media and Acoustical Engineering

Seminargruppe:
MG19wA-B

Erstprüfer:
Prof. Dr.-Ing. Alexander Lampe

Zweitprüfer:
Prof. Dr.-Ing. Jörn Hübelt

Einreichung:
Mittweida, 17.01.2025

Faculty of Media

BACHELOR THESIS

Development of an audio plugin for adaptive frequency selective compression in Max/MSP

author:
Mr. Kendy Drechsler

course of studies:
Media and Acoustical Engineering

seminar group:
MG19wA-B

first examiner:
Prof. Dr.-Ing. Alexander Lampe

second examiner:
Prof. Dr.-Ing. Jörn Hübelt

submission:
Mittweida, 17.01.2025

Bibliografische Angaben

Drechsler, Kendy:

Entwicklung eines Audio-Plugins zur adaptiven frequenzselektiven Kompression in Max/MSP

Development of an audio plugin for adaptive frequency selective compression in Max/MSP

137 Seiten, Hochschule Mittweida, University of Applied Sciences,
Fakultät Medien, Bachelorarbeit, 2025

Abstract

Diese Arbeit beschreibt die Entwicklung eines Audio-Plugins zur dynamischen Anpassung von subjektiv als störend empfundenen Frequenzen in Max/MSP. Nach der Analyse von Anforderungen und Grundlagen wie Signalanalyse, Filterung und menschlichem Hörempfinden wurde ein spektral modifizierendes Plugin implementiert. Das Eingangssignal wird in überlappende Zeitabschnitte zerlegt, spektral analysiert und mittels Frequenzbewertungskurven sowie dynamischer Anpassung modifiziert. Tests zu den Wahrnehmungsindikatoren Lautheit und Schärfe zeigten frequenz- und zeitabhängige Kompressionseffekte, für Einzeltonmessungen jedoch auch Abweichungen der Amplituden bei idealisierten Testsignalen. Einschränkungen ergaben sich aus der datenstromorientierten Architektur von Max/MSP, insbesondere bei der linearen Faltungsoperation. Die Arbeit liefert erste Hinweise, dass das Plugin zur ausgeglicheneren Frequenzwahrnehmung beiträgt und Potenzial für Weiterentwicklungen in Max/MSP bietet.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API ..... Application Programming Interface
BIBO ..... Bounded Input Bounded Output
COLA ..... Constant Overlap-Add
CPU ..... Central Processing Unit
DAW ..... Digitale Audio Workstation
DFT ..... Diskrete Fourier-Transformation
DIF ..... Decimation in Frequency
DIT ..... Decimation in Time
DSP ..... Digital Signal Processing
DTFT ..... Zeitdiskrete Fourier-Transformation
EQ ..... Equalizer
FBS-Methode ..... Filter-Bank-Summation-Methode
FIR-Filter ..... Finite Impulse Response Filter
FT ..... Fourier-Transformation
FFT ..... Fast Fourier Transform
FFTW ..... Fastest Fourier Transform in the West
GUI ..... Graphical User Interface
IDFT ..... Inverse Diskrete Fourier-Transformation
IIR-Filter ..... Infinite Impulse Response Filter
IRCAM ..... Institut de recherche et coordination acoustique/musique
LU ..... Loudness Units
LUFS ..... Loudness Units relative to Full Scale
LZI-System ..... lineares zeitinvariantes System
M4L ..... Max for Live
OLA-Methode ..... Overlap-Add-Methode
OpenGL ..... Open Graphics Library
SIMD ..... Single Instruction, Multiple Data
STFT ..... Kurzzeit-Fourier-Transformation
VST ..... Virtual Studio Technology
WAVE ..... Waveform Audio File Format
WOLA ..... Weighted-Overlap-Add-Methode

Formelverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Der Einsatz moderner Technologien und komplexer Algorithmen in der digitalen Signalverarbeitung ermöglicht die Entwicklung neuartiger Tools für Toningenieure und Audio-Kreative. So bieten aktuelle Softwarelösungen spezifische Eingriffsmöglichkeiten in das Klanggeschehen von Musik- und Sprachaufnahmen. Mit ihrer Hilfe kann eine ausgeglichene Klangwahrnehmung erzeugt werden, was mit klassischen Audiobearbeitungswerkzeugen wie Equalizer (EQ) oder Kompressoren nur bedingt oder unter hohem Zeitaufwand zu bewerkstelligen ist. Praktische Arbeiten und Forschungsarbeiten zu diesen modernen Technologien tragen u.a. zur Erprobung von alternativen Entwicklungsmöglichkeiten dieser bei.

Altbekannte Tools erfordern die manuelle Einstellung ihrer Parametern wie z.B. Filterfrequenzen und Schwellwerten. Aufgrund der charakteristischen instationären Eigenschaft von Musik und Sprache ist für eine zeitgenaue Signalbearbeitung zudem häufig das Modulieren der Tool-Parameter über die Zeit notwendig. Hierzu ist eine gängige Methode das Zeichnen von Automationskurven in einer Digitalen Audio Workstation (DAW). Insbesondere in professionellen Umgebungen wie Tonstudios, im Broadcasting oder bei Live-Veranstaltungen, wo der Faktor Zeit eine maßgebende Rolle spielt, sind solche zeitaufwendigen Bearbeitungsschritte oft hinderlich und können nur bedingt umgesetzt werden. Daneben ist die Präzision der manuellen Bearbeitung eingeschränkt und resultiert mitunter nicht in dem gewünschten Klangergebnis.

Dynamische Resonanzunterdrücker (engl. Dynamic Resonance Suppressor) in Form von Audio-Plugins zur Einbindung in eine DAW sind aktuell im Trend in der modernen Audiowelt. Sie erkennen anhand der Messergebnisse von Frequenzanalysen für den Tontechniker interessante Spektralamplituden und störende Überhöhungen in den einzelnen Frequenzabschnitten. So errechnet die Software algorithmisch Filterprofile, welche sich dynamisch an das Klanggeschehen in der zu bearbeitenden Tonspur anpassen und dadurch einen Ausgleich der Klangwahrnehmung ermöglichen sollen. Durch die automatisierte Bearbeitung wird der Arbeits- bzw. Zeitaufwand im Prozess des Abmischens für Toningenieure geringer und führt folglich zur Steigerung der Effektivität, einhergehend mit einer qualitativen Verbesserung des Klangergebnisses. Dies birgt letztlich auch ökonomische Vorteile für den Tonschaffenden, da so Audioproduktionen schneller in der gewünschten Qualität umgesetzt und veröffentlicht werden können. Insbesondere für Podcasts oder zeitkritische Musikprojekte ist dies von Vorteil.

Kommerzielle Audio-Plugins basieren mehrheitlich auf dem Format der Virtual Studio Technology (VST) [1]. Dadurch lassen sie sich in beliebige DAWs integrieren. Die Programmierung erfolgt meist auf Basis der objektorientierten Programmierweise in der Spra-

che C++. Es stellt sich die Frage, wie die Implementierung eines Plugins zur dynamischen Resonanzunterdrückung in einer datenstromorientierten Entwicklungsumgebung wie Max/MSP [2] funktioniert. Um die Implementierung eines solchen Plugins zu erproben, ergibt sich die forschungsleitende Frage dieser Arbeit: Inwieweit lässt sich ein Audio-Plugin zur dynamischen Anpassung von subjektiv als störend empfundenen Frequenzamplituden in der Entwicklungsumgebung Max/MSP umsetzen? Das zu entwickelnde Plugin soll sich in Erscheinung und Parametrisierung an bereits bestehenden Anwendungen orientieren. Zunächst soll das eingegebene Audiomaterial automatisiert und entlang des Zeitverlaufs einer Spektralanalyse unterzogen werden. Anschließend sollen für den Nutzer mehrere Möglichkeiten zur weiteren Bearbeitung bestehen. Der Anwender soll in der Lage sein, zur Frequenzbewertung bekannte Kurven zu nutzen oder diese mittels EQ anzupassen. Auf Basis von einzugebenden Schwellwerten soll der Algorithmus Amplituden im Frequenzbereich als störend identifizieren. Der Nutzer soll nun Kompressionsparameter zum Umgang mit diesen Frequenzen definieren können. Eine benutzerfreundliche Oberfläche soll die Arbeit mit dem Plugin zusätzlich unterstützen. Um Parameteränderungen direkt hörbar zu machen und somit einen flüssigen Workflow bei der Anwendung zu unterstützen, soll das Plugin für die Applikation in Echtzeit konzipiert werden.

Die vorliegende Abschlussarbeit ist wie folgt aufgebaut: Zunächst dient eine Übersicht bestehender Audiobearbeitungs-Tools und eine Anforderungsanalyse der Ableitung der initialen Parameterwahl und zentralen Design-Entscheidungen. Eine umfassende Literaturrecherche lieferte Informationen zu den fachlichen Hintergründen der Arbeit. Im Anschluss wurde die Entwicklungsumgebung Max/MSP beschrieben und diese Auswahl begründet. Es folgte der Entwurf und die Implementierung unter Nutzung eines Ansatzes auf Basis der Kurzzeit-Fourier-Transformation. Anhand von praktischen Applikationsbeispielen wurde dann geprüft, ob und inwiefern sich das entwickelte Plugin für die Anpassung von spezifischer Lautheit und Schärfe als ausgewählte Indikatoren für eine ausgeglichene Klangwahrnehmung eignet. Die Arbeit schließt mit einer Zusammenfassung sowie einem Ausblick auf alternative Entwicklungsansätze ab.

2 Anforderungsanalyse und Design-Ansatz

Die Programmierung von Audio-Plugins stellt in der Praxis ein interdisziplinäres Anwendungsfeld im Bereich der Softwareentwicklung dar. Neben Kenntnissen in den Gebieten der digitalen Signalverarbeitung und der Audiotechnik auch Fähigkeiten im Design von Benutzeroberflächen notwendig. Vorausgesetzt ist demnach neben einer Entwicklungsumgebung für Audioanwendungen ebenfalls die Möglichkeit der Erstellung und Integration einer grafischen Benutzeroberfläche (engl. Graphical User Interface (GUI)). Im Zuge dieser Arbeit wurde sich hierbei für die datenstrombasierte Entwicklungsumgebung Max/MSP entschieden, da sie neben der Vielfalt an mitgelieferten Funktionen zur Signalverarbeitung und dem Monitoring auch umfangreiche Optionen zur Implementierung von GUIs beinhaltet. Unter Nutzung der Variante *Max for Live* ist zudem die Integration in die DAW *Ableton Live* [3] gegeben, um das resultierende Plugin anhand von Hörbeispielen testen und die Effizienz des Effekts bewerten zu können.

Die hier beschriebenen Aspekte sollen einen ersten Überblick über den Implementierungsansatz geben. Der ausgewählte Ansatz ist leitend für die in Kapitel 3 vorgestellten Grundlagen. Aus diesem Grund werden die hier beschriebenen Aspekte vorrangig aus praktischer Perspektive beleuchtet. Eine Erläuterung der Begriffe erfolgt dann im theoretischen Teil dieser Arbeit.

2.1 Traditionelle Audiobearbeitungs-Tools in der Praxis

In der Tonpraxis haben sich verschiedene Methoden zur Bearbeitung von Audiomaterial bewährt. Neben dem Schnitt von Audiodateien spielen Effekte wie Filter, Dynamikkompressoren, Echo- und Halleffekte sowie Kreativeffekte wie Phaser, Flanger oder Verzerrer eine wichtige Rolle [4]. Im Fokus für die gewählte Implementierung liegen die Funktionsweise von EQs und Kompressoren sowie dem Spectrum Analyser.

Equalizer

Der klassische EQ stellt eine Reihe an regelbaren Filtern für die Klangbearbeitung zur Verfügung, mit dessen Hilfe gezielt in das Frequenzspektrum eingegriffen werden kann [5]. So beschreibt Maempel [5, S. 14] die Relevanz dieser Bearbeitungstechnik, da Zusammenhänge zwischen dem Betragsspektrum eines Signals und z.B. dem Helligkeitseindruck oder dem Eindruck klanglicher Inhomogenität bestehen. Der EQ wird so genutzt, um empfundene Unebenheiten im Frequenzspektrum wie Überhöhungen bestimmter Frequenzbereiche oder einzelne Störfrequenzen zur Schaffung eines kohärenten Gesamtwerks auszugleichen. Aber auch die Anpassung der Klangfarbe spielt im Sound

Design eine wesentliche Rolle. So weist [5] auf die Applikation von Filtertechniken zur kreativen Anwendung hin, da insbesondere in der modernen Popularmusik im Gegensatz zur Kunstmusik derartige Mittel eher zur Klanggestaltung im Produktionsprozess eingesetzt werden.

Während der *grafische* EQ eine Filterbank an Bandpassfiltern mit fixen Mittenfrequenzen und bandspezifischer Verstärkungsregelung darstellt, ist der *parametrische* EQ deutlich freier in der Wahl von Grenz- bzw. Mittenfrequenzen, Verstärkungsfaktor (*Gain*) und der Güte Q. Letztere beinhalten meist eine Auswahl an Tiefpass-, Hochpass-, Peak-/Bell-, Highshelf- und Lowshelf-Filtern sowie ggf. Bandsperren oder Bandpässe. Bekannte parametrische Software-EQs wie [6] haben den Vorteil, dass die Filterparameter über Knotenpunkte mit der Computermaus auf dem GUI angepasst werden können. Das Prinzip ist vergleichbar mit Bézierkurven in der Computergrafik. Das GUI gibt dem Anwender ein nachvollziehbares visuelles Feedback durch die grafische Dartellung der Übertragungsfunktion des EQs in Echtzeit.

Kompressoren

Dynamikkompressoren hingegen bezeichnen Regelverstärker [5, S. 6 f.], welche bei Überschreitung des Signals über einen eingestellten Schwellwert (*Threshold*) ihren Verstärkungsfaktor gegensinnig ändern. Sie werden vorrangig zur Begrenzung der Dynamik eines Signals genutzt, aber auch in der kreativen Klanggestaltung in der Musikproduktion angewandt. Das statische Verhalten des Kompressors beschreibt die pegelabhängige Regelung. Der Parameter *Ratio* wird zur Einstellung des Verhältnisses von Eingangs- und Ausgangspegeldifferenz oberhalb des Schwellwerts verwendet [5, S. 7]:

$$R = \frac{\Delta L_{\mathrm{in}}}{\Delta L_{\mathrm{out}}} \tag{2.1}$$

Der Effekt unterscheidet sich dadurch von Limiter bzw. Clipper, welche keine Überschreitung des Schwellwerts am Ausgang zulassen. Der *Knee*-Parameter beschreibt dagegen eine Glättung des Übergangs in der nicht-linearen Kennlinie des Kompressors [5, S. 7].

Der dynamische Charakter des Kompressors wird im Zeitbereich deutlich. Nach [5, S. 8] beschreibt der *Attack*-Parameter die Ansprechzeit, die das System für das Einstellen der Reduktion nach Überschreitung des Schwellwerts benötigt; dagegen definiert die Rückstellzeit über den *Release*-Parameter, wie lang das System zur Rückkehr auf den Verstärkungsfaktor 1 benötigt. Bei langen Attack-Zeiten ist es möglich, dass Transienten nahezu unbearbeitet den Effekt passieren und ggf. zur Übersteuerung am Ausgang führen. Lange Release-Zeiten hingegen sorgen für einen starken Einfluss von Pegelspitzen auf den mittleren Pegel durch die lange Abklingzeit des Effekts [5]. Zu kurz gewählte Zeiten können insbesondere bei tieffrequentem Material zu hörbaren Verzerrungen führen.

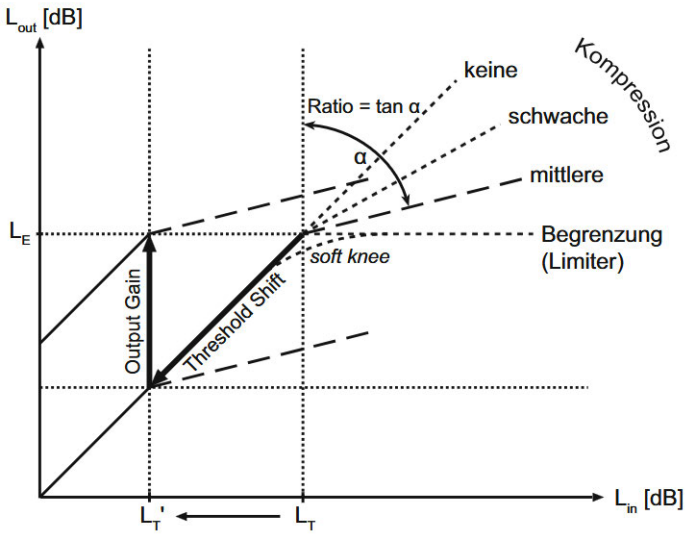


Abbildung 2.1: Nicht-lineare Kennlinie des Kompressors [5, S. 7]

Der Regelkreis, welcher die Hüllkurve des Eingangssignals zur Pegelbestimmung bildet, kann oft für Effekte wie dem *Ducking* durch ein zweites Signal von außen angesteuert werden. Dadurch wird die Kompression eines Signals durch die gemessenen Pegel eines zweiten Signals am sog. *Sidechain*-Eingang bestimmt. Dadurch werden Pumpeffekte in Popproduktionen [5] oder das temporäre Absenken von Hintergrundmusik bei erklingender Sprecherstimme ermöglicht.

Software-Kompressoren wie [7] bieten zudem eine Anpassung des Frequenzgangs des Signals im Regelkreis via Filter. Die gefilterten Frequenzen tragen dann in Summe weniger zum Auslösen der Kompression bei. Maempel [5, S. 10] erwähnt die Anwendung dieser Technik bei sog. *De-Essern*, welche durch diese Technik speziell auf den Frequenzbereich von vorkommenden S-Lauten eingestellt werden.

Frequenzanalyse mittels FFT

Die Analyse im Frequenzbereich mittels dedizierter *Spectrum Analyzer* wird in der praktischen Audiotechnik meist zur Bestimmung der spektralen Energieverteilung eines Signals angewandt. Aus Effizienzgründen basieren diese meist auf der Schnellen Fourier-Transformation (engl. Fast Fourier Transform (FFT)). Auf Basis der Analyseergebnisse kann der Toningenieur Entscheidungen zur weiteren Bearbeitung des Audiomaterials treffen. Störende Überhöhungen bestimmter Frequenzbereiche oder einzelne Resonanzen werden durch eine visuelle Darstellung der spektralen Amplituden sichtbar. Im Gegensatz zu Terzbandanalysatoren können sie bei entsprechend gewählter FFT-Länge aufgrund der höheren Auflösung einen genaueren Einblick in das Frequenzspektrum ermöglichen.

Moderne Analyzer wie [8] implementieren zudem typische Frequenzbewertungskurven

für bestimmte Anwendungszwecke. Dadurch entspricht die visuelle Darstellung eher dem Empfinden des Hörers. Der Toningenieur kann auf Basis dessen Entscheidungen treffen, welche sich mehr an seiner auditiven Wahrnehmung orientieren. Voraussetzung der korrekten Wahl einer Frequenzbewertungskurve ist jedoch das Bewusstsein des Tonschaffenden über Faktoren wie die Abhörlautstärke, Abhörposition oder Raumakustik. Die FFT wird auch zur kreativen Klanggestaltung genutzt. So beschreibt [9] eine Applikationsmöglichkeit am Beispiel des sog. *Phase Vocoders* in Max/MSP.

Stereokodierung

Mischpulte und moderne Plugins bieten oft die Möglichkeit, über eine Kanalmatrix von der klassischen Stereobearbeitung vom linken (L) und rechten Kanal (R) auf die Bearbeitung von Summen- (Mid) und Differenzsignal (Side) umzuschalten. Die Mid-Side-Kodierung wird nicht nur zur Umsetzung bestimmter Mikrofonierungsverfahren, sondern auch für spezielle auditive Effekte [5] angewandt.

2.2 Anforderungen an eine Plugin-Implementierung

Die Aufgabenstellung setzt eine Reihe an Faktoren für eine erfolgreiche Implementierung des Audio-Plugins voraus. So soll das Audiomaterial in Echtzeit auf seine Frequenzanteile analysiert werden. Dies fordert einen Algorithmus, welcher die Frequenzanalyse in Echtzeit aus technischer Sicht bewerkstelligen kann. Zum anderen muss er in der gewählten Entwicklungsumgebung Max/MSP bzw. Programmierparadigma umsetzbar sein. Da sich die Frequenzanteile über die Zeit ändern können, soll zudem die Analyse möglichst kurzer Abschnitte gegeben werden, um die Anpassbarkeit an temporale Änderungen im Signal zu gewährleisten. Echtzeitanwendungen unterscheiden sich hierbei insofern von sog. *Offline*-Applikationen, indem sie eingehende Werte direkt verarbeiten und nach kurzer Zeit ausgeben. Als nachteilig ist jedoch der Faktor Latenz zu betrachten. Komplexe Algorithmen wie die FFT erfordern zuerst die Ansammlung einer gewissen Anzahl an eingehenden Werten, um eine Verarbeitung zu ermöglichen. Es ist dabei je nach Applikation abzuwägen, inwiefern eine Verzögerung zwischen Ein- und Ausgangssignal zulässig ist. Offline-Applikationen hingegen können oft komplexere Analysen durchführen, da ein Ausblick auf Folgewerte möglich und eine möglichst sofortige Ausgabe der Berechnungen nicht erforderlich ist.

In diesem Zusammenhang spielt zudem die Beachtung der Belastung des Computerprozessors (engl. Central Processing Unit (CPU)), welcher die Berechnungen durchführt, besonders bei Echtzeitanwendungen eine wichtige Rolle. In der Praxis, wo in Mehrspuraufnahmen in einer DAW Audio-Plugins in hoher Anzahl simultan zum Einsatz kommen, führen zu hohe CPU-Belastungen meist zu Aussetzern in der Audiowiedergabe. Eine

Optimierung der CPU-Last kommt demzufolge der Benutzerfreundlichkeit zugute.

Die Benutzerfreundlichkeit und eine simple Bedienbarkeit sind zudem praxisrelevant, um den Zeitaspekt bei der Anwendung zu berücksichtigen. Eine Orientierung an bewährten Prinzipien und Parametern aus Abschnitt 2.1 ermöglicht dem Anwender ein rasches Verständnis der Funktionsweise des Plugins. Hierzu zählt auch die Übersichtlichkeit des visuellen Designs des GUI, welches maßgeblich zur Benutzerfreundlichkeit beiträgt. Auch die gezielte Gestaltung der Steuerbarkeit der Plugin-Parameter nützt diesem Aspekt.

2.3 Gewählter Design-Ansatz zur Implementierung in Max/MSP

Maempel [5, S. 24] erwähnt bereits die Kombination traditioneller Techniken zur Schaffung innovativer Audiobearbeitungs-Tools. Er erwähnt dabei aber auch die Relevanz der letztlichen Klangästhetik als Zielsetzung. Dies zeigt, dass übliche DSP-Blöcke wie Filter oder Analysetechniken auch in kreativer Weise zur Verwirklichung des gewünschten Klangs miteinander verschaltet werden können. Der hier vorgestellte Ansatz nutzt ebenfalls bekannte Prinzipien aus Abschnitt 2.1, um die gegebene Zielstellung zu erreichen.

Das grundlegende Prinzip des zu entwickelnden Plugins ist in Abbildung 2.2 dargestellt. Auf Basis eines momentanen Eingangsspektrums wird adaptiv eine Übertragungsfunktion generiert, die Frequenzamplituden entgegenwirken soll, welche über einen eingestellten Schwellwert ausschlagen. Der Schwellwert (Threshold) dient dem Anwender als Orientierungspegel, ab welchem die Frequenzen als störend erkannt werden. Jedoch werden nicht immer alle Frequenzen mit hohen Amplituden vom Anwender je nach Applikation und gewünschter Klangästhetik auch wirklich als störend angesehen. So sollen individuelle Frequenzbewertungsmöglichkeiten dazu dienen, bestimmte Frequenzbereiche empfindlicher oder unempfindlicher für Schwellwertüberschreitungen zu machen. Dadurch wird es dem Anwender möglich sein, den Fokus der spektralen Kompression auf z.B. jenen Frequenzbereich von als *scharf* empfundenen S-Lauten zur Applikation im De-Essing zu setzen. Auch kurzzeitige, wahrgenommene Überhöhungen von Frequenzbereichen sollen so ausgeglichen werden, wo statische EQ-Filter stattdessen manuell automatisiert werden müssten.

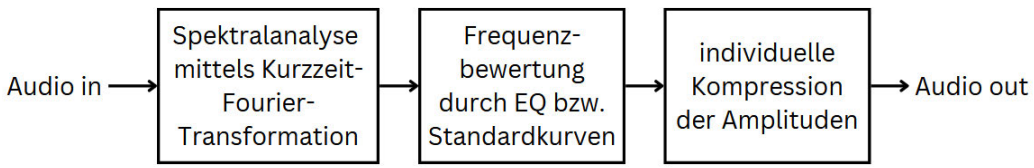


Abbildung 2.2: Grundlegendes Funktionsprinzip des Plugins (Eigene Darstellung)

Frequenzanalyse

Zu Beginn wird das Audiosignal auf seine Frequenzanteile hin untersucht. Die Kurzzeit-Fourier-Transformation stellt eine effiziente und ressourcenschonende, nativ integrierte Variante zur Spektralanalyse in Max/MSP dar. Ein Vorteil dieser Methodik ist die Wahl der Frequenzauflösung, welche durch die FFT-Länge bestimmt wird. Gleichzeitig ist die zeitliche Auflösung, d.h. die Frequenz, mit welcher nachfolgende Spektren aufgenommen werden, mithilfe eines Overlap-Faktors einstellbar. Die Parametrisierung erlaubt eine individuelle Anpassung dieser Kenngrößen an verschiedene Anwendungssituationen wie der Detektion einzelner, temporär störender Frequenzamplituden.

Frequenzbewertung

Die analysierten Frequenzamplituden sollen mittels EQ und standardisierten Kurven bewertet werden. Die Implementierung eines parametrischen EQs bietet sich an, da das Prinzip einen in der Praxis bekannten Ansatz darstellt. Eine individuelle Bewertung der Frequency Bins kann aufgrund der hohen Anzahl dieser bei hohen Frequenzauflösungen den Workflow gefährden. Stattdessen können die Werte der Übertragungsfunktion des parametrischen EQs an diesen Frequenzen für eine zeiteffiziente Bewertung genutzt werden. Die Implementierung verschiedener Filtertypen wie Peak-, Bandpass- oder Hochpass und Tiefpassfilter bietet sich an, um bestimmte Frequenzbereiche nahezu gänzlich vom Kompressionsprozess auszuschließen oder andere stattdessen zu fokussieren. Eine Steuerung des EQs über Knotenpunkte auf dem GUI gewährt dem Toningenieur außerdem einen von Software-EQs bekannten Workflow.

Die Möglichkeit, in der Praxis standardisierte Frequenzbewertungsfunktionen auszuwählen, ist zudem in Situationen sinnvoll, wo der Toningenieur unter eingemessenen Bedingungen arbeitet. Die Anwendung des Plugins soll dann ein ausgeglichenes Frequenzspektrum im Sinne der Klangwahrnehmung ermöglichen. Die Effizienz dieser Methode wird jedoch nicht in dieser Arbeit geprüft und lediglich in das Plugin als Option implementiert. Der Fokus liegt hier auf der Frequenzbewertungsoption mittels EQ.

Kompression der Frequenzen

Die frequenzabhängige Erkennung und Anpassung der Amplituden wird über einen dem Dynamikkompressor ähnlichen Prinzip erfolgen. Die momentane Übertragungsfunktion des Systems generiert sich aus diesem Algorithmus. Der Threshold dient als Schwellwert, *ab welcher Amplitude* eine Frequency Bin komprimiert wird. Ratio und Knee geben an, *wie stark* die Kompression wirkt. Um starke Schwankungen zwischen sukzessiven Spektren auszugleichen und ggf. Artefakte durch Diskontinuitäten vorzubeugen, dienen Attack- und Release-Parameter als zeitliche Glättung der Kompression. Jede Frequency

Bin verfolgt dabei auf Basis der Umsetzbarkeit in Max/MSP, wie in den folgenden Kapiteln beschrieben, ihre eigene Hüllkurve.

Am Ende erfolgt die Kalkulation des Ausgangssignals durch die Verrechnung des generierten Filterprofils mit dem Spektrum des Eingangssignals. Erwartet wird am Ausgang ein Signal, dessen Frequenzamplituden um den Schwellwert herum ausgeglichen wurden. Eine leichte Überschreitung ist je nach eingestellter Kompressionsrate möglich. Ein hartes Clipping im Spektralbereich kann die Wahrnehmung bestimmter Klangstrukturen wie die Verteilung der einzelnen Harmonischen eines Musikinstruments gefährden und den Klang stark verfremden.

Ein Vorwort zum psychoakustischen Effekt des Plugins

Verschiedene Faktoren erschweren eine objektive Bewertung dessen, was als klanglich angenehm, ausgeglichen, harsch oder inhomogen wirkt. So spielen neben subjektiven Vorlieben und Hörgewohnheiten auch technische Einflussfaktoren wie der Frequenzgang eines Lautsprechers, binaurale Effekte sowie Resonanzen und Kammfiltereffekte durch die Raumakustik und Abhörposition eine wichtige Rolle. Der in dieser Arbeit geprüfte Effekt auf die Schärfe soll die Wirkung auf Zischlaute verdeutlichen. Die Schärfe ist bei Zischlauten der menschlichen Stimme aufgrund der starken Anteile in den hohen Frequenzen des menschlichen Hörbereichs ausgeprägt und lässt darauf schließen, *wie scharf* ein Geräusch wirkt. Die spezifische Lautheit ist dagegen ein Indikator für die Intensität der Wahrnehmung des Frequenzbereichs eines Geräusches und kann zeigen, *wie laut* die Frequenzbereiche wahrgenommen werden. Es soll gezeigt werden, ob sich hohe Werte der spezifischen Lautheit spezieller Frequenzbereiche mit dem Plugin reduzieren lassen. Damit verbunden ist die sog. Maskierung bzw. Verdeckung. In der Audiopraxis wird mit einem angenehmen und ausgeglichenen Klangbild oft die Hörbarkeit von Details referenziert, was durch ausgeprägte Anteile in einzelnen Frequenzbereichen erschwert wird. Diese maskieren umliegende Töne, was meist nicht gewünscht ist.

3 Theoretische Grundlagen

Die Analyse und Verarbeitung von Signalen in digitalen Systemen setzen Kenntnisse der Signal- und Systemtheorie voraus. Diese stellt grundlegende mathematische Verfahren bereit, wonach sich Signal- und Systemparameter sowie Methoden im Forschungsgebiet der digitalen Signalverarbeitung (engl. Digital Signal Processing (DSP)) ableiten lassen. Die für die Entwicklung des Audio-Plugins relevanten Grundlagen sollen hier beleuchtet werden. Dies sind insbesondere Aspekte der Frequenzanalyse mittels Fourier-Transformation sowie digitale Filter. Zudem sollen für die anschließenden Applikationsbeispiele die akustischen Grundlagen erarbeitet werden. Im Folgenden werden zur Unterscheidung die Schreibweisen mit runden Klammern für kontinuierliche Funktionen wie $x(t)$ und eckige Klammern für diskrete Zahlenfolgen wie $x[n]$ verwendet.

3.1 Signalanalyse im Frequenzbereich

Für ein Signal existieren verschiedene Analysemethoden im Frequenzbereich. Ein gängiges Verfahren ist die Anwendung der Fourier-Transformation (FT) und ihrer Varianten. Diese lässt sich in der Praxis unter Berücksichtigung bestimmter Voraussetzungen und Einschränkungen in digitalen Systemen wirksam implementieren. Daneben existieren weitere Methoden wie die Spektralanalyse über Filterbänke, welche je nach Anwendungsfall in Betracht gezogen werden können. Das Ziel dieser Arbeit beinhaltet jedoch nicht den Vergleich solcher Verfahren und beschränkt sich daher auf die Anwendung der FT. Für die Implementierung in ein Computerprogramm ist besonders die Diskrete Fourier-Transformation (DFT) von großem Interesse. Diese steht in enger Verbindung zu ihrem kontinuierlichen Pendant.

3.1.1 Fourier-Transformation

Nach der Theorie der Fourier-Analysis lassen sich Signale als eine Summe mehrerer Sinusschwingungen unterschiedlicher Frequenz, Amplitude und Phasenlage darstellen. Als Fourier-Transformierte eines zeit- und wertkontinuierlichen Signals $x(t)$ wird dessen Spektralfunktion bezeichnet [10], [11]. Diese beschreibt die Verteilung dieser Sinusschwingungen, den Spektralkomponenten, im Frequenzbereich.

$$X(\omega) = \int_{-\infty}^{\infty} x(t)\mathrm{e}^{-j\omega t}dt \tag{3.1}$$

Hierbei stellt ω die Kreisfrequenz dar, welche als $\omega = 2\pi f$ definiert ist [12]. Die resultierende komplexwertige Exponentialfunktion $\mathcal{F}\{x(t)\} = X(\omega) = |X(\omega)|e^{j\varphi(\omega)}$ wird Spektrum [10] oder auch Bildfunktion [11] des Signals $x(t)$ genannt. Aus dieser lassen sich das Betragsspektrum $|X(\omega)|$ und das Phasenspektrum $\varphi(\omega)$ extrahieren. In der Messtechnik

wird das Betragsspektrum typischerweise auch als Amplitudenspektrum bezeichnet und ermöglicht Rückschlüsse auf die Energieverteilung eines Signals in Abhängigkeit von der Frequenz. Betragsspektren werden hierbei meist logarithmiert in Dezibel mithilfe der Formel

$$|X(\omega)|_{\mathrm{dB}} = 20 \log_{10} (|X(\omega)|) \tag{3.2}$$

abgebildet. Das Phasenspektrum hingegen beschreibt die Phasenverschiebungen der einzelnen Spektralkomponenten zueinander. Die Korrespondenz zwischen einem Signal im Zeitbereich und seiner zugehörigen Fourier-Transformierten wird formuliert als:

$$x(t) \circ\!\!-\!\!\bullet\, X(\omega) \tag{3.3}$$

Diese Darstellung lässt auf eine Reversibilität der FT schließen. Die Umkehrung wird als inverse FT bezeichnet, wodurch sich zu einer gegebenen Funktion im Bildbereich ein Signal im Zeitbereich berechnen lässt. Diese ist auch als Synthesefunktion bekannt [12].

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega) \mathrm{e}^{j\omega t} d\omega \tag{3.4}$$

Wie von Goebbels & Ritter [13, S. 345] bewiesen, sind insbesondere für rein reellwertige Signale, wie sie in der Praxis meist anzutreffen sind, spezielle Symmetrieeigenschaften zu berücksichtigen. Die Spektralfunktion ist in diesem Zusammenhang eine komplex konjugierte Funktion. Für eine positive Frequenz f existiert stets die negierte Frequenz $-f$, sodass der Zusammenhang $X(f) = X^*(-f)$ gilt [10]. Dadurch kann das Betragsspektrum als eine gerade Funktion und das Phasenspektrum als eine ungerade Funktion beschrieben werden [12]. Diese Eigenschaften können bei der Entwicklung von Algorithmen zur effizienten Kalkulation von Frequenzspektren durch Computersysteme einbezogen werden. Von Interesse sind in der praktischen Tontechnik, z.B. bei der Analyse des Frequenzbereichs von akustischen Signalen, jedoch lediglich die positiven Frequenzen. Die zur y-Achse symmetrischen Informationen im negativen Frequenzbereich werden als redundant angesehen und auf vielen sog. Spektrumanalyzern nicht dargestellt.

In Abbildung 3.1 ist die Korrespondenz eines Rechteckimpulses zu seiner Spektralfunktion, der Sinc-Funktion, dargestellt. Ersichtlich wird hier das Verhältnis von Impulsbreite und der Stauchung bzw. Streckung der Sinc-Funktion. Je kleiner T gewählt wird, desto schmaler und höher wird der Rechteckimpuls. Die Fläche unter der Funktion, zu berechnen über die Integration, bleibt konstant gleich Eins. Gleichzeitig wird dadurch die korrespondierende Sinc-Funktion gestreckt. Läuft T gegen Null, so strebt die Breite des Impulses gegen Null, seine Höhe gegen unendlich und die Abstände der Nulldurchgänge der Sinc-Funktion gegen unendlich. Daraus ergibt sich der sog. Dirac-Impuls und seine

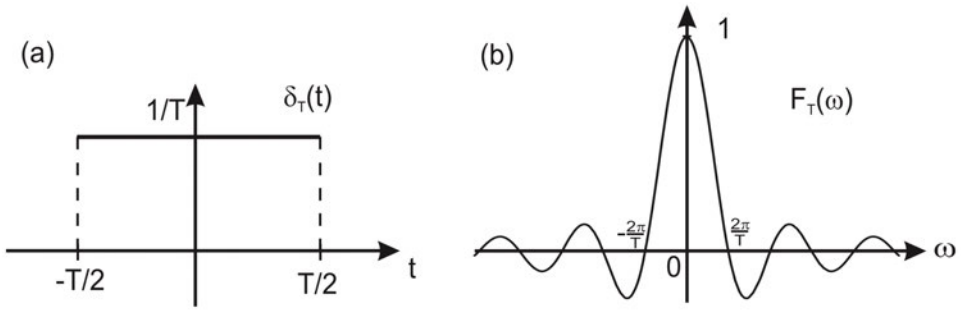


Abbildung 3.1: Korrespondenz eines Rechteckimpulses im a) Zeitbereich und seiner b) Spektralfunktion [14, S. 646]

Spektralfunktion mit $X(\omega) = 1$. Dieser ist in der Praxis nur annähernd darstellbar. Stattdessen wird für Berechnungen in realen, diskreten Systemen der Einheitsimpuls definiert [11, S. 183]:

$$\delta[k] = \begin{cases} 1 & \text{für } k = 0 \\ 0 & \text{für } k \neq 0 \end{cases} \tag{3.5}$$

3.1.2 Zeitdiskrete Fourier-Transformation

Die FT bezieht sich in ihrer Integralschreibweise auf Signale im kontinuierlichen Zeit- und Wertebereich. Signale werden in digitalen Systemen hingegen als zeit- und wertdiskrete Zahlenfolgen $x[n]$ angenommen, da solche Systeme nicht mit kontinuierlichen Größen wie elektrischen Spannungen und Strömen rechnen können. Für diesen Fall lässt sich die zeitdiskrete Fourier-Transformation (engl. Discrete-Time Fourier Transform (DTFT)) ableiten [15].

$$X(\Omega) = \sum_{n=-\infty}^{\infty} x[n] \mathrm{e}^{-j\Omega n} \tag{3.6}$$

Hierfür wird ein zeitkontinuierliches Signal durch die Abtastung (engl. Sampling) von Funktionswerten an äquidistanten Stützstellen in eine zeitdiskrete Folge überführt. Dabei wird das Zeitintervall T_A als Abtastintervall und der Kehrwert $f_A = 1/T_A$ als Abtastfrequenz charakterisiert [16, S. 113]. Technisch realisiert wird der Prozess der Abtastung in Analog-Digital-Wandlern über sog. Abtast-und-Halte-Schaltungen (engl. Sample-and-Hold) [17]. Diese speichern mit jedem Intervall T_A den momentan abgetasteten Signalwert in Form einer elektrischen Spannung bis zur erneuten Entnahme eines Wertes. Um ein über die Zeit diskretisiertes Signal korrekt rekonstruieren zu können, muss die Abtastfrequenz hierbei mehr als das doppelte der im Signal enthaltenen maximalen Frequenz betragen. Der Zusammenhang

$$f_A \geq 2 \cdot f_g \tag{3.7}$$

ist als Shannon-Nyquist-Theorem bekannt [18]. Die halbe Abtastfrequenz wird auch als Nyquist-Frequenz bezeichnet, wobei diese größer als jede der im Signal enthaltenen Nutzfrequenzen sein muss. Das Konzept wird anhand der Darstellung der Spektralfunktionen für zeitdiskrete Folgen deutlich.

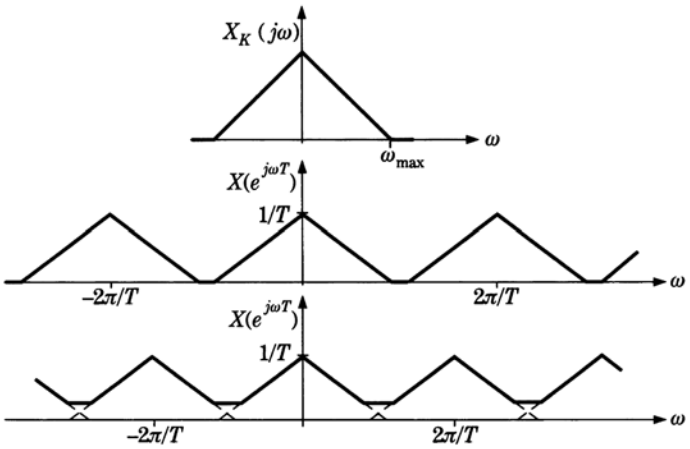


Abbildung 3.2: Darstellung der Periodizität des Spektrums zeitdiskreter Signale [15, S. 23]

Im Gegensatz zur zeitkontinuierlichen FT ist das Spektrum der DTFT nach Kammeyer & Kroschel zwar ebenso kontinuierlich, allerdings auch 2π-periodisch [15, S. 18]. Nach Gomes & Velho lässt sich das Spektrum eines abgetasteten Signals aus der Summe des dem originalen zeitkontinuierlichen Signal zugehörigen Spektrums und dessen um k/T_A bzw. $k \cdot f_A$ verschobenen Kopien bilden [19]. In Abbildung 3.2 wird deutlich, dass es bei einem zu groß gewählten Abtastintervall bzw. einer zu geringen Abtastfrequenz zu sichtbaren Überschneidungen der gespiegelten Spektren kommt. Dieses Phänomen wird als *Aliasing* bezeichnet und führt im Sinne des Shannon-Nyquist-Theorems zur Verfälschung des Spektrums und des tatsächlichen Signals.

3.1.3 Diskrete Fourier-Transformation

In realen Computersystemen ist die Berechnung der exakten DTFT aufgrund ihrer kontinuierlichen Spektralfunktion nicht möglich. Gleichzeitig würde sie eine nahezu unendliche Anzahl an Abtastwerten voraussetzen, was aufgrund der begrenzten Speicherkapazitäten von Computern nicht der Realität entspricht. Für eine endliche Sequenz $x[n]$ mit N Folgenglieder kann stattdessen die Diskrete Fourier-Transformation (DFT) herangezogen werden [20, S. 293], wobei im Zuge der im Abschnitt 3.1.6 beschriebenen häufig angewandten Implementierungen der DFT in digitalen Systemen im Folgenden geradzahlige N angenommen werden.

$$X[k] = \sum_{n=0}^{N-1} x[n] \mathrm{e}^{-j2\pi kn/N} \qquad mit \;\; k = 0, ..., N-1 \tag{3.8}$$

Die berechneten Spektralkomponenten $X[k]$ sind nicht nur N-periodisch, sondern entsprechen gleichzeitig den abgetasteten Werten der Spektralfunktion der DTFT an äquidistanten Stützstellen [21]. Die spektrale Auflösung ist von der Anzahl an Abtastpunkten N und der Abtastfrequenz f_A abhängig. Als Abstand auf der Frequenzachse zwischen diesen Spektrallinien, welche auch als *Frequency Bins* bezeichnet werden, ergibt sich das Verhältnis $\Delta f = f_A/N$ [22, S. 220 f.]. Die Spektralkomponente $X[0]$ bildet hierbei den Gleichanteil des Signals, während $X[N/2]$ die Komponente an der Nyquist-Frequenz widerspiegelt. Auf Basis der in Abschnitt 3.1.1 beschriebenen Symmetrieeigenschaften wird für reellwertige Sequenzen deutlich, dass die Frequenzkomponenten für $k = N/2 + 1, ..., N - 1$ den komplex konjugierten Varianten der Komponenten für $k = 1, ..., N/2 - 1$ entsprechen, sodass der Zusammenhang gilt [20, S. 336]:

$$X[k] = X^*(N - k) \tag{3.9}$$

In der praktischen Anwendung bedeutet dies, dass bei einer Frequenzanalyse eines Audiosignals mit einer festgelegten Anzahl N an Abtastwerten lediglich die Hälfte der berechneten Spektralkomponenten zum tatsächlichen Informationsgehalt im Frequenzbereich beitragen und die ab $k = N/2$ symmetrischen Komponenten als redundante Informationen wegfallen.

Nach Kammeyer & Kroschel führt die Rückführung des diskreten Spektrums in den Zeitbereich durch die Inverse Diskrete Fourier-Transformation (IDFT) zu einem ebenfalls in N periodischen Signal, weshalb die Werte der Spektralfolge der DFT als Koeffizienten der Fourier-Reihe des zugehörigen periodischen Zeitsignals zu interpretieren sind [22, S. 221 ff.]. Die IDFT ist gegeben durch die Formel [20, S. 293]

$$x[n] = \sum_{n=0}^{N-1} X[k] \mathrm{e}^{j2\pi kn/N} \qquad mit \;\; k = 0, ..., N - 1 \tag{3.10}$$

und unterscheidet sich zur DFT lediglich in einem Vorfaktor $1/N$ und dem Vorzeichen des Exponenten der komplexen Drehfaktoren (engl. Twiddle factor) mit $W_N = \mathrm{e}^{-j2\pi/N}$ [22, S. 222]. Dabei dient der eben genannte Vorfaktor der Umkehrung der Skalierung der Spektralkomponenten mit dem Faktor N durch die DFT [23]. Die Periodizität der komplexen Drehoperatoren und deren Symmetrieverhältnisse in der Gaußschen Zahlenebene stellen die Basis für Algorithmen zur effizienten Berechnung der DFT und IDFT dar.

3.1.4 Konsequenzen der Fensterung

Die DFT führt nur dann zu einem korrekten Linienspektrum, wenn bei einem periodischen Signal exakt eine oder mehrere vollständige Perioden analysiert werden. Dies bedeutet

auch, dass alle in dem Signal auftretenden Frequenzen aufgrund der in Abschnitt 3.1.3 beschriebenen Abstände Δf der Frequency Bins nur ein ganzzahliges Vielfaches dieser Abstände betragen dürften. In realistischen Anwendungen, wie bei der Analyse von Sprache oder Musik, kommen jedoch hauptsächlich aperiodische Signale unterschiedlicher Längen vor. Gleichzeitig sind Signale in der Praxis zeitlich begrenzt oder nur bestimmte Zeitabschnitte von Interesse. Mertins beschreibt den sog. Leck-Effekt (engl. Leakage effect) als eine Verfälschung der Spektralkomponenten $X(k)$, die durch die Abweichung der existierenden Frequenzen von jenem ganzzahligen Vielfachen von Δf in dem analysierten Zeitabschnitt entsteht [24, S. 187 f.]. Mathematisch lässt sich das Eingrenzen eines spezifischen Zeitabschnittes eines Signals, in der Literatur als Fensterung (engl. Windowing) bezeichnet, als Multiplikation des Signals mit einer Rechteckfunktion als Fensterfunktion beschreiben. Konsequenz dessen ist die Faltung der Spektralfunktion des Signals mit der Spektralfunktion des Rechteckfensters, der Sinc-Funktion, und führt zur Verzerrung des Spektrums [25, S. 216].

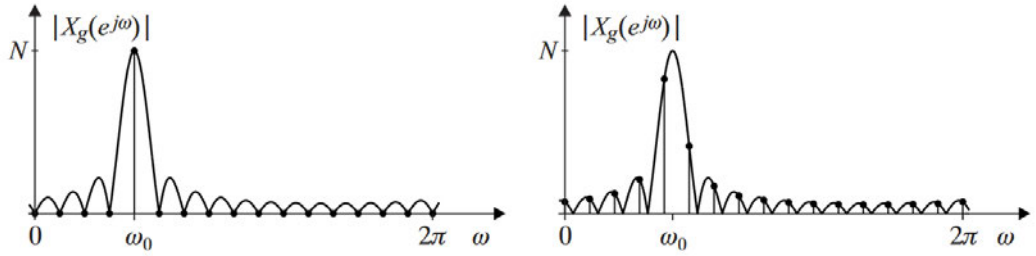


Abbildung 3.3: Darstellung des Leck-Effektes anhand der 16-Punkte-DFT komplexer Exponentialsignale mit $\omega_0 = 4 \cdot 2\pi/16$ (links) und $\omega_0 = (4 + 1/3) \cdot 2\pi/16$ (rechts) [24, S. 188]

Kammeyer & Kroschel [26, S. 276 ff.] beschreiben diesen Effekt anhand einer einfachen Exponentialfolge als Signal wie folgt: ist deren Frequenz als ein ganzzahliges Vielfaches des Frequenzinkrements f_A/N gegeben, wird die verschobene Sinc-Funktion als Hüllkurve des Spektrums einmal im Maximum und an den übrigen äquidistanten Nullstellen abgetastet; andernfalls liegen die Nulldurchgänge dieser Sinc-Funktion zwischen den diskreten Frequenzpunkten der DFT, sodass die Sinc-Funktion einerseits nicht mehr in ihrem Maximum abgetastet wird, es nun zwei Hauptlinien gibt und andererseits ebenfalls an den DFT-Rasterpunkten fehlerhafte Anteile auftreten.

Im Zeitbereich werden Signalanteile durch die Fensterung mit der Rechteckfunktion ggf. abgeschnitten. Dies führt meist zu Sprungstellen aufgrund der steilen Flanken des Rechteckfensters und da die DFT von einer periodischen Fortsetzung des gefensterten Signals ausgeht. Infolgedessen kommt es zum Leck-Effekt und auch Aliasing bei diskretisierten Signalen [18]. Zur Reduktion des Leck-Effektes werden in der Praxis bei der Frequenzanalyse und im Filter-Design typischerweise eine Reihe anderer spezifizierter Fensterfunktionen angewandt. Diese wirken der Entstehung von Sprungstellen an den beiden

Enden des Fensters entgegen. Ferner ist die Fensterung mit einer gewählten Fensterfunktion als eine Gewichtung der Abtastwerte mit den Funktionswerten des Fensters im Zeitbereich zu sehen [27]. Dadurch kommen Signalwerte an den Rändern des Fensters in der Berechnung der DFT weniger zur Geltung als Werte im Zentrum. Während der Leck-Effekt, auch zu betrachten als die Größe der Nebenkeulen (engl. Side Lobes) der Sinc-Funktion, dadurch deutlich reduziert wird, verbreitert sich dagegen die Hauptkeule (engl. Main Lobe) und die Frequenzselektivität sinkt. Der Zusammenhang ist in Abbildung 3.4 für häufig angewandte Fensterfunktionen dargestellt.

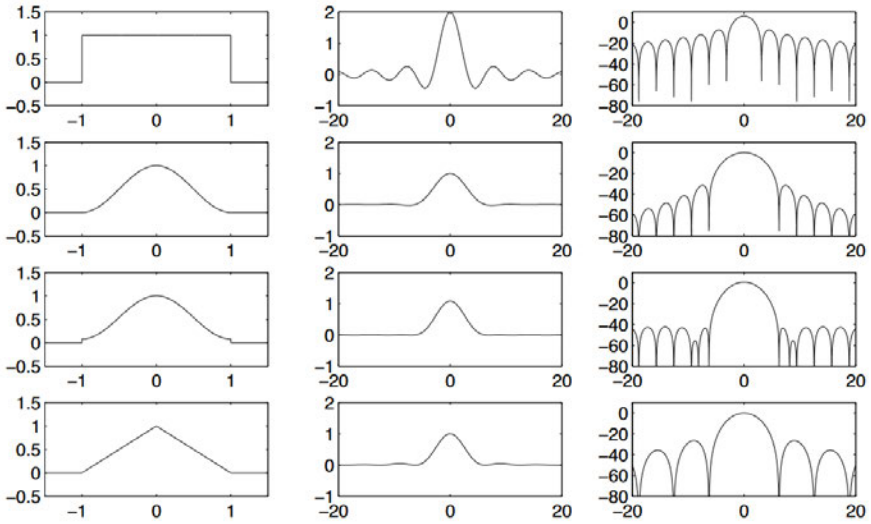


Abbildung 3.4: Ausgewählte Fensterfunktionen im Zeitbereich (links) und ihre Fourier-Transformierten linear (Mitte) und logarithmiert (rechts): Rechteck-, Hann-, Hamming- und Bartlett-Fenster (von oben nach unten) [18, S. 437]

Die Fourier-Transformierte der Rechteckfunktion weist zwar eine geringe Breite des Hauptmaximums und damit eine hohe Selektivität auf, jedoch ist die Höhe der Nebenmaxima in der logarithmierten Darstellung im Vergleich zu anderen Fensterfunktionen äußerst groß. Die sich nach [28] für verschiedene Fenster ergebenden Eigenschaften sind in Tabelle 3.1 aufgeschlüsselt.

Die Frequenzauflösung ist gleichzeitig abhängig von der gewählten Fenstergröße N und damit von der Länge des zu analysierenden Zeitabschnittes. Je größer dieser gewählt wird, desto höher ist auch die Auflösung im Spektralbereich, da die Abstände Δt zwischen den DFT-Rasterpunkten mit steigenden N kleiner werden.

3.1.5 Kurzzeit-Fourier-Transformation

Eine DFT liefert zwar Informationen über das mittlere Spektrum eines analysierten Zeitabschnittes, jedoch sind insbesondere akustische Signale häufig als instationär anzusehen und das tatsächliche Spektrum kann über die Zeit stark variieren [12, S. 6]. Die Kurzzeit-Fourier-Transformation (engl. Short-Time Fourier Transform (STFT)) dient der

Tabelle 3.1: Vergleich häufig angewandter Fensterfunktionen

Fenstertyp	Definition der Fensterfunktion	rel. Spitzenwert d. Side-Lobe-Amplitude [dB]	approx. Breite d. Main Lobe
Rechteck	$w[n] = \begin{cases} 1, & 0 \leq n \leq N \\ 0, & \text{sonst} \end{cases}$	-13	$4\pi/(N+1)$
Bartlett	$w[n] = \begin{cases} 2n/N, & 0 \leq n \leq N/2 \\ 2 - 2n/N, & N/2 < n \leq N \\ 0, & \text{sonst} \end{cases}$	-25	$8\pi/N$
Hanning	$w[n] = \begin{cases} 0.5 - 0.5\cos(2\pi n/N), & 0 \leq n \leq N \\ 0, & \text{sonst} \end{cases}$	-31	$8\pi/N$
Hamming	$w[n] = \begin{cases} 0.54 - 0.46\cos(2\pi n/N), & 0 \leq n \leq N \\ 0, & \text{sonst} \end{cases}$	-41	$8\pi/N$
Blackman	$w[n] = \begin{cases} 0.42 - 0.5\cos(2\pi n/N) + 0.08\cos(4\pi n/N), & 0 \leq n \leq N \\ 0, & \text{sonst} \end{cases}$	-57	$12\pi/N$

Darstellung dieser Veränderungen im Frequenzbereich über die Zeit. Hierfür wird eine Reihe von DFTs auf aufeinanderfolgende, kurze Zeitabschnitte des Signals mit einer vordefinierten Fensterlänge angewandt. Die Fensterfunktion wird dabei um die sog. Hop Size von H Samples über die Zeitachse verschoben und für jeden gefensterten Ausschnitt eine neue DFT berechnet [29, S. 196 f.].

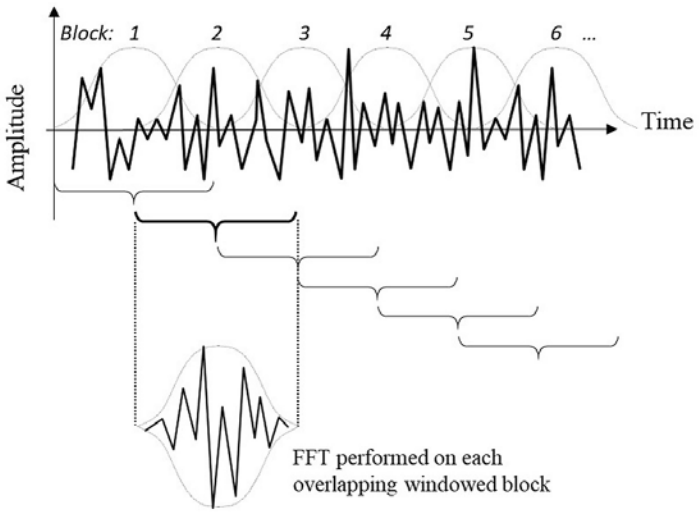


Abbildung 3.5: Darstellung des Konzepts der STFT [30, S. 4]

Je geringer die Hop Size gewählt wird, desto höher ist hierbei die zeitliche Auflösung [29]. Ein digitales System kann ein in Echtzeit eingehendes Signal auf diese Weise, z.B. bei der Aufnahme eines Mikrofonsignals in einem Studio, lediglich mit der gewählten Fenstergröße als gegebene Latenz in nahezu Echtzeit analysieren. Insbesondere für eine gewählte Hop Size, welche kleiner als die definierte Fenstergröße ist, bedeutet die

Berechnung in Echtzeit oft eine erhöhte Rechenlast für das operierende System, da so mehrere DFTs parallel berechnet werden müssen.

Die Beträge der einzelnen Spektren können in Form eines Spektrogramms visualisiert werden. Dabei ist der Zusammenhang zwischen der zeitlichen und spektralen Auflösung vergleichbar mit der in der Quantenphysik bekannten Heisenbergschen Unschärferelation [12]. Eine simultane, exakte Lokalisierung im Zeit- und Frequenzbereich ist demnach nicht möglich [31]. Während die Analyse kurzer Zeitabschnitte das Spektrum nur grob abbilden kann, ermöglichen längere Zeitfenster zwar eine hohe Auflösung im Frequenzbereich, lassen jedoch keine Aussage mehr über den genauen Zeitpunkt des Auftretens der Signalkomponenten zu [30].

3.1.6 Schnelle Fourier-Transformation

Die direkte Implementierung der DFT nach ihrer in Formel 3.8 gegebenen Definition in einem Computerprogramm ist in der Praxis mit erheblichen, technischen Schwierigkeiten verbunden. Während der Rechenaufwand in realen Computersystemen für eine große DFT-Längen stark ansteigt und damit die Berechnungsgeschwindigkeit enorm beeinträchtigt wird, nimmt gleichzeitig die Anfälligkeit für Rundungsfehler bei der Berechnung selbst unter Nutzung präziser Gleitkommaoperationen in modernen Prozessoren zu [32]. Die Fast Fourier Transform stellt eine Klasse an verschiedenen Algorithmen zur effizienten Berechnung der DFT dar, welche diverse mathematische Zusammenhänge und die Vorteile moderner Prozessorarchitekturen zu Gunsten der Einsparung an Rechenoperationen und Speicherzugriffen nutzen. Da das Ziel dieser Arbeit nicht die eigenständige Implementierung einer solchen FFT darstellt und in der Praxis meist auf höchst optimierte Softwarebibliotheken wie die kostenlose *Fastest Fourier Transform in the West* (FFTW) zurückgegriffen wird, stellt dieser Abschnitt lediglich einen kurzen Abriss zur Wirkweise solcher Algorithmen dar. Gleichzeitig soll damit die Wahl gewisser Parameter in Kapitel 4 nachvollziehbar gemacht werden.

Die Anzahl an Rechenoperationen einer N-Punkte-DFT ist proportional zu N^2 und die Komplexität wird meist mit $O(N^2)$ konnotiert [32, S. 18]. Die FFT reduziert diese Komplexität auf $O(N\,log(N))$ und ist damit wesentlich effizienter. Die am häufigsten in der Literatur erwähnten Ansätze stellen die durch Cooley und Tukey im Jahr 1965 bekannt gewordenen Algorithmen dar [20], [24], [32]. Diese zerlegen eine große N-Punkte-DFT in P kleinere DFTs der Größe Q [24]. Insbesondere, wenn N eine Potenz von 2 ist, lässt sich diese Teilung iterativ anwenden und es ergibt sich der Spezialfall der am häufigsten genutzten Radix-2-FFT [22]. Ist die Länge eines Datensatzes ungleich einer Potenz von 2, wird dieser einfach bis zur nächsthöheren validen Potenz von 2 mit Nullen aufgefüllt. Hierbei unterscheidet sich die Methodik zwischen der Reduktion im Zeitbereich (engl. Decimation in Time (DIT)) und der Reduktion im Frequenzbereich (engl. Decimation in

Frequency (DIF)) [20]. Nach [22, S. 235 ff.] lässt sich am Beispiel einer Radix-2-DIT-FFT eine DFT der Länge N in zwei Teilsummen mit je $N/2$ Summanden und jeweils geraden und ungeraden Indizes der Eingangsfolge $x[n]$ unterteilen:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} = \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{kn} + W_N^k \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{kn} \tag{3.11}$$

Dabei ist auch hier $W_N = \mathrm{e}^{-j2\pi/N}$ als komplexer Drehfaktor zu verstehen, dessen Symmetrieverhältnisse in der Gaußschen Zahlenebene bei der Teilung von geraden und ungeraden Indizes für die Bildung der Teilsummen ausgenutzt werden. Erneute Zerlegungen dieser Teilsummen führen zu der in Abbildung 3.6 dargestellten Struktur.

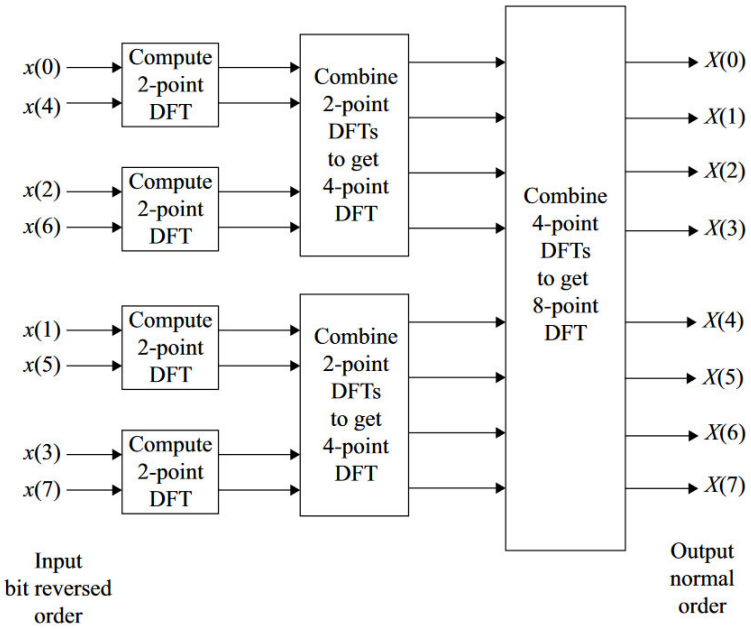


Abbildung 3.6: Block-Diagramm einer Radix-2-DIT-FFT der Länge 8 [20, S. 367]

Zu beachten ist, dass die stetige Unterscheidung der Indizes zwangsläufig zu einer Neuanordnung in der sog. Bit-umgekehrten (engl. bit-reversed) Reihenfolge der Eingangswerte führt, um die berechneten Spektralkomponenten in korrekter Anordnung zu erhalten [22].

Weitere, jedoch wesentlich komplexere Algorithmen mit unterschiedlichen Optimierungszielen umfassen u.a. höherwertige Radix-N-FFTs wie Radix-4-FFTs, Split-Radix-FFTs, der Winograd-Algorithmus oder auch Primfaktor-Algorithmen [22, S. 245 f.]. Manche Algorithmen nutzen auch die komplex konjugierten Symmetrieverhältnisse im Frequenzbereich für reellwertige Eingangssequenzen aus und erzielen somit eine weitere Reduktion der benötigten Rechenoperationen [33]. Professionelle Bibliotheken wie die FFTW nutzen außerdem Möglichkeiten zur zusätzlichen Geschwindigkeitssteigerung der FFT wie die vorausgehende Berechnung der Drehfaktoren und deren Speicherung in Lookup-

Tabellen, die Nutzung von SIMD-Operationen (Single Instruction, Multiple Data) und die Anpassung des Programmcodes an spezifische Prozessorarchitekturen.

3.2 Filterung von Signalen

Digitale Filter dienen in der digitalen Signalverarbeitung der gezielten Manipulation von Signalen. Dadurch können z.B. bestimmte Frequenzbereiche verstärkt oder unterdrückt werden oder es können bestimmte Frequenzen im sog. Durchlassbereich passieren, während andere Frequenzen im sog. Sperrbereich nahezu vollständig abgeblockt werden. In der Signal- und Systemtheorie werden Filter als Systeme betrachtet, die ein anliegendes Eingangssignal $x(t)$ zu einem Ausgangssignal $y(t)$ transformieren [10]. Meist wird in der Literatur der Fall linearer zeitinvarianter Systeme (LZI-Systeme) betrachtet. Diese erfüllen die Bedingungen der Linearität, welche mit dem Superpositionsgesetz einhergeht, und der Zeitinvarianz [34]. Demzufolge lässt sich die Systemantwort einer Linearkombination von Eingangssignalen ebenfalls anhand einer Linearkombination der jeweiligen, zugehörigen Ausgangssignale beschreiben. Gleichzeitig muss auf ein zeitlich verzögertes und kausales Eingangssignal eine ebenso verzögerte und kausale Systemantwort erfolgen. Im Folgenden soll sich auf jene LZI-Systeme bezogen werden und darauf aufbauend auf eine Möglichkeit der Implementierung zeitvariabler Filter mithilfe der STFT eingegangen werden. Aufgrund der vielfältigen Möglichkeiten der Konzeption und Implementierung von digitalen Filtern liegt der Fokus auf den in dieser Arbeit angewandten Zusammenhängen. Darüber hinaus bieten andere als die hier genannten Techniken Material für die weitere Forschung in diesem Gebiet.

3.2.1 Die Impulsantwort zur Charakterisierung eines Systems

Die Systemantwort eines LZI-Systems wird anhand der Anregung dessen mit einem Dirac-Impuls δ für zeitkontinuierliche Systeme bzw. dem Einheitsimpuls für zeitdiskrete Systeme charakterisiert. Diese ist auch als Impulsantwort bekannt und wird im Folgenden mit $h(t)$ beschrieben. Im Zeitbereich lässt sich zu einem beliebigen Eingangssignal $x(t)$ und einer definierten Impulsantwort $h(t)$ eines LZI-Systems das Ausgangssignal $y(t)$ eindeutig über das Faltungsintegral bestimmen [10, S. 54 f.]:

$$y(t) = x(t) * h(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau = \int_{-\infty}^{\infty} h(\tau)x(t-\tau)d\tau \tag{3.12}$$

Entspricht das Eingangssignal dem erwähnten Dirac-Impuls, gilt so der Zusammenhang $y(t) = \delta(t) * h(t) = h(t)$. Für zeitdiskrete Signale und Systeme entspricht das Faltungsintegral stattdessen der Faltungssumme. Auch hier sind die Spiegelung und die zeitliche Verschiebung unter der Summe für $x[n]$ oder äquivalent $h[n]$ aufgrund der kommutativen

Eigenschaft der Faltungsoperation frei vertauschbar [35]:

$$y[n] = x[n] * h[n] = \sum_{m=-\infty}^{\infty} x[m]h[n-m] \tag{3.13}$$

Über die Anwendung der FT lässt sich das Faltungstheorem beweisen [15, S. 19 f.]. Folglich entspricht die Faltungsoperation im Zeitbereich einer Multiplikation der beiden Fourier-Transformierten der zu faltenden Funktionen im Frequenzbereich. Es gilt der Zusammenhang:

$$Y(\omega) = X(\omega) \cdot H(\omega) \tag{3.14}$$

Umgekehrt bedeutet eine Multiplikation zweier Funktionen im Zeitbereich ebenfalls eine Faltung im Frequenzbereich. Dies verdeutlicht auch den in Abschnitt 3.1.4 erläuterten Effekt der Fensterung. Die Auflösung nach $H(\omega)$ in Formel 3.14 bildet die frequenzabhängige Übertragungsfunktion des LZI-Systems, welche sich als der komplexe Quotient aus den Spektren von Ausgangssignal und Eingangssignal versteht [36, S. 4]. Wird die komplexwertige Übertragungsfunktion $H(\omega)$ stattdessen anhand ihres Betragsfrequenzgangs $|H(\omega)|$ und ihres Phasengangs $\varphi_H(\omega)$ abgebildet, ergeben sich für Betrags- und Phasenspektrum des Ausgangssignals nach [10, S. 87 f.]:

$$|Y(\omega)| = |X(\omega)| \cdot |H(\omega)|\,, \;\; \varphi_Y(\omega) = \varphi_H(\omega) + \varphi_X(\omega) \tag{3.15}$$

Das Betragsspektrum $|H(\omega)|$ ist demnach maßgeblich für die frequenzabhängige Verstärkung bzw. Abschwächung der Spektralkomponenten eines Eingangssignals, während das Phasenspektrum $\varphi_H(\omega)$ deren frequenzabhängigen Phasenverschiebungen beschreibt. Eine in der Literatur häufig beschriebene Kenngröße ist zudem neben der Phasenlaufzeit τ_p, welche die frequenzabhängigen Verzögerungen der einzelnen zugrundeliegenden Sinusschwingungen angibt, die sog. Gruppenlaufzeit τ_g. Diese ist als

$$\tau_g = -\frac{d\varphi_H(\omega)}{d\omega} \tag{3.16}$$

gegeben und definiert die Verzögerung schmalbandiger Frequenzgruppen am Systemausgang [10, S. 88]. Dabei werden LZI-Systeme mit einer konstanten Gruppenlaufzeit auch als linearphasige Systeme bezeichnet, da das Ausgangssignal gegenüber dem Eingangssignal in diesem Fall keine Phasenverzerrungen aufweist.

3.2.2 Kausalität als Einflussfaktor realer Systeme

Kausale Filtersysteme sind insbesondere für Anwendungen in Echtzeit von Interesse, da bekanntlich eine Wirkung, in diesem Fall das Ausgangssignal $y(t)$, nicht vor ihrer Ursache, hier das Eingangssignal $x(t)$, auftreten kann. Entsprechend muss das zugehörige

LZI-System ebenfalls kausal sein. Ein System ist dabei als kausal einzustufen, wenn dessen Impulsantwort $h(t) = 0$ für alle $t < 0$ ist. Das ideale Tiefpassfilter wird in der Literatur meist als Beispiel nicht-realisierbarer Filter angeführt, da die inverse Fourier-Transformierte der Rechteckfunktion im Frequenzbereich als Sinc-Funktion im Zeitbereich einerseits von unendlicher Länge und andererseits nicht kausal ist [37, S. 190]. Derartige Filter können nur approximiert werden, wenn die Impulsantwort mittels Fensterung als zeitbegrenzt und durch eine eingeführte Verzögerung bzw. Gruppenlaufzeit als kausal erscheint.

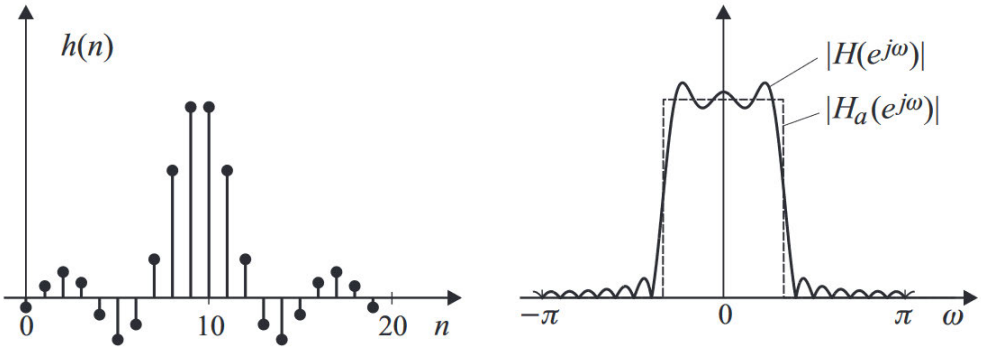


Abbildung 3.7: gefensterte und auf der Zeitachse verschobene Impulsantwort zur Approximation eines idealen Tiefpasses [16, S. 162]

Dieser Vorgang führt nicht nur zu einem linearen Phasengang $\varphi_H(\omega)$, denn durch die Zeitbegrenzung mittels Rechteckfenster kommt es gleichzeitig zum merklichen Überschwingen (engl. Ripple) in der Region der Grenzfrequenz des Tiefpassfilters ω_g im Zuge des Gibbs'schen Phänomens [16]. Dem Leck-Effekt ähnlich kann dieses Überschwingen durch die Wahl einer geeigneten Fensterfunktion deutlich reduziert werden. Im Gegenzug sinkt allerdings auch die Flankensteilheit zwischen Durchlass- und Sperrbereich. Ein Filter mit unendlich steilen Flanken ist somit in der Praxis nicht realisierbar.

3.2.3 Klassifizierung von Filtern nach der Impulslänge

Eine Klassifizierung digitaler bzw. zeitdiskreter Filter kann nach der Betrachtung der Impulslänge eines LZI-Systems erfolgen. Dabei wird nach [28, S. 245 f.] für das Verhältnis von Ein- und Ausgang die Darstellung als Differenzengleichung zu Rate gezogen.

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k] \tag{3.17}$$

Demnach werden Filter mit einer unendlichen Impulsantwort auch als Infinite Impulse Response Filter (IIR-Filter) bezeichnet und zeichnen sich durch den in Formel 3.17 gegebenen rekursiven Anteil aus [38]. Dies bedeutet, dass die Ausgabewerte $y[n]$ auch von früheren Ausgabewerten $y[n-k]$ abhängen und die Impulsantwort $h[n]$ daher nicht

vollständig auf den exakten Wert Null ausklingt. Im Gegensatz dazu werden Finite Impulse Response Filter (FIR-Filter) durch eine endliche Impulsantwort und einer oft nichtrekursiven Filterstruktur beschrieben, wodurch sich die Impulsantwort des Systems exakt anhand der Koeffizienten b_k bestimmen lässt [39, S. 102]. Die Zahl M gibt hierbei die Länge der Impulsantwort im Zeitbereich an. Üblicherweise erfolgt die Darstellung eines LZI-Systems in Form eines Blockdiagramms. Verschiedene mögliche Formen der Filterstruktur geben Aufschluss über die Implementierung im Programmcode, sollen allerdings hier nicht weiter ausgeführt werden.

3.2.4 Übertragungsfunktionen von Filtern

Die Übertragungsfunktion $H(z)$ für zeitdiskrete LZI-Systeme wird unter Nutzung der z-Transformation berechnet. Die z-Transformierte eines zeitdiskreten Signals $x[n]$ ist nach [16, S. 138] definiert als:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]\, z^{-n} \tag{3.18}$$

Der komplexwertige Variable z wird nach [40, S. 55 f.] als $z = r \cdot \mathrm{e}^{j\Omega}$ notiert, wobei die z-Transformation für $r = 1$ beim Vergleich der Formeln 3.6 und 3.18 in die Definition der DTFT übergeht. In Bezug auf die Darstellung eines LZI-Systems als Differenzengleichung ergibt sich für die Übertragungsfunktion [28, S. 246]:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum\limits_{k=0}^{M} b_k z^{-k}}{\sum\limits_{k=0}^{N} a_k z^{-k}} \tag{3.19}$$

Die größere der beiden Zahlen M und N gibt die Ordnung des Systems bzw. die Filterordnung an [40, S. 68]. Alle z_0, für welche der Zähler Null wird, werden als Nullstellen bezeichnet, wohingegen alle möglichen Nullstellen des Nenners z_p Polstellen genannt werden. Die Platzierung der Pol- und Nullstellen weisen nicht nur auf das prinzipielle Aussehen des Betrags $|H(z)|$ der Übertragungsfunktion und damit verbunden den Filtertyp wie z.B. Tiefpass-, Hochpass-, oder Bandpassfilter hin. Auch geben sie Aufschluss über die Stabilität des LZI-Systems. Die BIBO-Stabilität (Bounded Input Bounded Output) ist nur unter der Bedingung gegeben, dass die Impulsantwort eines Systems absolut summierbar ist [41, S. 175].

$$\sum_{n=-\infty}^{\infty} |h[n]| < \infty \tag{3.20}$$

Dies bedeutet, dass die Ausgangsfolge $y[n]$ bei einem stabilen System und einer beschränkten Eingangsfolge $x[n]$ nicht ins Unendliche wachsen kann. Für $H(z)$ besagt das außerdem, dass die Darstellung in der z-Ebene für die Bedingung der Stabilität den Ein-

heitskreis für $|z| = 1$ als Konvergenzbereich mit einschließt [41, S. 175][40, S. 73]. Alle Polstellen außerhalb dieses Einheitskreises führen dadurch zur Instabilität des Systems. Ein Vorteil von FIR-Filtern liegt in der Tatsache, dass diese aufgrund ihrer nicht-rekursiven Struktur stets stabil sind und sich deshalb alle Polstellen im Ursprung der z-Ebene befinden.

3.2.5 Entwurf digitaler Filter

Die Ansätze zum Design von FIR-Filtern und IIR-Filtern sind vielfältig und versuchen, die Impulsantwort diverser, teils idealer Filterprototypen im Zeit- und Frequenzbereich bestmöglich zu approximieren. Die Unterschiede liegen insbesondere in der Genauigkeit der Approximationen und der Effizienz der Implementierung. Außerdem sind nicht alle Filtertypen wie z.B. Hilbert-Transformatoren, welche einen speziellen und eher selten angewandten Filtertyp darstellen, mit allen Methoden umsetzbar.

Design-Ansatz für FIR-Filter

Eine bewährte Methode zum Entwurf von FIR-Filtern stellt die Fourier-Approximation und die damit verbundene Fenstertechnik dar. Der Ansatz wurde bereits in Abschnitt 3.2.2 angedeutet und wird in Abbildung 3.8 deutlich. Ausgangspunkt bildet eine gewünschte Übertragungsfunktion $H_a(\mathrm{e}^{j\omega})$. Kammeyer & Kroschel beschreiben den Grundgedanken dieser Methode als die „Minimierung des Integrals des quadrierten Approximationsfehlers über das gesamte Frequenzband“[38, S. 178]. Demnach ergibt sich die Impulsantwort des resultierenden Filters als die inverse DTFT der idealisierten Übertragungsfunktion [16]. Die Eingrenzung der meist unendlich langen Impulsantwort unter Verwendung des Rechteckfenster im Zeitbereich führt zu starken Oszillationen im Durchlass- und Sperrbereich der tatsächlichen Übertragungsfunktion $H(\mathrm{e}^{j\omega})$. Die Höhe der ersten Überschwinger im Bereich der Filterflanke beträgt dabei etwa 9% [38, S. 180]. Je größer der Ausschnitt der Impulsantwort im Zeitbereich und damit die Filterordnung gewählt wird, desto genauer wird die Approximation und dementsprechend können umso steilere Filterflanken realisiert werden. Die Fenstertechnik ersetzt das Rechteckfenster durch eine andere geeignete Fensterfunktion, um die Oszillationen in $H(\mathrm{e}^{j\omega})$ zu verringern. Nachteil ist die dadurch sinkende Steilheit der Filterflanken. Der mathematische Zusammenhang der Fenstermethode kann nach [16, S. 161] als die Faltung der Impulsantwort mit der Fensterfunktion im Frequenzbereich dargestellt werden.

$$H(\mathrm{e}^{j\omega}) = \frac{1}{2\pi} H_a(\mathrm{e}^{j\omega}) * W(\mathrm{e}^{j\omega}) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H_a(\mathrm{e}^{j\theta})\, W(\mathrm{e}^{j(\omega-\theta)}) d\theta \tag{3.21}$$

Wird eine kontinuierliche, ideale Übertragungsfunktion im Frequenzbereich stattdessen an äquidistanten DFT-Rasterpunkten abgetastet und mithilfe der IDFT die Impulsantwort berechnet, ist in der Literatur stattdessen von der Frequency-Sampling-Methode die Re-

de, die sich aus der Beziehung der DFT zur z-Transformation ergibt [42, S. 675].

Für alle genannten Methoden muss die Kausalität des Filters gewährleistet werden, indem eine entsprechende Verzögerung bzw. Gruppenlaufzeit τ_g eingeführt wird. Daraus ergibt sich ein weiterer Vorteil von FIR-Filtern, denn durch die Wahl einer konstanten Gruppenlaufzeit können linearphasige Filter realisiert werden, welche die Phaseninformationen eines Signals nicht beeinflussen. Für linearphasige Filter beträgt τ_g die Hälfte der gewählten Filterordnung [38].

Da, wie in [42, S. 621 f.] ausgeführt wird, insbesondere für reellwertige linearphasige FIR-Filter gewisse Symmetrieverhältnisse für die Impulsantwort im Zeitbereich gelten und sich das Symmetriezentrum in Höhe der Gruppenlaufzeit befindet, ist durch den Faltungsvorgang ein Einschwingverhalten des Systems erkennbar. In der praktischen Anwendung in üblichen Audio-Applikationen wird dies oft als Reaktionszeit des Filters betrachtet und kann je nach Ausprägung des Effekts zur hörbaren Reduktion der Wahrnehmbarkeit von Transienten führen. Wird z.B. der regelmäßige Anschlag einer Trommel vereinfacht und modellhaft als Dirac-Stoßfolge angenommen, entspricht das Ausgangssignal der Impulsfolge, gefaltet mit der Impulsantwort des FIR-Filters. Gleichzeitig erscheinen die Impulse am Ausgang um τ_g verzögert, was bei Echtzeitanwendungen berücksichtigt werden muss.

3.2.6 Effiziente Implementierung von FIR-Filtern

Der Faltungsprozess stellt besonders für lange Impulsantworten einen äußerst rechenaufwändigen Prozess dar. Unter Berücksichtigung des Faltungstheorems und der Nutzung von FFT-basierten Transformationen in den Frequenzbereich sowie die anschließende Resynthese des Ausgangsspektrums in den Zeitbereich kann die Anzahl an nötigen Rechenschritten stark reduziert werden [43]. Dieser Prozess wird als schnelle Faltung bezeichnet und bildet die Grundlage für die effiziente Implementierung von FIR-Filtern mit hoher Filterordnung.

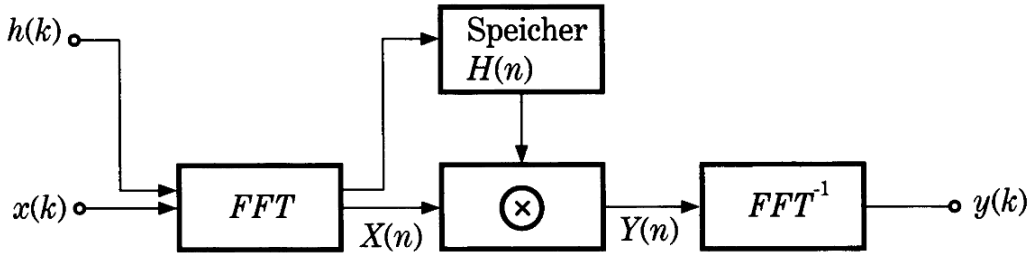


Abbildung 3.8: Blockschema zur Realisierung eines FIR-Filters mithilfe der schnellen Faltung [22, S. 249]

Es muss bei digitalen Filtern allerdings die Periodizität der DFT berücksichtigt werden. Nach [22] ist zwischen der zyklischen und der aperiodischen Faltung zu differenzieren. Die Multiplikation $X(k) \cdot H(k)$ entspricht dabei der zyklischen Faltung im Frequenzbereich. Die aperiodische oder auch lineare Faltung zweier Sequenzen, gegeben in Formel 3.13, unterscheidet sich klar von der zyklischen Faltung mit [24, S. 185]:

$$x_1[n] \circledast x_2[n] = \sum_{p=0}^{N-1} x_1[n]\, x_2[(n-p) \bmod N] \tag{3.22}$$

Die aperiodische Faltung zweier Sequenzen der Längen M und N würde $M + N - 1$ Abtastwerte umfassen [22, S. 227]. Damit das Ergebnis dem der zyklischen Faltung für einen Zeitabschnitt entspricht, werden die Sequenzen $x_1[n]$ und $x_2[n]$ in der Praxis auf die Länge $M+N-1$ mit Nullen aufgefüllt und danach die DFTs der verlängerten Sequenzen berechnet. Bei Anwendung der FFT muss die nächsthöhere Zweierpotenz als Ziellänge gewählt werden. Dies wird in der Literatur oft als *Zero Padding* referenziert und bildet die Basis für blockbasierte Filtermethoden wie die Overlap-Save-Methode oder die Overlap-Add-Methode [20], [24].

Beispielhaft beschreibt Mertins die Overlap-Add-Technik wie folgt [24, S. 199]: es sei eine Eingangssequenz und eine zweite Sequenz wie die Impulsantwort eines Systems mit einer kürzeren Länge M gegeben. Die Eingangssequenz wird in Blöcke der Länge $N - M+1$ zerlegt und mit Nullen auf die DFT-Länge N aufgefüllt. Die $M-1$ inkorrekten Werte am Ende eines jeden Blocks und am Anfang des jeweils nächsten Blocks werden addiert und führen stattdessen zum korrekten Ergebnis. Die Fortsetzung dieses Prozesses über die einzelnen Blöcke führt zu einem mit der linearen Faltung identischen Ergebnis.

3.2.7 Zeitvariable Filter unter Anwendung der STFT

Ist die korrekte Resynthese der einzelnen FFT-Frames bei der STFT von Interesse, müssen sich alle um $mH, m \in \mathbb{Z}$ verschobenen Varianten der gewählten Fensterfunktion, wobei H die gewählte Hop Size darstellt, in Summe zu Eins addieren [44].

$$\sum_{m=-\infty}^{\infty} w[n - mH] = 1,\ \forall n \in \mathbb{Z} \tag{3.23}$$

Dieses Prinzip wird auch als COLA-Bedingung (Constant Overlap-Add) angeführt [45]. Demnach entspricht die Summe aller aufeinanderfolgenden DTFTs der einzelnen Zeitabschnitte der DTFT des gesamten Signals.

Während [44, S. 325 f.] zeigt, dass sich die STFT zur Implementierung der schnellen Faltung ähnlich dem Overlap-Add-Verfahren und damit für zeitinvariante Filter eignet, führt [45] an, dass die STFT auch zur Implementierung zeitvarianter Filtersysteme angewandt werden kann. Nach [46] kann die STFT aus zwei unterschiedlichen Perspektiven betrach-

tet werden. Beide Darstellungsweisen gehen von der Definition der STFT als

$$X_n[\mathrm{e}^{j\omega_k}] = \sum_{m=-\infty}^{\infty} w[n-m]x[m]\mathrm{e}^{-j\omega_k m} \tag{3.24}$$

aus [46], wobei bei Vergleich mit Gleichung 3.23 die Hop Size hier als $H = 1$ definiert ist. Nach den in [46, S. 1559] gezeigten Umstellungen dieser Gleichung betrachtet die Filter-Bank-Summation-Methode (FBS-Methode) die STFT als Filterbank mit uniformen Abständen der Subfilter und summiert bei der Rekonstruktion des transformierten Signals die einzelnen Kanäle nach zusätzlicher Modulation und Filterung auf. Die Overlap-Add-Methode (OLA-Methode) hingegen basiert auf der in Abschnitt 3.1.5 erläuterten Darstellung, indem mit der Fensterfunktion gewichtete Signalabschnitte analysiert und bei Erfüllung der COLA-Bedingung eine korrekte Rekonstruktion stets gegeben ist. Bei dem in der Praxis oft gegebenen Fall von $H > 1$ kann die Fensterung bei der Resynthese als Überblendung der aufeinanderfolgenden Frames bzw. LZI-Systeme betrachtet werden, wohingegen dies bei der FBS-Methode als Unterabtastung zu verstehen ist und deshalb zusätzliche Maßnahmen zur Verminderung von Aliasing-Artefakten ergriffen werden müssen [45]. [47] erörtert zudem die Auswirkungen spektraler Modifikationen. Diese sind als Multiplikation im Frequenzbereich und damit als Filterung zu verstehen. Zur Vermeidung von Aliasing im Zeitbereich durch die zyklische Faltung muss jedoch auch hier das Zero Padding erfolgen, sodass die Anzahl der Nullen jener Länge der Impulsantwort der spektralen Modifikationen im Zeitbereich entspricht [47, S. 237]. Wenn dies nicht gegeben ist, sind die spektralen Modifikationen als nicht-linearer Prozess zu verstehen. Dies ist z.B. der Fall, wenn die spektralen Modifikationen als Funktion der Analyseergebnisse der STFT gegeben sind und über die Zeit variieren [46, S. 1564]. Portnoff [48] beschreibt die Anwendung dieser Technik anhand des sog. Phase-Vocoders bei der Analyse, Modifikation und Resynthese von Sprache. Eine Methode zur Reduktion des Aliasing-Effekts im Zeitbereich durch die zyklische Faltung ist die erneute Fensterung des resynthetisierten Zeitsignals mit einer geeigneten Fensterfunktion [49]. Diese modifizierte Variante der OLA-Methode wird auch als Weighted-Overlap-Add-Methode (WOLA) bezeichnet, wobei die Fensterfunktion nach Anwendung der IDFT auch als Synthesefenster bekannt ist. Die von Crochiere [50] entwickelte WOLA-Prozedur in Abbildung 3.9 weist zudem Faktoren $(-1)^k$ zur zyklischen Rotation der Samples um $M/2$ auf. Dies führt zur Einführung einer linearphasigen Komponente, wobei sie im Fall, dass die Länge des Eingangsblocks R der Länge des Ausgangsblocks R' entspricht, weggelassen werden kann. Voraussetzung sei jedoch, dass die Reihenfolge von Phasen- und Spektralmodifikation vertauschbar ist [50, S. 101].

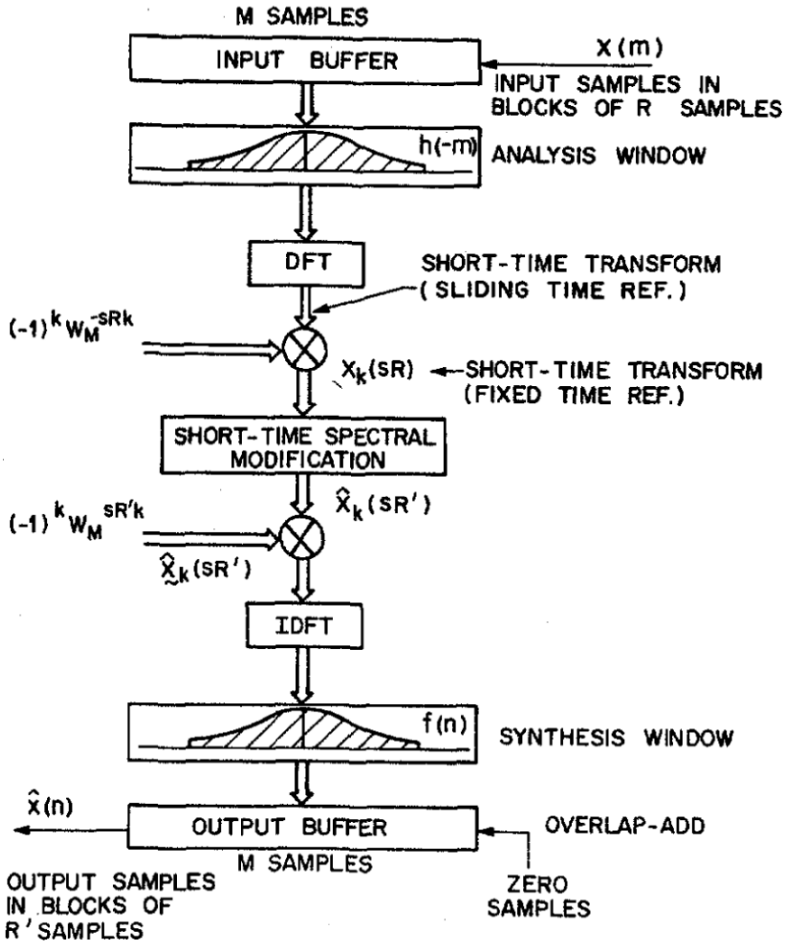


Abbildung 3.9: Weighted-Overlap-Add-Methode mit linearer Phasenmodifikation im Frequenzbereich [50, S. 101]

3.3 Menschliches Hörempfinden

Nach der Entwicklung des Plugins soll der Effekt anhand von Hörbeispielen im Hinblick auf ausgewählte Eigenschaften des menschlichen Hörvermögens, genauer wahrgenommene Lautheit und Schärfe, untersucht werden. Diese dienen der Arbeit als Indikatorgrößen für die tatsächlich als *störend* wahrgenommenen Frequenzen in einem Signal. Aus dem Fachbereich der Psychoakustik lassen sich mehrere Parameter ableiten, anhand derer die menschliche Wahrnehmung zu einem Schallereignis eingeschätzt werden kann. Dies sind z.B. die Lautheit, Schärfe, Rauigkeit, Schwankungsstärke oder Tonhaftigkeit. Anzumerken ist, dass sich diese Größen in der Literatur häufig auf vereinfachte Signale wie breitbandigem Rauschen oder Einzeltöne beziehen [51],[52],[53]. Zudem werden viele der Kenngrößen mit der empfundenen Lästigkeit beim Lärmschutz in Verbindung gebracht. In der Audiopraxis spielt jedoch neben einem subjektiv als angenehm empfundenen Hörerlebnis auch die Intention des Tonschaffenden bei der Klanggestaltung insbesondere in verschiedenen Musik-Genres eine Rolle. In diesem Zusammenhang können Effekte wie Vibrato, Tremolo oder Verzerrungen, die einem Klang wahrnehmbare Mo-

dulationen und Artefakte beifügen, als stilistisches Mittel zur ästhetischen Klanggestaltung genutzt werden [5]. Neben dem gezielten Gehörtraining beeinflussen auch äußere, technische Faktoren wie der Frequenzgang des verwendeten Lautsprechermodells, die Abhörposition und die Raumakustik den letztlich wahrgenommenen Klang oder das Heraushören von Details.

3.3.1 Kurven gleicher Lautstärke

Die menschliche Wahrnehmung der Lautstärke ist stark von den in einem Schallereignis vorkommenden Frequenzen und deren Schalldruckpegel abhängig. So beschreibt [51, S. 253], dass der nötige Schalldruckpegel zur tatsächlichen Wahrnehmung eines Tons an der Hörschwelle als untere Wahrnehmungsgrenze in den Übergängen in den Infra- und Ultraschallbereich stark zunimmt und das Gehör bei einer Frequenz von etwa 4000 Hz am empfindlichsten ist. Der hörbare Frequenzbereich des Menschen liegt hierbei je nach Quelle zwischen 16 Hz und 16 kHz [51] bzw. 20 Hz und 20 kHz [54]. Die sog. *Isophonenkurven*, auch als Kurven gleicher Lautstärke bezeichnet, in Abbildung 3.10 stellen diesen Zusammenhang grafisch dar.

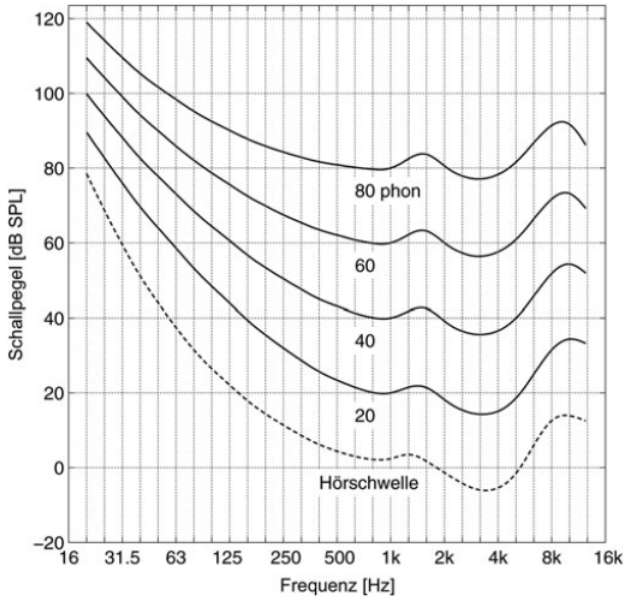


Abbildung 3.10: Kurven gleicher Lautstärke [54, S. 3]

Diese sind in der ISO 226:2023-03 [55] festgehalten. Der Lautstärkepegel in der Einheit Phon gibt an, wie hoch der Schalldruckpegel eines reinen Tons mit einer Frequenz von 1 kHz ist, welcher als gleichlaut wie der zu bewertende Ton an anderer Frequenz wahrgenommen wird [51, S. 253]. Demnach orientieren sich die Lautstärkewerte in Phon in Abbildung 3.10 am Schalldruckpegel bei 1 kHz. Erkennbar ist zudem, dass der wahrgenommene Lautstärkeunterschied zwischen dieser Referenzfrequenz und sehr tiefen Frequenzen mit steigender Lautstärke abnimmt, da die Kurven nach oben hin flacher

werden.

3.3.2 Standardisierte Frequenzbewertungsfilter

In der Praxis wurden zur gehörrichtigen Lautstärkemessung verschiedene Frequenzbewertungsfilter eingeführt. In diesem Abschnitt soll eine Übersicht an Frequenzbewertungsfunktionen gegeben werden, welche in das Audio-Plugin einprogrammiert wurden und je nach Aufgabenstellung vom Nutzer situativ angewandt werden können.

Die häufig angewandte A-Bewertung orientiert sich an der 40-Phon-Kurve, d.h. an eher geringen Lautstärken [56]. Im Vergleich dazu ist die C-Bewertung an hohen Lautstärken von 80 bis 90 Phon ausgerichtet [51, S. 254]. In verschiedenen Anwendungsfeldern wie der Beurteilung von Industrie- und Fluglärm sind zudem weitere Bewertungen wie die Z-, B- oder D-Bewertung zu finden. Die Z-Bewertung entspricht dabei einer Nullbewertung, d.h. es findet keine Korrektur der gemessenen Amplituden im Frequenzbereich statt. Die oft angewandten A- und C-Bewertungen sind als Filter in IEC 61672-1:2013-09 [57] normiert. Zeller [51, S. 254] merkt an, dass die Frequenzbewertungen bei komplexeren Geräuschen lediglich eine grobe Näherung der tatsächlich empfundenen Lautstärke ermöglichen können. Nach Schick [53] ist die oftmals gängige Anwendung der A-Bewertung in vielen Bereichen umstritten, da der starke Abfall der Kurve in den tiefen Frequenzen oft zu einer Unterbewertung dieser Frequenzen führt und sich die Anwendung eher auf sehr schmalbandige Geräusche geringer Lautstärke bezieht.

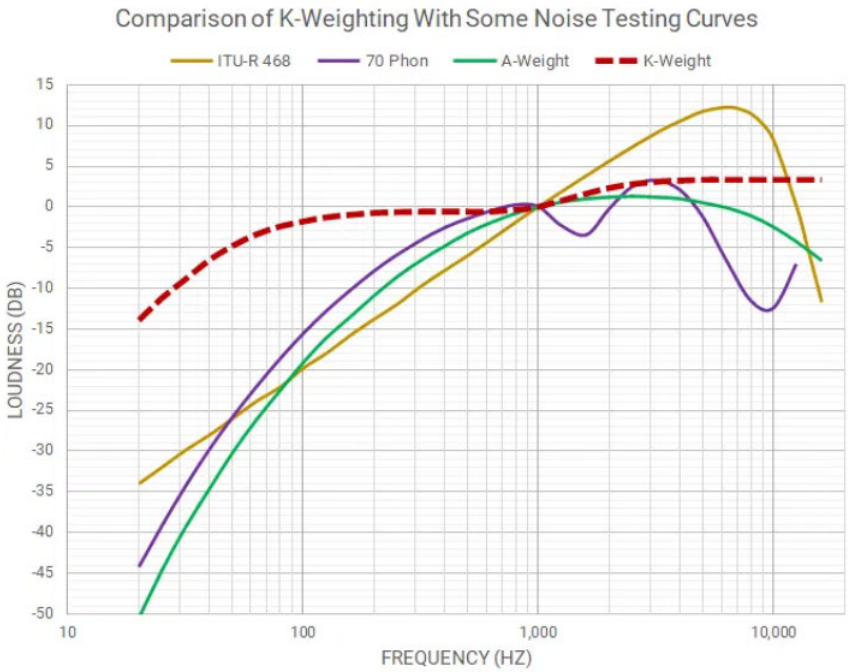


Abbildung 3.11: Verschiedene Frequenzbewertungskurven im Vergleich: nach ITU-R 468 (gelb), invertierte 70-Phon-Kurve (lila), A-Bewertung (grün) und K-Bewertung (rot gestrichelt) [58]

In der Messung von breitbandigem Rauschen in elektrischen Schaltungen wie Mikrofonverstärkern kommt zudem die Frequenzbewertung nach der Norm ITU-R BS.468-4 [59] in Kombination mit einem Quasispitzendetektor für die Lautheitsmessung zum Einsatz. Wie in Abbildung 3.11 ersichtlich ist, erfährt insbesondere der Bereich oberhalb von 1 kHz eine deutliche Anhebung, da Rauschen in dieser Region schnell als störend empfunden werden kann. Eine bei 2 kHz normalisierte Variante dieser Kurve wird dagegen bei der Bewertung von Kino-Soundtracks zur Erfassung von als scharf bezeichneten Geräuschen wie Explosionen oder Schussgeräuschen angewandt [58]. Voraussetzung ist das Abhören in einem entsprechend eingemessenen Raum. Diese Version ist in ISO 21727:2016-05 [60] normiert und wird auch als M-Bewertung bezeichnet.

Von besonderem Interesse im Rundfunk, im Fernsehen und in der Produktion von Musik sind die Empfehlungen nach der EBU R 128-2023-11 [61]. Diese definieren die Messung der Lautheit, der Dynamik und der exakten Spitzenwerte in digitalen Produktionsumgebungen. Das Ziel dieser Empfehlung stellt eine Vereinheitlichung der Programmlautheit zum Vorteil des Konsumenten dar, um hohe Unterschiede des Lautheitseindrucks von bspw. wechselnden Fernsehprogrammen zu vermeiden. Zur Messung der Lautheit, hier angegeben in *Loudness Units relative to Full Scale* (LUFS) bzw. dessen relativer Wert *Loudness Units* (LU), kommt demnach ein nach ITU-R BS.1770-5 [62] standardisiertes Messverfahren unter Nutzung der sog. K-Bewertung zum Einsatz. Das zweistufige Filter sorgt für die Absenkung tiefer Frequenzen unterhalb von 100 Hz und für eine Betonung der Frequenzen oberhalb von 2 kHz.

3.3.3 Anwendungsbezogene akustische Parameter

Lautheit und Spezifische Lautheit

Die Lautheit wird in der Akustik mit der tatsächlich wahrgenommenen Lautstärke in Verbindung gebracht [54]. Als Bezugspunkt wird der 1-kHz-Ton bei einem Schalldruckpegel von 40 dB für 1 sone als Einheit der Lautheit angenommen [51, S. 256]. So verdoppelt sich die Lautheit für einen Sinuston von 1 kHz bei einer Erhöhung des Schalldruckpegels um jeweils 10 dB. Dabei ist eine Umrechnung von der Lautheit und dem Lautstärkepegel über die Formel

$$\frac{L_N}{Phon} = \begin{cases} 40 + 33.22 \cdot \lg(\frac{N}{sone}) & \text{für } \ N \geq 1\,sone \\ 40 \cdot (\frac{N}{sone} + 0.0005)^{0.35} & \text{für } \ N < 1\,sone \end{cases} \tag{3.25}$$

gegeben [51, S. 256] und wird in der DIN 45631/A1:2010-03 [63] näher erläutert. Dabei ist die Wahrnehmung dieser Kenngröße jedoch auch von weiteren Faktoren wie der Frequenz und der Bandbreite, [54, S. 6] der Dauer des Schallereignisses [64] sowie zeitlichen und frequenzbezogenen Verdeckungseffekten abhängig. Nach Zwicker [52] teilt das menschliche Gehör Schalle bei der Lautheitswahrnehmung aufgrund seines frequenzse-

lektiven Charakters in 25 Bänder, sog. Frequenzgruppen, definiert als Tonheit z von 0 bis 24 Bark ein. Eine Aufschlüsselung dieser Frequenzgruppen in Hz ist in [65] zu finden. Diese weisen unterhalb von 500 Hz eine relativ konstante Bandbreite, darüber jedoch etwa die Breite einer Terz auf [51, S. 256]. Die sog. spezifische Lautheit N' hingegen stellt die Verteilung der Lautheit über die Tonheit dar und wird in sone/Bark angegeben [52]. Die Gesamtlautheit kann durch Integration der spezifischen Lautheit über die Tonheit errechnet werden.

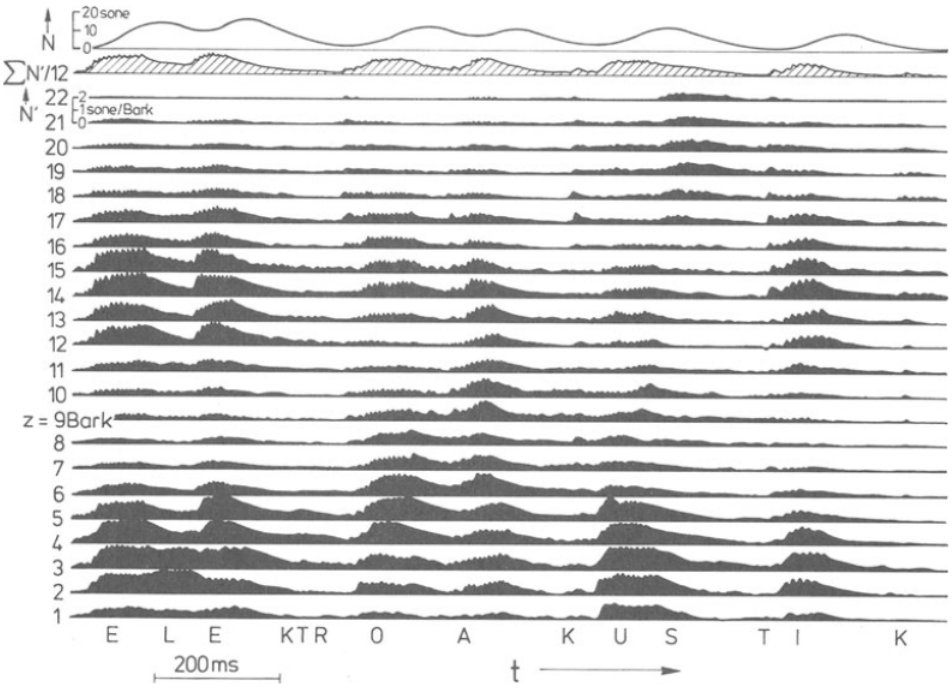


Abbildung 3.12: Spezifische Lautheiten N', ihre Summe $\sum N'$ (durch 12 geteilt) und ihre Gesamtlautheit N sind für das gesprochene Wort *Elektroakustik* jeweils als Funktion der Zeit t aufgetragen [66, S. 137]. Sichtbar ist eine unterschiedliche Verteilung bei Vokalen und Zischlauten.

Da das zu entwickelnde Audio-Plugin in Tonstudioapplikationen für Musik und Sprache angewandt wird, ist eine Berücksichtigung der instationären Eigenschaft solcher Signale nötig. Die in DIN 45631/A1:2010-03 [63] gezeigte Methode stellt neben der Moore-Glasberg-Methode [67] eine geeignete Annäherung zur Bestimmung der Lautheit über die Zeit dar.

Schärfe

Die Schärfe als ausgewählter Parameter hängt eng mit der Wahrnehmung der Klangfarbe eines Schallereignisses zusammen und gilt als Indikator der Balance tiefer und hoher Frequenzen in einem Signal [51, S. 259]. Es kann hierbei die Verallgemeinerung getroffen werden, dass ein Schallereignis als umso schärfer wahrgenommen wird, je mehr hohe Frequenzanteile darin vorkommen. Sie wird in der Einheit *acum* angegeben. Zwicker [68,

S. 84 f.] führt an, dass die Schärfe als Empfindungsgröße vergleichbar ist und sie bei der Betrachtung von Schmalbandrauschen erheblich mit steigender Mittenfrequenz wächst. Dagegen ist die Schärfe für breitbandige Signale abhängig von der unteren bzw. oberen

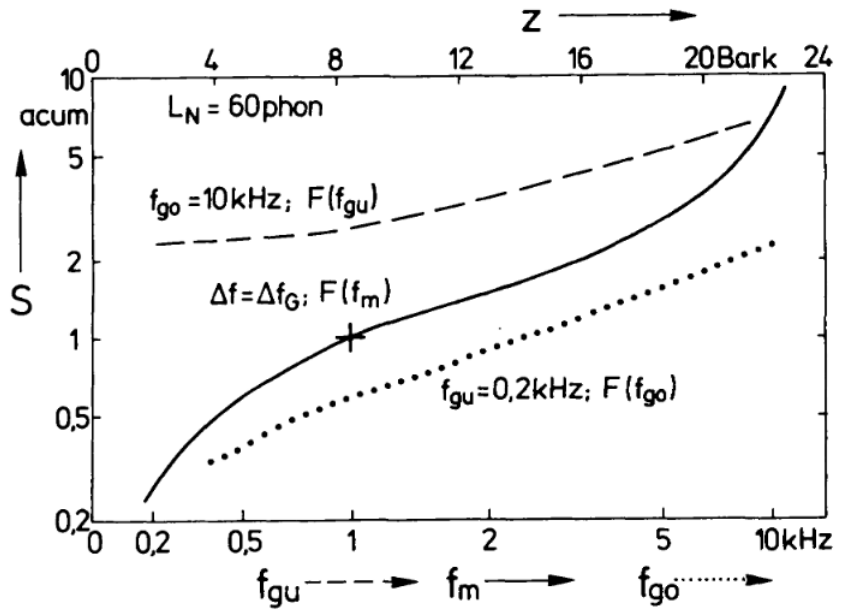


Abbildung 3.13: Schärfe von Schmalbandrauschen (durchgezogen), Tiefpassrauschen (punktiert) und von Hochpassrauschen (gestrichelt) als Funktion der Mittenfrequenz f_m bzw. der oberen Grenzfrequenz f_{go} bzw. der unteren Grenzfrequenz f_{gu} [68, S. 84]

Bandbegrenzung. Das in dieser Arbeit verwendete Berechnungsverfahren basiert auf der DIN 45692:2009-08 [69].

4 Entwurf und Implementierung des Audio-Plugins in Max/MSP

In den folgenden Abschnitten wird die Implementierung des Plugins in der Entwicklungsumgebung Max/MSP beschrieben. Die im Wesentlichen zur Anwendung kommenden Algorithmen für die Signalverarbeitung werden anhand von Programmcode-Ausschnitten genauer beschrieben. Für das Verständnis der erläuterten Implementierung in Max/MSP ist außerdem ein grundlegendes Verständis der Funktionsweise dieser Entwicklungsumgebung nötig. Gleichzeitig werden damit technische Grenzen, die bei der Implementierung im weiteren Verlauf deutlich wurden, aufgedeckt.

4.1 Die Entwicklungsumgebung Max/MSP

Max/MSP, oft auch nur als Max bezeichnet, ist eine interaktive, grafische Entwicklungsumgebung des Softwareunternehmens Cycling ’74 und wurde ursprünglich am Institut de recherche et coordination acoustique/musique (IRCAM) entwickelt [70]. Besonders beliebt ist sie aufgrund der vielfältigen Einbindungs- und Verarbeitungsmöglichkeiten von Audio- und Videoinhalten in Echtzeit bei Künstlern und Entwicklern im audiovisuellen Bereich. Programme werden anhand der logischen Verknüpfung von Objekten und Signalpfaden in einem visuellen Format erstellt. Die Programmierweise ist damit der datenstromorientierten Programmierung zuzuordnen. Die Kompilierung der Programme erfolgt in Max mit wenigen Ausnahmen nahezu in Echtzeit. Dies bedeutet, dass Änderungen an Algorithmen für das Audio- und Video-Processing bei laufendem Programm zu einem sofortigen hörbaren bzw. sichtbaren Ergebnis führen. Eventuelle Fehlermeldungen werden direkt über die integrierte *Max Console* ausgegeben. Dadurch eignet sich Max vor allem zur schnellen Konzeption und Visualisierung von Multimedia-Applikationen. Auch findet die Entwicklungsumgebung dadurch Anwendung bei audiovisuellen Live-Performances. Die folgenden Erläuterungen über die Funktionsweise von Max/MSP beruhen auf Informationen aus der Max Documentation [71] und eigenen Nutzungserfahrungen des Verfassers. Die in dieser Arbeit verwendete Version von Max ist Version 8.5.8. Auf die neueste Version von Max, Version 9, wurde dagegen verzichtet, da sie zum Zeitpunkt dieser Arbeit noch keine Kompatibilität zu der genutzten Version 11 von Ableton Live aufweist und zuverlässige Tests des Audio-Plugins unter diesen Umständen nicht möglich waren. Zur Visualisierung der Arbeitsweise in Max/MSP dienen dem Leser die Abbildungen in Anhang A.

4.1.1 Grundlegende Arbeitsweise in Max/MSP

Eine Quellcode-Datei wird in Max als *Patcher* bezeichnet und bildet die Grundlage eines Projekts. Das Hinzufügen einer Funktion erfolgt über das Erstellen von *Objekten*. Dies

kann über die Auswahl eines Objekts am Rand des Programmfensters oder über entsprechende Tastenkürzel geschehen. Die standardmäßig verfügbaren Objekte umfassen u.a. mathematische Operatoren, Befehle zur Verarbeitung von Listen, Elemente für die Erstellung von GUIs oder auch die Nutzung von proprietären Funktionsweisen in Max. Ein definiertes Objekt besitzt eine bestimmte Anzahl an *Inlets* (Quellen), die sich an der Oberseite des Objekts befindlichen Verbindungspunkte; sowie *Outlets* (Senken) an dessen Unterseite. Während Inlets hauptsächlich der Übergabe von Eingabewerten an das Objekt dienen, geben Outlets die durch das Objekt verarbeiteten Rückgabewerte aus. Die Kommunikation der Objekte untereinander erfolgt mithilfe von *Patch Cords*, welche die Inlets und Outlets der Objekte durch virtuelle Kabel miteinander verbinden. So können Daten von Objekt zu Objekt weitergegeben und verarbeitet werden.

Objekten können bei der Erstellung Argumente und Attribute zugewiesen werden. Gültige Werte sind für jedes Objekt in der Max Documentation [71] ausführlich beschrieben. Gleichzeitig sind für jedes Objekt Hilfsdateien mit direkt ausführbaren Beispiel-Patches abrufbar. Meist beziehen sich die übergebenen Argumente auf den initialen Status eines Objekts und können von außen durch die Übergabe von Parametern an den Inlets modifiziert werden. Das *Inspector*-Fenster in Max bietet Zugriff auf das visuelle Erscheinungsbild eines Objekts und stellt Informationen und Einstellungsmöglichkeiten für das Objekt bereit.

4.1.2 Erstellung komplexer Programmstrukturen

Ein Patcher in Max kann mehrere *Subpatcher* beinhalten und ermöglicht so den Aufbau hierarchischer Programmstrukturen. Sie werden direkt in der Datei des Parent Patchers, dem in der Hierarchieebene nächsthöheren Patcher, eingebettet [72]. Wird ein Patcher hingegen in Form einer *.maxpat*-Datei auf dem Computer abgespeichert, ist die Wiederverwendung in Form sog. *Abstractions* möglich. Bei der Erstellung eines *patcher*-Objekts kann der Dateiname, mit dem die Abstraction in den Suchpfad von Max aufgenommen wurde, angegeben werden. Dadurch wird die gespeicherte *.maxpat*-Datei in das Projekt eingebunden. Im Gegensatz zum Subpatcher ergibt sich dadurch die Tatsache, dass jede Änderung des Inhalts einer Abstraction gleichzeitig zur Änderung aller Instanzen in allen Max-Projekten führt, welche diese Abstraction einbinden [72].

Die Kommunikation einer Abstraction oder eines Subpatchers mit dem Parent Patcher kann auf zwei Arten erfolgen. Über das Erstellen von *inlet*- und *outlet*-Objekte innerhalb eines Subpatchers oder einer Abstraction können dem darauf zugreifenden *patcher*-Objekt neue Inlets und Outlets hinzugefügt werden. Die Verbindungen von Objekten im Parent Patcher mit den hinzugefügten Inlets und Outlets führen zur Kommunikation mit den Objekten im Patcher, welche darin mit den *inlet*- und *outlet*-Objekten verbunden sind. Wenn Daten über viele Hierarchieebenen hinweg transportiert werden müssen, birgt die-

se Methode den Nachteil, dass entsprechend viele Inlets und Outlets für jeden Subpatcher in jeder Ebene erstellt werden müssen. Abhilfe schaffen hierfür *send*- und *receive*-Objekte. Ein *send*-Objekt mit einem zugewiesenem Namen sendet dabei die an seinem Inlet eingehenden Daten an *receive*-Objekte mit gleichem Namen in allen geladenen Patcher-Dateien.

4.1.3 Signalverarbeitung in Max/MSP

Max unterscheidet zwischen verschiedenen Arten von Signalen. Dazu gehören neben Audiosignalen die in Max proprietären *Messages* sowie Bilddaten in Form von Matrizen und Texturen. Für jeden Signaltyp existieren verschiedene Objekte, die auf die Verarbeitung dieser Datenströme spezialisiert sind. Auch sind sie durch die unterschiedliche Färbung der Patchkabel zwischen den Objekten visuell voneinander zu trennen.

Messages

Messages bilden das Rückgrat eines Patches in Max. Dadurch können u.a. die Eigenschaften von Objekten dynamisch geändert, Funktionen über spezielle *Bang*-Messages ausgelöst und komplexe Programmabläufe wie GUI-Steuerungen realisiert werden. Sie umfassen verschiedene Datentypen wie Integer, Float, Symbole und Listen und können durch Objekte modifiziert werden. Messages werden nicht kontinuierlich über die Patchkabel zwischen den Objekten ausgetauscht. Stattdessen müssen sie durch einen Trigger, dem sog. *Bang*, ausgelöst werden. Auch sind in diesem Zusammenhang nicht alle Inlets und Outlets eines Objekts gleichwertig. Meist führt das Empfangen einer Message oder eines Bangs am linkesten Inlet eines Objekts erst zur Ausgabe des Ergebnisses. Die beispielhafte Übergabe einer Zahl am rechten Inlet eines mathematischen Objekts wie *+* oder - speichert lediglich das berechnete Ergebnis in dem Objekt. Zur Ausgabe kommt es erst bei Aktualisierung des Wertes am linken Inlet oder beim dortigen Empfang eines Bangs. Für ein präzises Timing des Sendens von Messages ist die Aktivierung der Einstellung *Overdrive* im *Options*-Menü von Max notwendig. So werden Messages im Prozess hoher Priorität verarbeitet, während Aufgaben wie die Darstellung des GUI auf dem Bildschirm im Prozess niedriger Priorität untergeordnet werden [73].

Audiosignale und die Gen-Umgebung

Der Begriff MSP steht für *Max Signal Processing* und bildet die zentrale Engine für Audio in Max. Objekte zur Synthese und Verarbeitung von Audiosignalen sind in Max mit einer Tilde (~) gekennzeichnet. Anders als Messages durchlaufen die Audiodaten das Programm bei aktiviertem DSP kontinuierlich. Jede vorgenommene Änderung am Signalpfad ist somit direkt hörbar. Max bietet eine Auswahl an Objekten zur Synthese, Manipulation, Analyse und Speicherung von Audiosignalen. Dazu gehören u.a. Signalgeneratoren,

verschiedene Filter, Buffer-Objekte zur Speicherung, Oszilloskope und Spektrumanalyzatoren zum Signalmonitoring sowie Implementierungen der FFT und diverse Rechenoperationen.

Max berechnet die Signalwerte aus Effizienzgründen in Form von Signalvektoren. Die standardmäßig eingestellte Vektorlänge beträgt 64 Samples. Die direkte Implementierung von Schleifen zur Rückkopplung (Feedback) einzelner Samples ist deshalb nicht möglich und führt zur Ausgabe von Fehlermeldungen. Eine hierfür existierende Lösung sind die Objekte *Gen~* und *Gen*. Zur Programmierung von Audio-Anwendungen kommt speziell das *Gen~*-Objekt zum Einsatz, da es in der Signal Processing Routine der Software verarbeitet wird. *Gen* hingegen funktioniert dagegen, ähnlich wie Messages, im Event Processing Thread. Mit der Erstellung eines solchen Objekts wird ein Gen-Subpatcher angelegt. Alle in Gen ausgeführten Operationen erfolgen *Sample-by-Sample*. Das Abspeichern von Signalwerten für die Wiederverwendung in späteren Zyklen, z.B. für die Programmierung von Verzögerungsgliedern, ist nur unter Nutzung spezieller *History*-, *Buffer*- oder *Data*-Objekte möglich.

History-Objekte speichern nur einzelne Werte für nachfolgende Durchläufe. *Buffer* und *Data* können hingegen längere Signale in mehreren Kanälen aufnehmen. Ein benanntes *Buffer*-Objekt in Gen referenziert im Gegensatz zu *Data* ein außerhalb der Gen-Umgebung definiertes, gleichnamiges *Buffer~*-Objekt. So können gespeicherte Signalwerte zwischen klassischen Max-Patchern und Gen-Patcher ausgetauscht werden. Das *codebox*-Objekt dient in der Gen-Umgebung als Alternative zur Arbeitsweise mit Objekten, indem Code in Form von Quelltext verfasst werden kann. Die hierfür genutzte Skriptsprache *GenExpr* ähnelt in der Syntax Programmiersprachen wie C und wird von den *gen*-Objekten in nativen Maschinencode übersetzt [74]. Ein Vorteil von *codebox*-Objekten ist außerdem die Möglichkeit der Nutzung von *for*- und *while*-Schleifen, was durch die Verknüpfung von Objekten hingegen nicht möglich ist. Die kombinierte Nutzung von *codebox* und anderen Objekten in Gen ist möglich. Der sukzessive Aufbau eines Gen-Patches generiert automatisch den effektiv kompilierten GenExpr-Code und kann im *Code*-Tab eingesehen werden.

4.1.4 Nutzung gängiger Programmiersprachen in Max

Max bietet umfassende Möglichkeiten zur Erweiterung des Funktionsumfangs und zur Einbindung von Drittanbieter-Software. So stehen zusätzliche Objekte für bestimmte Anwendungsfelder im *Package Manager* zum Download zur Verfügung. Eigene Objekte können unter Anwendung des Max Software Development Kit in der Sprache C programmiert werden. Derartige Objekte werden in Max als *Externals* bezeichnet. Daneben können Objekte zur Interaktion mit Messages mit dem *js*-Objekt oder komplexe GUIs mit dem *jsui*-Objekt unter Anwendung der unterstützten JavaScript-Funktionen entwickelt werden.

Die in dieser Arbeit genutzte Version 8 von Max unterstützt das ECMAScript 5 von JavaScript [75]. Weiterhin bietet Max auch Programmiersprachen wie HTML, Java oder Lua eine Grundlage für spezifische Anwendungsfälle.

4.1.5 Max for Live

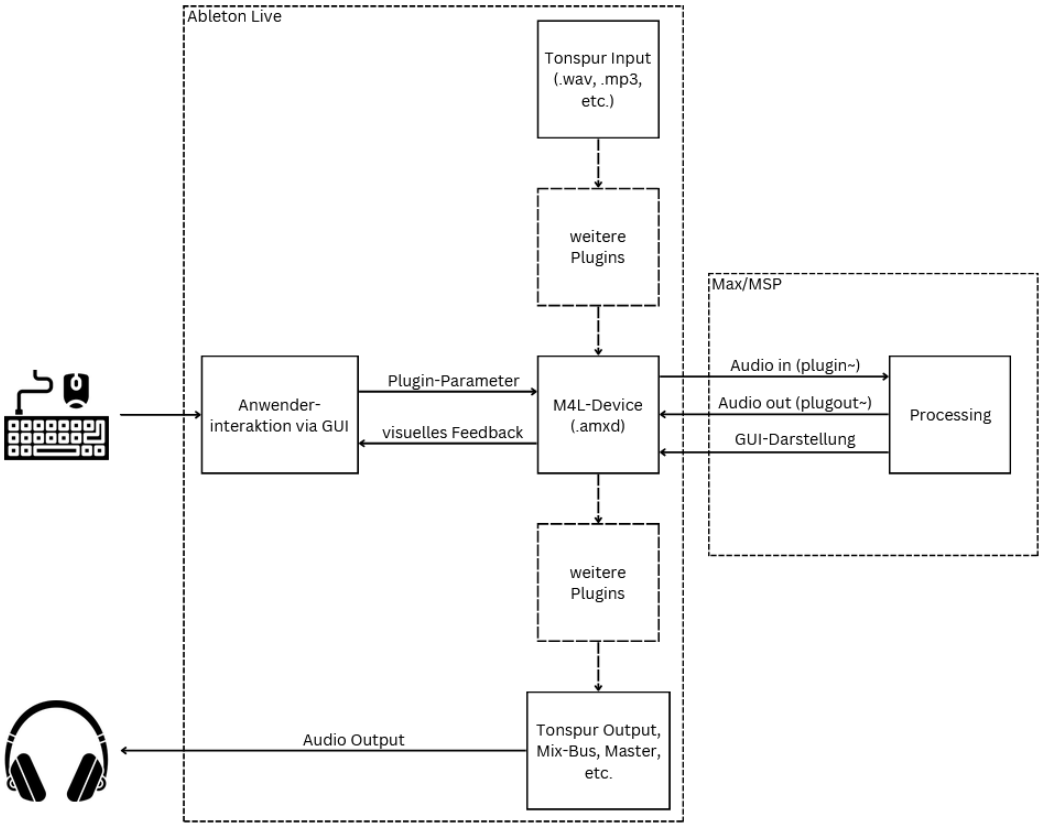


Abbildung 4.1: Integrationsprinzip von M4L-Plugins in Ableton Live (Eigene Darstellung)

Max for Live, oft mit dem Kürzel M4L gekennzeichnet, beschreibt die nativ integrierte Variante von Max in die DAW Ableton Live. Voraussetzung ist die Nutzung der Standard- oder Suite-Version von Ableton Live. Mit Max erstellte Plugins werden in Ableton Live als *Max for Live Device* oder kurz *Device* synonymisiert. Abbildung 4.1 zeigt, wie ein solches Device in die DAW integriert wird. Die Nutzerinteraktion findet über die Oberfläche von Ableton Live statt. Audio wird bei Einbindung eines solchen Plugins in eine Signalkette für die Signalverarbeitung in Max umgeleitet, wo auch das GUI für das Device berechnet wird. Abgespeichert wird das Device als *.amxd*-Datei auf dem Computer und kann mittels Drag-and-Drop in beliebige Tonspuren in Ableton Live geladen werden. Voraussetzung ist das ordnungsgemäße Speichern aller genutzten Abstractions und Skriptdateien in eine angelegte, individuelle Ordnerstruktur. Dies verhindert ungewollte Komplikationen bei späterer Abänderung einer Abstraction oder eines Skripts in Verbindung mit anderen Projekten.

Max for Live bietet Entwicklern Objekte zur nahtlosen visuellen Integration in die DAW

und zur Kommunikation mit programminternen Funktionen von Ableton Live über das *Live Application Programming Interface* (API). Solche Objekte sind mit dem Präfix *live.* gekennzeichnet.

4.2 Implementierung

Das Audio-Plugin wurde unter Nutzung der in Abschnitt 4.1 erläuterten Grundlagen und unter Maßgabe der als Zielstellung formulierten Kriterien entwickelt. Nach der Orientierung an den in Abschnitt 2 erläuterten Design-Prinzipien wurden zudem Funktionen wie die Option des Mid-Side-Processings oder die Wahl eines externen Sidechain-Eingangs implementiert. Für die visuelle und funktionale Integration in Ableton Live kamen neben klassischen Max-Objekten und Gen-Patchern auch *live.*-Objekte und eine Reihe an JavaScript-Dateien zur Implementierung des GUI zum Einsatz. Die folgenden Abschnitte beschreiben und begründen die Implementierung, jedoch kann in Hinsicht auf den Umfang des Quellcodes nur auf die wesentlichen Aspekte eingegangen werden.

4.2.1 Signalfluss

Die Verarbeitungskette wurde auf Basis eines erstellten Programmablaufplans implementiert. Dieser orientiert sich auch an der technischen Realisierbarkeit in Max/MSP. Wie sich im Verlauf der Programmierung herausstellte, ist die Implementierung der aperiodischen Faltung über die schnelle Faltung unter der Maßgabe zeitvarianter Filter in Max problematisch. Der im Folgenden als Filterung bezeichnete Verarbeitungsprozess ist deshalb als spektrale Modifikation gemäß der WOLA-Methode und damit als nicht-linearer Vorgang zu werten.

Abbildung 4.2 zeigt den Programmablaufplan. Der Signalfluss ist von oben nach unten zu lesen. Die schwarz dargestellten Signalwege beziehen sich jeweils auf die Anwendung auf ein Stereopaar. Orange dargestelle Blöcke und Pfeile charakterisieren hierbei auf dem GUI anwählbare Parameter. Das Prinzip beläuft sich auf die Anwendung der WOLA-Methode mit Analyseprozess, individueller Bewertung der Frequency Bins, spektraler Modifikation und anschließender Synthese inklusive Overlap-Adding und Fensterung. Bei Betrachtung fällt die Aufteilung in zwei grundlegende Signalpfade auf. Der linke Signalpfad beinhaltet das unbewertete Signal, auf welches die Anwendung der Filter erfolgen soll. Zur Einhaltung der COLA-Bedingung erfolgt die Fensterung mittels Hanning-Fenster. Diese Entscheidung belief sich aufgrund der Tatsache, dass diese Bedingung nicht für alle beliebigen Fensterfunktionen gültig und eine korrekte Rekonstruktion damit nicht immer gegeben ist. Der rechte Signalpfad hingegen beinhaltet die *Analyse* des Signals mithilfe eines vom Anwender ausgewählten Fensters. Die COLA-Bedingung ist somit bei der Resynthese stets erfüllt, auch wenn keine spektralen Modifikationen stattfin-

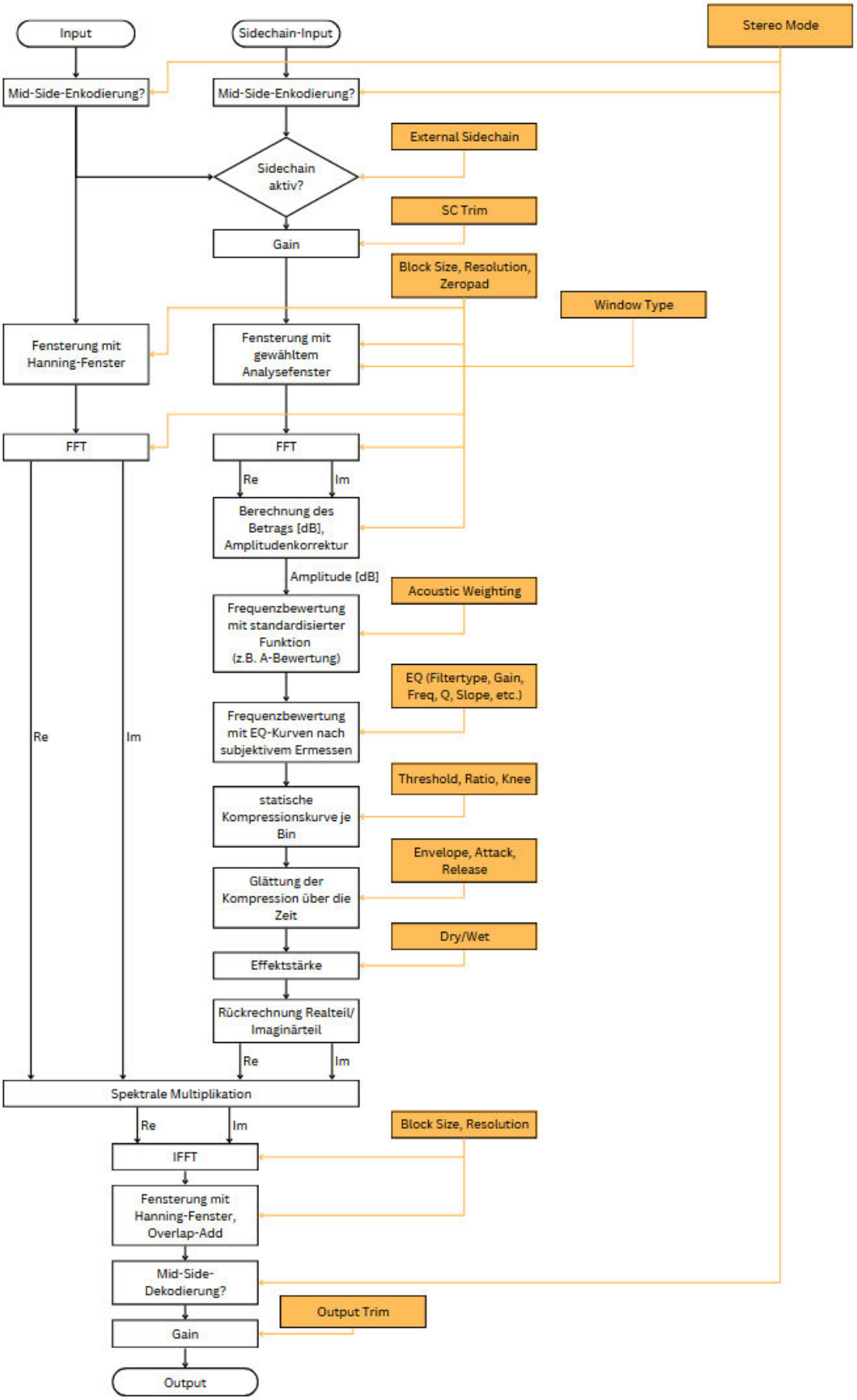


Abbildung 4.2: Programmablaufplan des entwickelten Audio-Plugins (Eigene Darstellung)

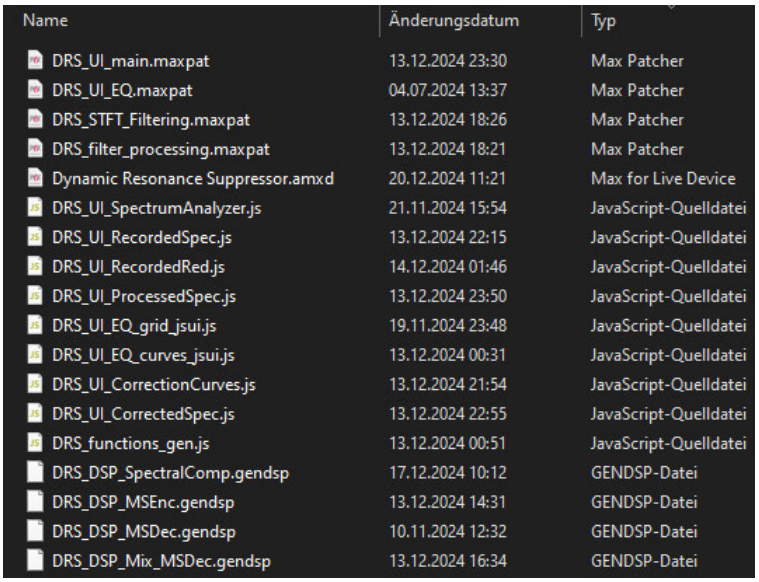
den. Dem Anwender wird durch die Integration des Sidechain-Eingangs die Möglichkeit der Einspeisung eines externen Regelsignals gegeben.

Nach der Transformation des Analysesignals in den Frequenzbereich erfolgen die Amplitudenberechnung und Amplitudenkompensation aufgrund der Amplitudenverzerrung durch die Fensterung [76]. Danach werden die Amplituden nach der eingestellten Frequenzbewertungsfunktion und den EQ-Kurven angepasst. Auf Basis dessen erfolgt die Berechnung der Reduktion der Frequenzamplituden anhand der Parameter *Threshold*, *Ratio* und *Knee*. Im Anschluss wird der berechnete Verstärkungsfaktor durch den Smoothing-Algorithmus mit *Attack* und *Release*-Parametern über die Zeit geglättet und die Intensität des Gesamteffekts über den *Dry/Wet*-Regler eingestellt. Das im Analysepfad generierte, effektive Filterprofil wird dann mit den Spektraldaten des zu filternden Signals durch Multiplikation im Frequenzbereich verrechnet. Das Ergebnis wird zum Schluss über die Inverse FFT resynthetisiert, erneut gefenstert und durch das Overlapping mit vorherigen Signalabschnitten addiert.

4.2.2 Vorbereitende Maßnahmen

Abbildung 4.3 zeigt den Inhalt des eingerichteten Projektordners. Dieser weist neben der Device-Datei als Main-Patcher des Projekts auch eine Reihe an Abstractions, JavaScript-Dateien und Gen-Dateien auf. Das separate Speichern der Gen-Patcher als GENDSP-Dateien erwies sich im Laufe der Programmierung als notwendig, da es aufgrund eines unbekannten Fehlers in Max gelegentlich zum Datenverlust bei eingebetteten Gen-Patchern kommen kann. Nicht alle der gezeigten Dateien wurden zudem im Endprodukt eingebunden. So beinhaltet die Datei *DRS_UI_EQ.maxpat* lediglich einen Patch zum Konzipieren und Testen der EQ-Oberfläche. Das separate Testen einzelner Programmabschnitte in ausgelagerten Umgebungen spielte im Prototyping-Prozess eine wichtige Rolle, um im Fall von Fehlermeldungen die Zahl anderer möglicher Fehlerquellen auf ein Minimum zu reduzieren.

Eine weitere im Vorfeld zu treffende Einstellung umfasste das Aktivieren der Option *Open in Presentation* im Inspector-Fenster des Main-Patches. Max unterscheidet beim Programmieren eines Patches zwischen dem *Presentation Mode* und dem *Patching Mode*. Ersterer ist für das Anordnen der Objekte in einem GUI konzipiert, wodurch visuell störende Elemente wie Patchkabel unsichtbar werden. Während der Programmierung kann über einen Button am Fensterrand zwischen den beiden Modi gewechselt werden. Erst das Einschalten der benannten Option führt zur korrekten Darstellung des Plugin-GUI in Ableton Live. Alle Elemente, die in dem GUI des M4L-Plugins dargestellt werden sollten, müssen im Main-Patcher sichtbar sein.



Name	Änderungsdatum	Typ
DRS_UI_main.maxpat	13.12.2024 23:30	Max Patcher
DRS_UI_EQ.maxpat	04.07.2024 13:37	Max Patcher
DRS_STFT_Filtering.maxpat	13.12.2024 18:26	Max Patcher
DRS_filter_processing.maxpat	13.12.2024 18:21	Max Patcher
Dynamic Resonance Suppressor.amxd	20.12.2024 11:21	Max for Live Device
DRS_UI_SpectrumAnalyzer.js	21.11.2024 15:54	JavaScript-Quelldatei
DRS_UI_RecordedSpec.js	13.12.2024 22:15	JavaScript-Quelldatei
DRS_UI_RecordedRed.js	14.12.2024 01:46	JavaScript-Quelldatei
DRS_UI_ProcessedSpec.js	13.12.2024 23:50	JavaScript-Quelldatei
DRS_UI_EQ_grid_jsui.js	19.11.2024 23:48	JavaScript-Quelldatei
DRS_UI_EQ_curves_jsui.js	13.12.2024 00:31	JavaScript-Quelldatei
DRS_UI_CorrectionCurves.js	13.12.2024 21:54	JavaScript-Quelldatei
DRS_UI_CorrectedSpec.js	13.12.2024 22:55	JavaScript-Quelldatei
DRS_functions_gen.js	13.12.2024 00:51	JavaScript-Quelldatei
DRS_DSP_SpectralComp.gendsp	17.12.2024 10:12	GENDSP-Datei
DRS_DSP_MSEnc.gendsp	13.12.2024 14:31	GENDSP-Datei
DRS_DSP_MSDec.gendsp	10.11.2024 12:32	GENDSP-Datei
DRS_DSP_Mix_MSDec.gendsp	13.12.2024 16:34	GENDSP-Datei

Abbildung 4.3: Ordnerstruktur des entwickelten Audio-Plugins (Eigene Darstellung)

4.2.3 Hierarchieebenen der Implementierung

In Abbildung 4.4 ist die Patcher- und Funktionsstruktur der Implementierung als Übersicht dargestellt. Das Ziel der Struktur besteht in der Aufteilung von Steuerungselementen und DSP-Elementen sowie der Übersichtlichkeit im Code. Ein Vorteil der gewählten Anordnung liegt in dem zentralen Management gleichartiger Objekte wie *buffer~*-Objekte innerhalb eines (Sub-)Patchers. Gleichzeitig wurde das Anordnen der GUI-Elemente beim Programmieren erleichtert, da diese in Max nur durch *bpatcher*-Objekte für darüberliegende Parent Patcher visuell sichtbar gemacht werden können.

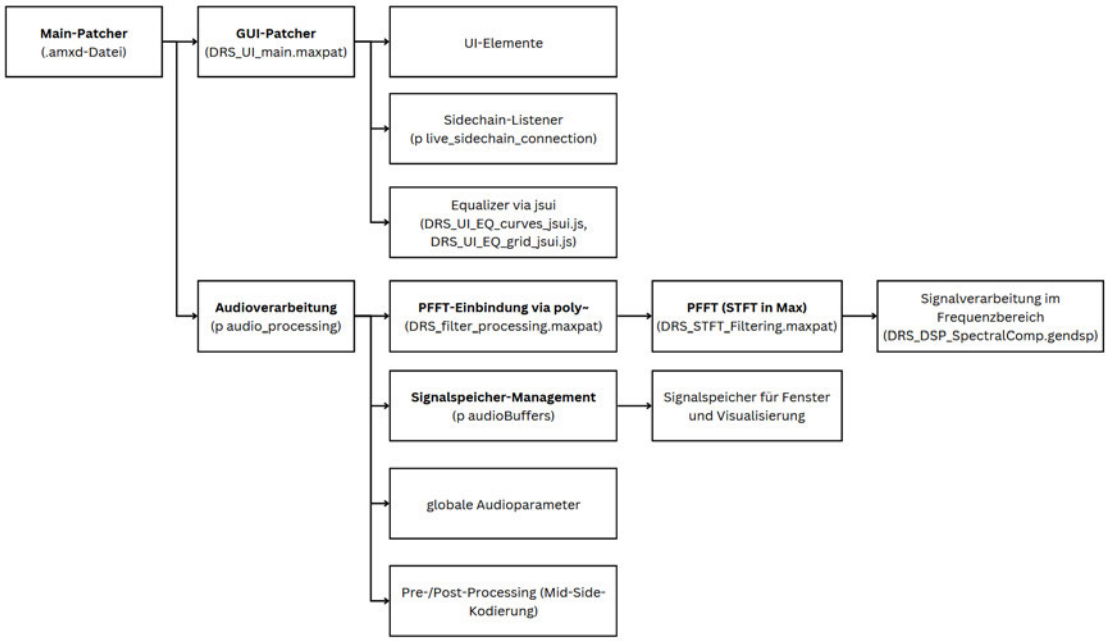


Abbildung 4.4: Darstellung der implementierten Patcher-Hierarchie (Eigene Darstellung)

4.2.4 Main-Patch

Abbildung 4.5 zeigt die implementierte Struktur des Main-Patchers des Plugins. Darin sind drei wesentliche Blöcke erkennbar. Der Block mit der Bezeichnung *GUI* fässt alle

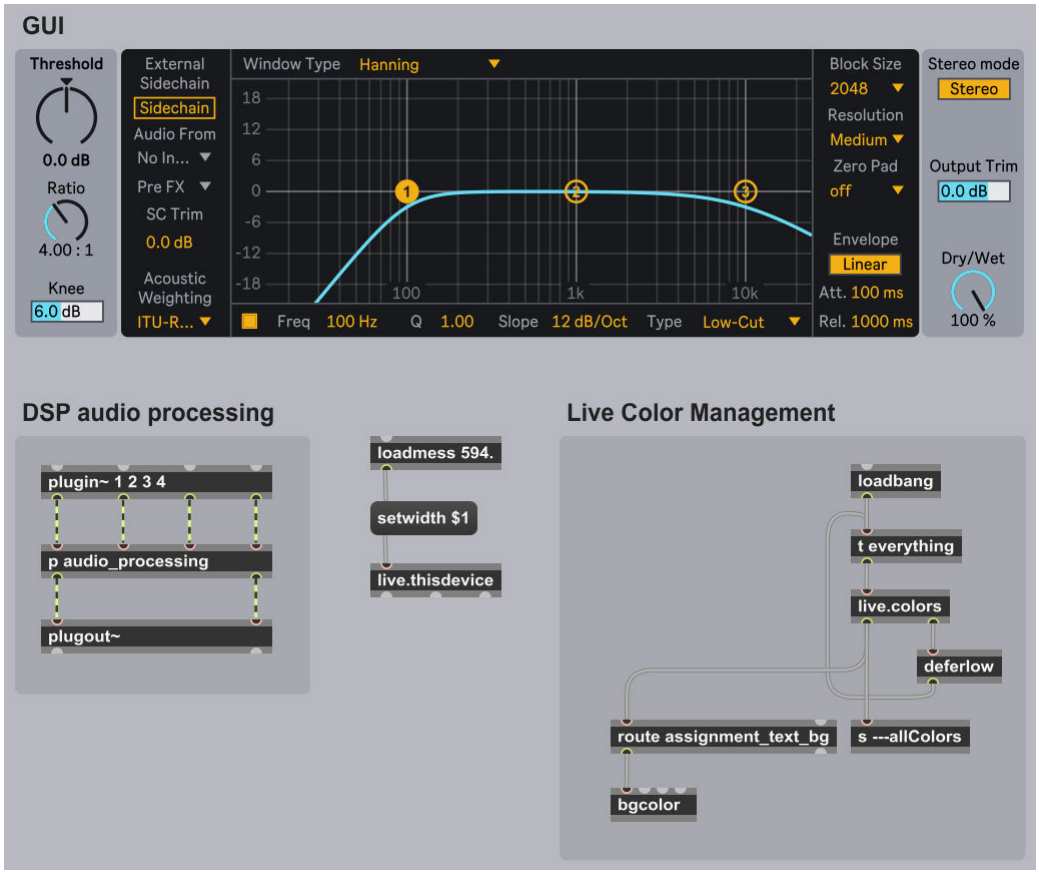


Abbildung 4.5: Main-Patch des Plugins in Max (Eigene Darstellung)

GUI-Elemente innerhalb eines *bpatcher*-Objekts zusammen. Dieses bindet über die Option *Patcher File* im Inspector-Fenster die Datei *DRS_UI_main.maxpat* ein. Im Gegensatz zu *patcher*-Objekten kann ein Ausschnitt der Oberfläche des referenzierten Subpatchers im Parent Patcher sichtbar gemacht werden. Im Block *DSP audio processing* findet die Signalverarbeitung innerhalb des Subpatchers *audio_processing* statt. Der Signalfluss ist in allen folgenden gezeigten Patchern von oben nach unten zu lesen. Der Block *Live Color Management* dient der visuellen Integration in Ableton Live durch das Erfassen des Farbschemas (Theme) über die Live API. Eine wichtige Rolle spielen die Objekte *loadbang* und *loadmess*. Diese dienen der Initialisierung von Parametern durch Senden eines Bangs oder einer im Argument spezifizierten Message beim erstmaligen Laden des Plugins.

4.2.5 Audiosignalverarbeitung mit MSP

Der in Abbildung 4.6 dargestellte Subpatcher *audio_processing* implementiert die Grundstruktur der Audioverarbeitung des Plugins. Das dem *poly~* vorgeschaltete *gen~*-Objekt

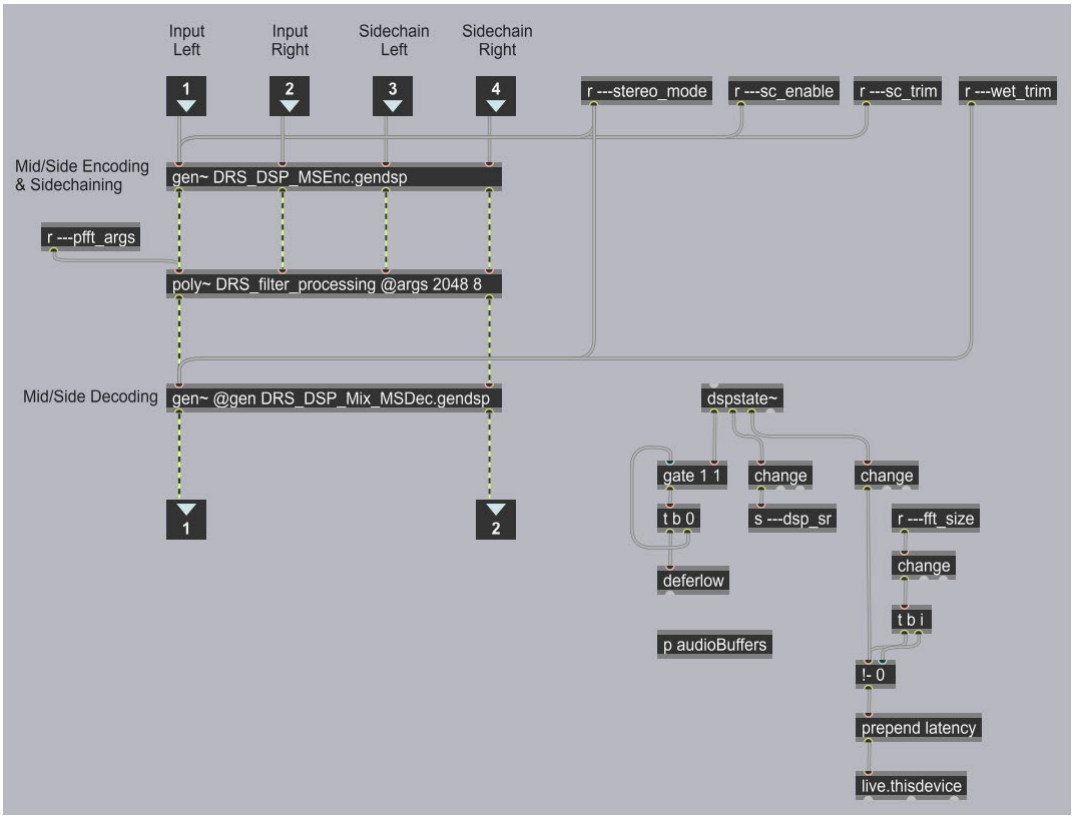


Abbildung 4.6: Für die Audioverarbeitung zuständiger Subpatcher (Eigene Darstellung)

implementiert das Enkodieren der Stereokanäle in die Mid-Side-Konfiguration sowie die Wahl eines Sidechain-Eingangs beim Aktivieren dieser Optionen. Der sich am Ende der Signalkette befindliche Gen-Patcher dekodiert die Signale ggf. wieder in *Links* und *Rechts*. Das STFT-Processing findet innerhalb des *poly~*-Patchers statt. Die *receive*-Objekte in dem Patcher, implementiert über die Objekte mit der Bezeichnung *r —BEZEICHNER*, empfangen die Messages der gleichnamigen *send*-Objekte aus dem GUI-Subpatcher. Sie dienen als Verknüpfungspunkte der über das GUI steuerbaren Parameter und dem DSP. Dadurch können z.B. die Parameter der STFT gesteuert werden. Das Voranstellen von drei Bindestrichsymbolen (-) bei der Namensdeklaration im Argument des Objekts ist für die Zuweisung einer einzigartigen Identifikationsnummer bei einer Instanzierung des Plugins nötig. Dies verhindert eine ungewollte Kommunikation der Parameter mehrerer M4L-Device-Instanzen untereinander. Das *dspstate~*-Objekt stellt Kennzahlen wie die eingestellte Abtastfrequenz zur Verfügung. In Max for Live sind diese Werte fest an die Einstellungen der DAW gekoppelt. Während der Programmierung wurde mit einer festgelegten Abtastfrequenz von $f_A = 44100\,\mathrm{Hz}$ gearbeitet.

Fensterfunktionen und Signalspeicher

Abbildung 4.7 zeigt den angelegten Subpatcher *audioBuffers* mit einer Reihe an *buffer~*-Objekten. Diese dienen in Max der Speicherung von Signalen mit im Argument des

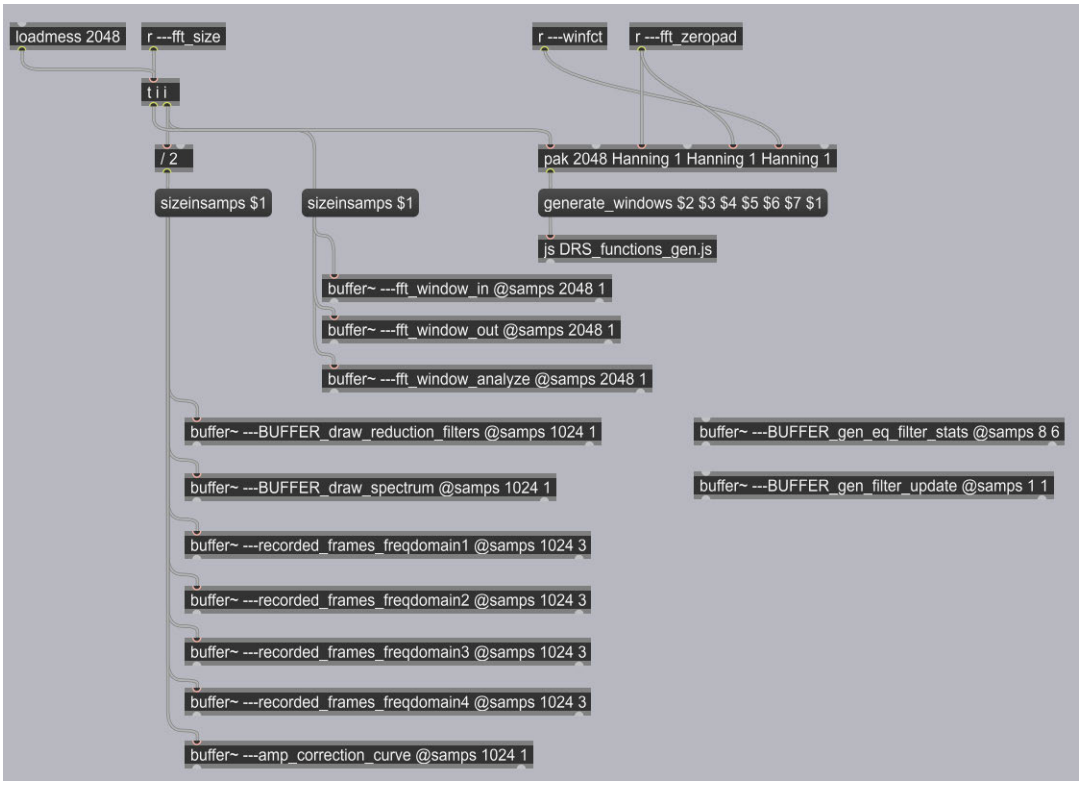


Abbildung 4.7: Gesammeltes Management aller Signal-Buffer (Eigene Darstellung)

Objekts festgelegten Längen und Anzahl an Kanälen. Die generierten Fensterfunktionen für die Analyse und Resynthese werden in den entsprechenden *buffer*~-Objekten mit dem Namensschema *—fft_window_NAME* gespeichert. Die Anzahl zu speichernder Samples wird über die Message *sizeinsamps* dynamisch an die verwendete FFT-Länge angepasst. Ein angelegtes *js*-Objekt bindet die JavaScript-Datei *DRS_functions_gen.js* ein und sorgt für die Generierung der Fensterfunktionen je nach spezifizierter Fensterlänge. Im Quellcode 4.1 wird deutlich, dass die *buffer*~-Objekte auch im JavaScript-Code durch Aufruf der Buffer-Klasse referenziert werden können.

```
1 inlets = 1;
2 outlets = 1;
3
4 var fft_window_in_buffer = new Buffer("---fft_window_in");
5 ...
6
7 var w_fct = new Array();
8 var w_audio_in = new Array();
9 ...
```

Quellcode 4.1: Einbindung von *buffer*~-Objekten in JavaScript

Die in Quellcode 4.2 gezeigte Funktion *generate_windows* nimmt als Argumente Parameter wie die FFT-Länge für alle Fensterfunktionen auf. Die Zero-Padding-Faktoren stellten einen Versuch der Implementierung von Zero Padding in Max dar. Dieser war jedoch, wie

im Folgenden näher beschrieben wird, nicht erfolgreich. Eine Besonderheit ist, dass die Funktion von außen in Max durch Senden einer Message mit dem Funktionsnamen und Argumenten an das *js*-Objekt aufgerufen werden kann. In Abbildung 4.7 ist die gleichnamige Message mit einer Reihe an Platzhaltern für diese Argumente erkennbar. Wird also z.B. die Fensterlänge von außen geändert, generiert die Funktion alle Fensterfunktionen für die eingestellten Parameter neu. In den Arrays mit der Namenskennung *w_NAME* werden die einzelnen Samples der Fenster gespeichert und mithilfe der *poke()*-Funktion in das jeweilige *buffer*~-Objekt eingelesen.

```
1  function generate_windows(window_audio_in, zero_pad_in, window_audio_out,
       zero_pad_out, window_analyze, zero_pad_analyze, length)
2  {
3      w_audio_in = new Array();
4      ...
5
6      w_audio_in = window_function(window_audio_in, length, zero_pad_in);
7      ...
8
9      fft_window_in_buffer.poke(1, 0, w_audio_in);
10     ...
11 }
```

Quellcode 4.2: Funktion für die Generierung der Fensterfunktionen

Die Berechnung der Fensterfunktionen findet in Quellcode 4.3 statt. Der Zero-Padding-Faktor teilt die eigentliche Fensterlänge durch den angegebenen Faktor. Wird z.B. ein Faktor von 2 und eine Fensterlänge von $N = 2048$ gewählt, wird die Fensterfunktion für die Länge $M = 2048/2 = 1024$ berechnet und die übrigen Samples mit Nullen aufgefüllt.

```
1  function window_function(window, length, zero_pad){
2      var zp = arguments.length >= 3 ? zero_pad : 1;
3      w_fct = new Array();
4      var M = Math.floor(length / zp);
5      for(var n = 0; n < length; n++){
6          if(n < M){
7              switch(window){
8                  case "Hanning":
9                      w_fct[n] = 0.5 * (1. - Math.cos(2. * Math.PI * n / (M -
                           1)));
10                     break;
11                 case "Rectangle":
12                     w_fct[n] = 1.;
13                     break;
14                 ...
15             }
16         } else {
17           w_fct[n] = 0.;
18         }
```

```
19      }
20      return w_fct;
21  }
```

Quellcode 4.3: Berechnungsfunktion für die Fensterfunktionen

Weitere *buffer*~-Objekte im Patcher aus Abbildung 4.7 dienen der Visualisierung der Daten, wobei die halbe FFT-Länge als Speichergröße zu beachten ist. Dies rührt auf der im Folgenden erläuterten Implementierung der STFT in Max, welche aufgrund der in Abschnitt 3.1.1 erwähnten Symmetrieverhältnisse für reellwertige Signale lediglich die halben Spektren ausgibt. Außerdem werden, wie im Folgenden näher behandelt, *buffer*~-Objekte zur Übergabe von Daten aus dem GUI an die Signalverarbeitungssektion genutzt.

STFT mithilfe des *pfft*~-Objekts

Die STFT konnte unter Nutzung des *pfft*~-Objekts programmiert werden. Dieses implementiert nativ ein System mit der Spektraldarstellung eines reellwertigen Signals mithilfe der FFT, Audioverarbeitung im Frequenzbereich, Resynthese in den Zeitbereich mittels Inverser FFT und der nötigen Fensterung sowie dem Overlap-Adding [77]. Voraussetzung ist auf Basis des FFT-Algorithmus die Wahl der Fensterlänge und des Overlap-Faktors als eine Zweierpotenz. Die Hop Size berechnet sich dabei als Quotient der gewählten Fensterlänge dividiert durch den Overlap-Faktor. Das *pfft*~-Objekt selbst fungiert jedoch lediglich als Host-Patcher für eine darin eingebettete Abstraction, welche zwingend auf dem Computer als *.maxpat*-Datei gespeichert sein muss. Jegliches Processing in dieser Abstraction findet dann im Spektralbereich statt.

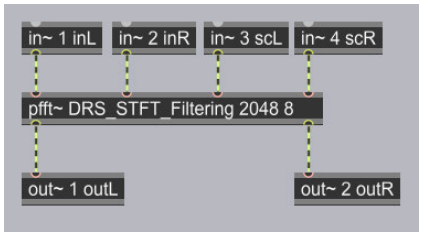


Abbildung 4.8: Im Poly-Patcher eingebetteter PFFT-Patcher (Eigene Darstellung)

Abbildung 4.8 zeigt den Patcher mit dem eingesetzten *pfft*~-Objekt, welcher als Abstraction in dem in Abbildung 4.6 gezeigten *poly*~-Objekt eingebunden ist. Der Nutzen des *poly*~-Objekts liegt in der Parametrisierung der Fensterlänge und des Overlap-Faktors in Max. Anders als bei den meisten anderen Objekten können diese Parameter für das *pfft*~-Objekt von außen nicht direkt durch Messages modifziert werden. Der Nutzer sollte jedoch Zugriff auf jene Werte haben, weshalb ein Umweg über das *poly*~-Objekt gefunden wurde. Durch die Einbettung eines Patchers mit besagtem *pfft*~-Objekt ist es möglich, dessen Fensterlänge und Overlap-Faktor als Platzhalter für Argumente mit der Bezeich-

nung *#1* und *#2* im Patching Mode einzutragen. Werden die tatsächliche Fensterlänge und der Overlap-Faktor als Attribute in beschriebenem *poly*~-Objekt mit vorangestelltem Attributbezeichner *@args* übergeben, so werden die Platzhalter im darunterliegenden Patcher durch diese Attribute bei der Instanzierung ersetzt. Eine Reinstanzierung ist mit

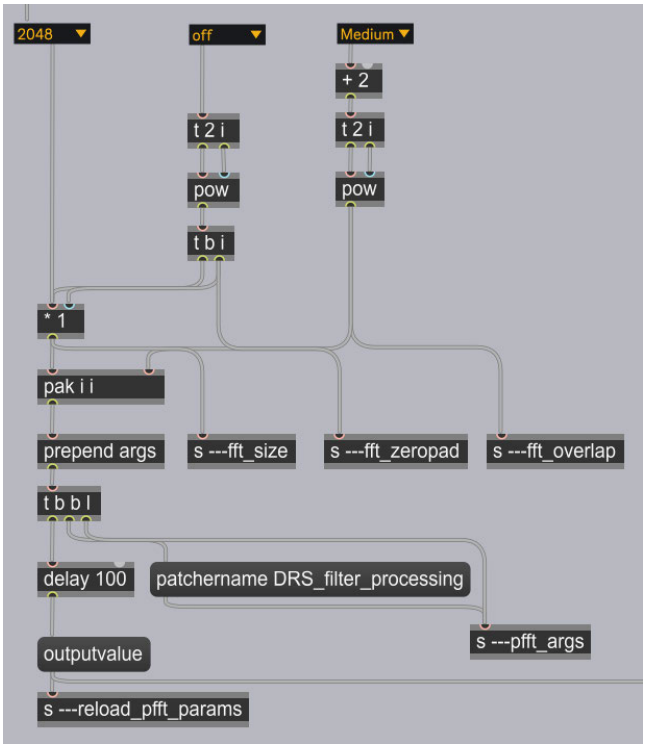


Abbildung 4.9: Einrichtung zur Reinstanzierung des PPFT-Patchers zur Parametrisierung der STFT-Parameter (Eigene Darstellung)

dem in Abbildung 4.9 gezeigten Patch möglich. Bei Änderung der Fensterlänge oder des Overlap-Faktors wird durch das Senden von zwei unmittelbar nacheinander folgenden Messages die angegebene Abstraction des *poly*~-Objekts neu instanziert. Die erste Message muss dabei die beiden Kennzahlen Fensterlänge und Overlap-Faktor beinhalten, die zweite Message leitet mit der Bezeichnung *patchername DRS_filter_processing* die Reinstanzierung des gleichnamigen Patchers ein.

Die *pfft*~-Umgebung ist in der Lage, mehrere STFTs parallel durchzuführen und damit mehrere Audiosignale gleichzeitig in den Spektralbereich zu überführen. Dieser Vorteil wurde in der Implementierung genutzt. Die Zahl der parallel zu analysierenden Audiosignale wird über die Anzahl der *fftin*~-Objekte im durch das *pfft*~-Objekt eingebundenen Patcher bestimmt, die Zahl der anzuwendenden Inversen FFTs durch die Anzahl der *fftout*~-Objekte. Abbildung 4.10 stellt den durch das *pfft*~-Objekt eingebundenen Patcher dar. In diesem Plugin werden zwei Stereopaare, d.h. die Signale für die Analyse des Sidechain-Eingangs und das Signal, auf welches die Filterung angewandt werden soll, als Eingangssignale genutzt. Das Ausgangssignalpaar bildet das verarbeitete Stereosignal. *fftin*~-Objekte haben im Gegensatz zu üblichen *inlet*-Objekten drei Outlets. Diese geben

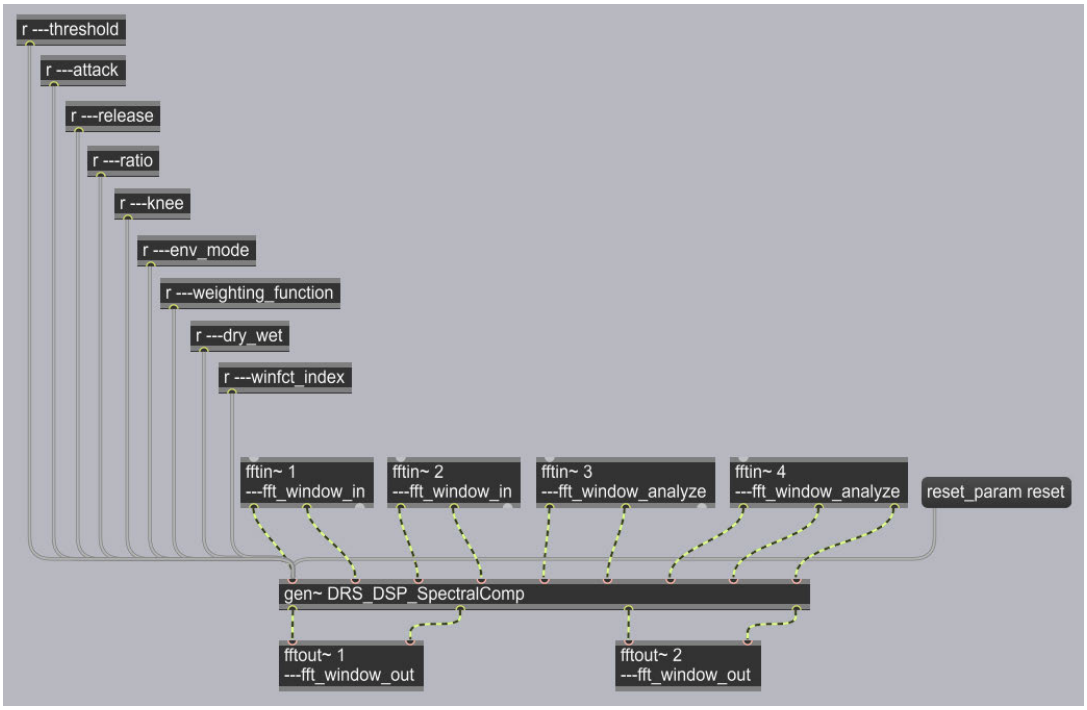


Abbildung 4.10: PFFT-Patcher mit Signalverarbeitung im Spektralbereich (Eigene Darstellung)

in der Reihenfolge von links nach rechts den Realteil, den Imaginärteil und den Index der jeweiligen Frequency Bin aus. *fftout*~-Objekte erwarten entsprechend den Realteil und den Imaginärteil für die Rücktransformation an ihren Inlets. Bei der Programmierung war zu beachten, dass die Anzahl der verfügbaren Frequency Bins bei *pfft*~-Objekten standardmäßig aufgrund der Symmetrieverhältnisse für reellwertige Eingangssignale nur die Hälfte der FFT-Länge N entspricht. Das Outlet für den Index gibt dadurch nur die Bin-Indizes $0, ..., N/2 - 1$ aus. Es ist dabei ausreichend, die Indizes nur bei einem der hier vier *fftin*~-Objekte abzugreifen. Die Argumente in den *fftin*~- bzw. *fftout*~-Objekten referenzieren die gleichnamigen *buffer*~-Objekte aus Abbildung 4.7 für die Nutzung der darin gespeicherten Fensterfunktionen.

Im Zuge der Implementierung ergab sich das datenstromorientierte Prinzip von Max unter Nutzung der *pfft*~-Umgebung als Nachteil. So werden die Frequency Bins von $0, ..., N/2 - 1$ nacheinander in dem darin enthaltenen Subpatcher verarbeitet. Dadurch können die Amplitudenwerte auch nur in dieser Reihenfolge abgegriffen werden. Eine gleichzeitige Ausgabe aller Bins aus den *fftin*~-Objekten ist dagegen nicht möglich. So können Algorithmen zwar von vorher ausgegebenen Bins abhängig gemacht werden, jedoch nicht von Bins mit höheren als dem aktuellen Index. Infolgedessen stellte sich heraus, dass die Implementierung einer vom aktuellen Spektrum abhängigen linearen Faltung über die FFT in Max ein Problem darstellt. So wäre hierfür nötig, ein vom derzeitigen Spektrum abhängiges neues Spektrum zurück in den Zeitbereich zu überführen, das Zero Padding durchzuführen und das resultierende Signal zurück in den Frequenzbereich zu transformieren. Eine spektrale Multiplikation mit dem ebenfalls unter Zero Padding fre-

quenztransformierten und zu filternden Signal ergebe dann nach Rücktransformation ein der linearen Faltung entsprechenden Ergebnis. Dadurch können unter Anwendung der STFT zeitvariable FIR-Filter in Abhängigkeit vom Eingangsspektrum implementiert werden. Die datenstromorientierte Arbeitsweise des *pfft~*-Objekts verhindert dies jedoch, da diese Schrittfolge das Verschachteln mehrerer FFTs in der *pfft~*-Umgebung mit gleichzeitiger Ausgabe der Werte, Anhängen von Nullen und die Berechnung unterschiedlicher FFT-Längen erfordern würde. Die WOLA-Methode in Max hingegen unterdrückt Fehler der zirkulären Faltung an den Rändern eines Frames im Zeitbereich bis zu einem gewissen Grad. Es muss vom Anwender abgewogen werden, inwiefern diese Methode für eine Applikation klanglich effektiv ist oder unter Umständen auf andere Plugins zurückgegriffen werden muss, welche ggf. eine zeitvariable lineare Faltung ermöglichen.

Es ist anzumerken, dass die Nutzung von Objekten mit Sample-Zwischenspeichern wie das *pfft~*-Objekt zu einer messbaren Latenz zwischen Ein- und Ausgangssignal führt. Dies ist auf die zugrundeliegende FFT zurückzuführen, welche erst eine gewisse Anzahl an Samples analysiert haben muss, um eine vollständige Ausgabe der Werte zu ermöglichen. Je größer die FFT-Länge gewählt wird, desto höher ist die Latenz. Hohe Werte sind deshalb ggf. für zeitkritische Anwendungen wie dem *Tracking* bei Studioaufnahmen von Nachteil. Ableton Live verfügt über eine interne Latenzkompensation. In Max musste dafür eine Logik implementiert werden, welche die anhand der FFT-Länge berechnete Latenz an die DAW übermittelt.

4.2.6 Signalverarbeitung im Frequenzbereich in Gen

Der durch das *gen~*-Objekt referenzierte Gen-Patcher *DRS_DSP_SpectralComp* in Abbildung 4.10 ist für den eigentlichen Frequenzbewertungsprozess und der Bildung der effektiven Filter im Frequenzbereich zuständig. Es wurde sich hier aufgrund der nötigen individuellen Bewertung der Frequency Bins für die Gen-Umgebung entschieden, da einfache MSP-Objekte in dieser Hinsicht wegen ihrer Verarbeitung von zusammengefassten Signalvektoren, wie in Abschnitt 4.1.3 beschrieben, nicht die nötige Präzision aufweisen. Im Rahmen dieser Implementierung wurde sämtlicher Programmcode innerhalb eines *codebox*-Objekts in Form von GenExpr-Code verfasst, da die Anwendung von *for*-Schleifen und *if/else*-Statements für die Abfrage von Bedingungen eine wesentliche Rolle für die Funktion des Plugins spielten. Im Folgenden werden wesentliche Ausschnitte des Programmcodes genauer erläutert, welche hier vor allem der Signalverarbeitung und nicht der Darstellung auf dem GUI dienlich sind.

Deklaration von Parametern und Speicherobjekten

```
1 Data DAT_filtcurve(fftsize / 2, 1);
2 Data DAT_freqwght(fftsize / 2, 5);
```

```
3  Data DAT_prevwindow(fftsize / 2, 2);
4
5  Buffer BUF_filtstats("---BUFFER_gen_eq_filter_stats");
6  Buffer BUF_filtupdate("---BUFFER_gen_filter_update");
7
8  History reset(1);
9  History filtrecalc(1);
10
11 Param weightfct(3, min = 0, max = 6);
12 Param threshold(0., min = -48., max = 48.);
13 Param ratio(4., min = 1., max = 64.);
14 Param knee(6., min = 0., max = 36.);
15 Param attack(100., min = 0., max = 3000.);
16 Param release(1000., min = 0., max = 10000.);
17 Param env_mode(0, min = 0, max = 1);
18 Param dry_wet(1., min = 0., max = 1.);
19 Param winfct_index(0., min = 0., max = 7.);
```

Quellcode 4.4: Deklaration von *Buffer*-, *Data*-, *History*- und *Param*-Objekten

Dafür wurden im Code eine Reihe an *Data*-, *Buffer*-, *History*- und *Param*-Objekten angelegt. Die Deklaration dieser Objekte muss in der Gen-Umgebung vor den ausführenden Prozeduren erfolgen. Die *Data*-Objekte speichern hierbei Daten für die berechneten Filterkurven und die Spektraldaten vorhergehender Daten für die Glättung der spektralen Hüllkurven der einzelnen Bins. Die *Buffer*-Objekte referenzieren die gleichnamigen *buffer*~-Objekte aus Abbildung 4.7 zur Kommunikation mit dem GUI. Die *Param*-Objekte ermöglichen die Einbindung von durch Messages veränderbare Parameter in Gen, welche an das erste Inlet des *gen*~-Objekts gesendet werden und dienen ebenfalls der Parametrisierung durch die Nutzerinteraktion mit dem GUI. Diese Messages werden hier durch die *receive*-Objekte in Abbildung 4.10 aus dem GUI-Patcher empfangen.

Berechnung der Bin-Breite und Amplitudenkompensation

```
1 CONST_bin_width = samplerate / fftsize;
2 CONST_wcompensation = 1.;
3 if(winfct_index == 0){
4   CONST_wcompensation = 2.;
5 } else if(winfct_index == 1){
6   CONST_wcompensation = 1.;
7 } else if ...
8 }
```

Quellcode 4.5: Kompensationsfaktoren und Bin-Breite für die Fenster

Im nächsten Schritt werden die konstante Frequenzbreite Δf der Frequency Bins und der Amplitudenkompensationsfaktor in Abhängigkeit von dem gewählten Fenstertyp be-

rechnet. Quellcode 4.5 zeigt die Berechnung für zwei Fensterfunktionen, dem Hanning- und dem Rechteckfenster. Diese Faktoren sind nötig, da die gemessenen Amplituden durch die Fensterung verzerrt, d.h. reduziert werden. Da die Gen-Umgebung den Datentyp *String* nicht verarbeiten kann, wurden für jedes verfügbare Fenster Indizes bzw. einfache Identitätsnummern vergeben, anhand deren das ausgewählte Fenster in Gen erkannt und die Berechnung auf Basis dessen stattfindet.

Implementierung standardisierter Frequenzbewertungskurven

```
1  if(reset == 1){
2    for(i = 0; i < (fftsize / 2); i += 1){
3      wght_fct = 0.;
4      bin_freq = i * CONST_bin_width;
5      bin_freq2 = bin_freq * bin_freq;
6      bin_freq3 = bin_freq2 * bin_freq;
7      bin_freq4 = bin_freq3 * bin_freq;
8      ...
9      // A-weighting function
10     wght_fct = atodb((12194. * 12194. * bin_freq4) / ((bin_freq2 + 20.6 *
           20.6) * sqrt((bin_freq2 + 107.7 * 107.7) * (bin_freq2 + 737.9 *
           737.9)) * (bin_freq2 + 12194 * 12194))) + 2.;
11     poke(DAT_freqwght, wght_fct, i, 3);
12     ...
13   }
14   reset = 0;
15 }
```

Quellcode 4.6: Berechnung der Frequenzbewertungskurven

Quellcode 4.6 zeigt die implementierten Berechnungsvorschriften für die Frequenzbewertung am Beispiel der A-Bewertung. Da die Gen-Umgebung mit jedem eingehenden Sample die Berechnungen für die gleichen Werte neu anstoßen würde, wurden die Berechnungen mithilfe eines *History*-Objekts namens *reset* und einer *if*-Bedingung aus dem fortlaufenden Prozess ausgelagert. So werden die gleichbleibenden Frequenzbewertungskurven nur einmalig bei Initialisierung der Patcher berechnet und in den *Data*-Objekten abgespeichert, da *reset* nach dem ersten Durchlauf auf 0 gesetzt wird und die Bedingung für die Berechnung nicht mehr erfüllt ist. Dies dient der Einsparung an CPU-Ressourcen. Auch für die weiteren Berechnungen der EQ-Filterkurven wurden solche Mechanismen einprogrammiert, sodass diese nur bei Änderung der Filterparameter kalkuliert und redundante Berechnungen vermieden werden. Die Implementierung ist ähnlich zu der hier beschriebenen und soll im Folgenden nicht weiter ausgeführt werden.

Berechnung der spektralen Amplituden

```
1 SIG_resg_inL, SIG_imsg_inL = in1, in2;
2 SIG_resc_inL, SIG_imsc_inL = in5, in6;
3 bin = in9;
4 w = CONST_bin_width * bin;
5
6 SIG_ampsc_dbL = atodb(CONST_wcompensation * 2. * sqrt(SIG_resc_inL *
    SIG_resc_inL + SIG_imsc_inL * SIG_imsc_inL) / fftsize);
```

Quellcode 4.7: Berechnung und Kompensation der Amplitudenwerte

Der Quellcode in 4.7 spiegelt die Berechnung und Kompensation der spektralen Amplituden wider. Für alle nachfolgenden Abschnitte sollen sich die Betrachtungen auf den linken Kanal eines Stereopaares beschränken, da die Berechnungen für den rechten Kanal redundant sind und sich nur in den Variablennamen unterscheiden. Die Variable *w* berechnet die Frequenz der Frequency Bins, d.h. die DFT-Rasterpunkte auf der Frequenzachse zur Berechnung der EQ-Filterkurven. Im Anschluss wird die Amplitude des Analyse-Signals nach der Fensterkompensation in Dezibel berechnet. Außerdem wird durch den Faktor 2 berücksichtigt, dass lediglich die positive Hälfte des Spektrums bei der Auswertung aufgrund der Redundanz der spektralen Spiegelung betrachtet wird.

EQ-Implementierung

Die zusätzlich implementierten und durch den Anwender parametrisierbaren Filterkurven sind für gängige parametrische EQs typisch und wurden anhand ihrer analogen Übertragungsfunktionen $H(s)$ implementiert. Üblicherweise wäre eine Überführung in eine zeitdiskrete Übertragungsfunktion $H(z)$ mithilfe bekannter Methoden des Designs von IIR-Filtern wie die bilineare Transformation nötig, wenn die Filter im Zeitbereich implementiert würden. Da jedoch für eine Bewertung der Amplituden nur die Betragsfunktion $|H(s)|$ von Interesse ist und die Berechnungen in diesem Gen-Patcher im Spektralbereich stattfinden, wurden hierfür direkt die umgeformten Betragsfunktionen der Filter genutzt. Die Betragsfunktionen $|H(s)|$ wurden aus den für verschiedene Filtertypen gegebenen Übertragungsfunktionen in [78] errechnet und in Code übersetzt, indem die Laplace-Variable s durch $j\omega$ ersetzt und der frequenzabhängige Betrag gebildet wurden. So stehen dem Anwender die Filtertypen Peak-, Notch-, Bandpass-, Highshelf- und Lowshelf-Filter zweiter Ordnung sowie Hoch- und Tiefpassfilter zur subjektiven Frequenzbewertung zur Verfügung. Die implementierten Hoch- und Tiefpassfilter besitzen hingegen eine variable Flankensteilheit und damit Filterordnung.

Quellcode 4.8 zeigt die Berechnungen anhand der Beispiele von Tiefpass- und Peak-Filter. Das System geht bei Neuberechnung der Filterkurven durch Parameteränderun-

gen von einer initialen Gesamtbetragsfunktion in Dezibel von *Hs_dB = 0* aus. Danach werden die Filterparameter für jedes Filter aus dem referenzierten *buffer*~-Objekt über die *peek()*-Befehle entnommen und bei eingeschaltetem Filter, ersichtlich durch die Variable *enable*, die entsprechenden Übertragungsfunktionen an der Bin-Frequenz *w* kalkuliert und akkumulativ in logarithmierter Einheit aufsummiert. Dem Anwender stehen neben dem auswählbaren Filtertyp Parameter wie Grenz- bzw. Mittenfrequenz, Verstärkung bzw. Dämpfung für Peak- und Shelving-Filter, der Gütefaktor Q, die Steilheit für Hoch- und Tiefpass und der On-/Off-Schalter für jedes einzelne Filter zur Verfügung. Bis zu acht Filter sind hier in Serie geschaltet maximal möglich. Es wurde sich aufgrund der Begrenzung der CPU-Last für diese Anzahl entschieden, welche hier durch die Anzahl der Samples des in Abbildung 4.7 gezeigten *buffer*~-Objekts mit der Bezeichnung *—BUFFER_gen_eq_filter_stats* begrenzt wird. Das Objekt dient dabei als 8x6-Matrix für die gespeicherten Filterparameter für sechs verschiedene Parameter in den sechs Kanälen und für jeweils acht Filter.

```
1  ampcorrection_db = peek(DAT_freqwght, bin, weightfct);
2
3  Hs_abs_square = 1.;
4  Hs_dB = 0.;
5  for(i = 0; i < dim(BUF_filtstats); i += 1.){
6    type   = peek(BUF_filtstats, i, 0);
7    w0     = peek(BUF_filtstats, i, 1);
8    gain   = peek(BUF_filtstats, i, 2);
9    q_val    = peek(BUF_filtstats, i, 3);
10   slope    = peek(BUF_filtstats, i, 4);
11   enable   = peek(BUF_filtstats, i, 5);
12
13   if(enable == 1){
14     wdw0 = w / w0;
15     wdw02 = wdw0 * wdw0;
16     A = pow(10., (gain / 40.));
17     N = slope / 6.;
18     uneven = round(N % 2.);
19     if(type == 2){    // Lowpass
20       Q = atan(0.1 * q_val) / atan(0.1);
21       a = wdw02 - 1.;
22       for(n = 0; n <= floor((N/2) - 1); n += 1){
23         angle = 2. * pi * ((2. * n + N + 1) / (4. * N));
24         b = 2. * wdw0;
25         if(n == 0){
26           b *= (cos(angle) / Q);
27         } else {
28           b *= cos(angle);
29         }
30         Hs_abs_square = 1. / (a * a + b * b);
31         Hs_dB += 10. * log10(Hs_abs_square);
32       }
```

```
33        if(uneven == 1){
34          Hs_abs_square = 1. / (1. + wdw02);
35          Hs_dB += 10. * log10(Hs_abs_square);
36        }
37      ...
38      } else if(type == 5){   // Peak
39        Q = q_val / SQRT2;
40        a = wdw02 - 1.;
41        b = wdw0 * (A / Q);
42        c = wdw0 / (A * Q);
43        Hs_abs_square = (a * a + b * b) / (a * a + c * c);
44        Hs_dB += 10. * log10(Hs_abs_square);
45      }
46      ...
47    }
48  }
49  weightedamp_dbL = SIG_ampsc_dbL + ampcorrection_db + Hs_dB;
```

Quellcode 4.8: Berechnung der Filterkurven und Frequenzbewertung

Im Quellcode 4.8 wird zudem die Besonderheit bei der Berechnung von Tief- bzw. Hochpassfiltern mit variabler Ordnung N deutlich. Diese orientieren sich am sog. Butterworth-Design. Die Filterordnung bestimmt dabei die Steilheit des Filters und wird in Software-EQs oft als *Slope* in dB/Oktave angegeben. Die Erhöhung der Filterordnung um 1 bedeutet hierbei die Erhöhung der Steilheit des Amplitudengangs im Übergangsbereich um rund 6 dB/Oktave. Je nach Filterordnung kann ein Filter hoher Ordnung in eine gewisse Anzahl zweipoliger bzw. bei ungerader Filterordnung einem zusätzlichen einpoligen Filter zerlegt werden. Dies ist auf Basis der Null- und Polstellendarstellung von Butterworth-Filtern möglich, wobei die Position der Polstellen über den Winkel *angle* im Code berechnet werden. Die Zerlegung von Filtern hoher Ordnung in mehrere serielle Filter erster und zweiter Ordnung dient im DSP hauptsächlich dem Entgegenwirken von Rundungsfehlern und damit ggf. Instabilitäten bei hohen Filterordnungen. Da die Slope in dem GUI ebenfalls in dB/Oktave angegeben wird, berechnet der zugrundeliegende Algorithmus die Filterordnung N anhand der eben beschriebenen Zusammenhänge. Die *atan()*-Operationen bei der Berechnung der Güte Q dienen hier lediglich einer Beschränkung des Resonanzverhaltens an der Grenzfrequenz bei hohen Q-Werten und wurden hierfür vom Verfasser beigefügt.

Nach der Berechnung der Filter wird die Bin-Amplitude mit den Werten der Frequenzbewertung und den Betragsfunktionen der EQ-Kurven an den jeweiligen Bin-Frequenzen verrechnet. Dies ergibt hier das gewichtete Amplitudenspektrum.

Berechnung der statischen Kompressionskurve pro Bin

```
1  x_dbL = weightedamp_dbL;
2  x_scL = 0.;
3
4  t = threshold;
5  k = knee;
6  r = ratio;
7  if(k == 0.){
8    if(x_dbL < t){
9      x_scL = x_dbL;
10   }
11   else if(x_dbL >= t){
12     x_scL = t + ((x_dbL - t) / r);
13   }
14 }
15 else if(k > 0.){
16   if(x_dbL < (t - (k / 2.))){
17     x_scL = x_dbL;
18   }
19   else if((x_dbL >= (t - (k / 2.))) && (x_dbL <= (t + (k / 2.)))){
20     x_scL = x_dbL + ((((1. / r) - 1.) * (x_dbL - t + (k / 2.)) * (x_dbL - t
         + (k / 2.))) / (2. * k));
21   }
22   else if(x_dbL > (t + (k / 2.))){
23     x_scL = t + ((x_dbL - t) / r);
24   }
25 }
26 g_outL = x_scL - x_dbL;
```

Quellcode 4.9: Berechnung der statischen Kompressionskurve

Wie in Quellcode 4.9 ersichtlich ist, wird je nach eingestelltem Threshold, Ratio und Knee zuerst eine statische Kompressionskurve auf die gewichteten Amplituden angewandt. Die Algorithmen für die Kompression und die nachfolgende zeitliche Glättung orientieren sich am Design digitaler Dynamikkompressoren und wurden aus [79] entnommen. Ersterer implementiert die Kompressionskurve als stückweise definierte Funktion mit einer je nach eingestelltem Knee ausgeprägten quadratischen Funktion zur Glättung des Bereichs um den Schwellwert. Die Parameter entsprechen der in Abschnitt 2.1 erläuterten Funktionsweise, wobei die gewichteten Amplituden als Eingangspegel zu betrachten sind. Bei einem auf den Wert 0 eingestellten Knee wird die Kompressionskurve unabhängig von jenem Parameter. Die *Gain Reduction* ergibt sich aus der im Quellcode 4.9 in Zeile 26 berechneten Differenz und gibt die kalkulierte Reduktion der Amplituden in Dezibel an.

Attack- und Release zur zeitlichen Glättung der Kompression

```
g_inL = g_outL;
g_sL = 0.;
g_prevL = peek(DAT_prevwindow, bin, 0);

Ta = (attack / 1000.) * (samplerate / (4. * ffthop));
Tr = (release / 1000.) * (samplerate / (4. * ffthop));

alpha_a = Ta > 0. ? exp(-log10(9.) / Ta) : 0.;
alpha_r = Tr > 0. ? exp(-log10(9.) / Tr) : 0.;

if(g_inL <= g_prevL){
  g_sL = (alpha_a * g_prevL) + ((1. - alpha_a) * g_inL);
}
else if(g_inL > g_prevL){
  g_sL = (alpha_r * g_prevL) + ((1. - alpha_r) * g_inL);
}
poke(DAT_prevwindow, g_sL, bin, 0);
```

Quellcode 4.10: Glättung des Kompressionseffekts über die Zeit

Der frequenzabhängige Kompressionseffekt wird über die Zeit auf Basis des in Quellcode 4.10 dargestellten Algorithmus geglättet. Es sind zwei verschiedene Glättungsalgorithmen, lineares und exponentielles Smoothing, implementiert. Der gezeigte Code bezieht sich auf den für den Parameter *Envelope* eingestellten Wert *Log* und berechnet eine Hüllkurve mit exponentiellen Anstiegs- und Abklingprofilen. Dieses exponentielle Glättungsfilter entspricht einem IIR-Filter erster Ordnung, wobei zwischen ansteigenden und absinkenden Werten unterschieden wird. Statt einem einzelnen Verzögerungselement wurde ein *Data*-Objekt mit der Bezeichnung *DAT_prevwindow* zur parallelen Speicherung vorangegangener Werte für die Gain Reduction zu jeder Frequency Bin einprogrammiert. Im Vergleich zu Dynamikkompressoren im Zeitbereich ist der Grad der Glättung hier für jede Frequency Bin individuell. Demnach kann für jede Frequency Bin im Spektralbereich eine eigene Hüllkurve der Amplitudenwerte und damit eine eigene Kompressionskurve betrachtet werden. Der Glättungseffekt der Gain Reduction über die Zeit ist mithilfe der *Attack*- und *Release*-Parameter steuerbar. Die Attack-Zeit bestimmt, wie schnell der vollständige Kompressionseffekt über die Zeit bei Überschreiten des Schwellwerts greift. Die Release-Zeit hingegen gibt an, wie lange der Kompressionseffekt nach Unterschreiten des Schwellwerts noch ausklingt. Die Werte sind in dem GUI zwar in (Milli-)Sekunden für den Anwender angegeben, sind jedoch aufgrund der Wirkweise der STFT in *Frames* und durch die damit zusammenhängende Abhängigkeit der Variablen *Ta* und *Tr* von Abtastrate und Hop Size eher als Richtwert zu interpretieren. Die Berechnung nach exakten Werten für Attack und Release ist für den gewählten Ansatz dagegen nicht möglich.

Anwendung des Kompressionseffekts auf das Signal

```
1 redfilt_amplinL = dbtoa(g_sL * dry_wet);
2
3 SIG_resg_outL = SIG_resg_inL * redfilt_amplinL;
4 SIG_imsg_outL = SIG_imsg_inL * redfilt_amplinL;
5
6 out1 = SIG_resg_outL;
7 out2 = SIG_imsg_outL;
```

Quellcode 4.11: Anwendung der spektralen Modifikation auf das Signal

Der Kompressionseffekt aller Bins kann einheitlich durch den Anwender über die Einstellung des *Dry/Wet*-Werts angepasst werden. Dieser Wert dient als prozentualer Anteil des Effekts und besteht im Code als zusätzlicher Multiplikator für die kalkulierte Gain Reduction. Der Dry/Wet-Parameter wird vom Anwender zwischen 0 und 1 bzw. 0% und 100% auf dem GUI gewählt. Im Anschluss wird die auf linearer Skala rückgerechnete Gain Reduction mit den Real- und Imaginärteilen des zu filternden Eingangssignals verrechnet. Da die Phase des Eingangssignals unberücksichtigt bleibt, ist die direkte Multiplikation der Variable *redfilt_amplinL* mit den komplexen Eingangswerten möglich. Die Phasenfunktion und damit auch der Imaginärteil der spektralen Modifikation entsprechen dann einer Konstante des Werts 0.

4.2.7 Implementierung der Benutzeroberfläche

Zur Gestaltung des GUI steht in Max eine Vielzahl an individualisierbaren Objekten zur Verfügung. Diese umfassen u.a. verschiedene Drehregler, Schieberegler, Buttons und Dropdown-Menüs. Sie können ebenfalls über Inlets angesteuert, über die Outlets eingestellte Parameter abgegriffen und damit direkt in die Programmierungs- und Testprozesse eingebunden werden. Über das Inspector-Fenster wird das visuelle Erscheinungsbild dieser Objekte wie das Farbschema oder die Positionskoordinaten auf dem GUI eingestellt. Auch kann der Wertebereich der Parameter oder die Optionsliste von Dropdown-Menüs sowie der initiale Startwert des Parameters bei Instanzierung des Plugins festgelegt werden. Voraussetzung für die Anzeige eines Objekts im GUI ist das Setzen der Option *Include in Presentation* im Inspector-Fenster. Die Objekte haben dann unterschiedliche Koordinaten für die Darstellung im Presentation Mode und Patching Mode. Die schlussendliche Anordnung der Objekte im GUI fand im erstgenannten Modus statt.

Einfache Steuerungselemente

Abbildung 4.11 zeigt einen Ausschnitt der implementierten GUI-Steuerung für einfache Parameter wie Threshold oder Ratio. Dabei wurde zur visuellen Integration in Ableton Live auf *live.*-Objekte zurückgegriffen. Die Outlets der Regler geben den eingestellten Wert als

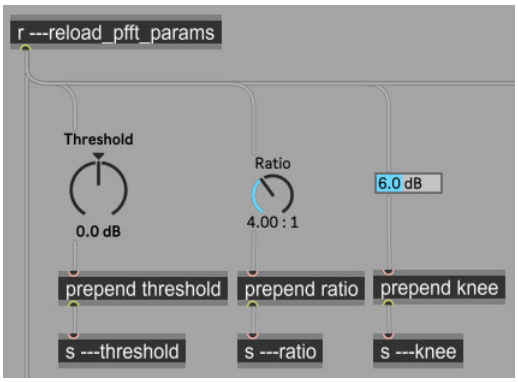


Abbildung 4.11: Parametersteuerung im Patching Mode für ausgewählte Parameter (Eigene Darstellung)

Messages aus. Über die gezeigten *prepend*-Objeke werden die Bezeichnungen in den Argumenten der jeweiligen Message vorangestellt. Dies dient nach Verteilung der Messages durch *send*-Objekte an alle gleichnamigen *receive*-Objekte der Kommunikation mit den gleichnamigen *Param*-Objekten in den Gen-Patchern. Der Zusammenhang wird bei Betrachtung der beiden Abbildungen 4.11 und 4.10 im Abgleich mit dem Quellcode 4.4 des implementierten Gen-Patchers deutlich. Die *outputvalue*-Message aus Abbildung 4.9, welche an alle Regler gesendet wird, dient der erneuten Ausgabe der eingestellten Parameter an den Gen-Patcher, wenn dieser bei Änderung der STFT-Parameter aufgrund der Reinstanzierung des *pfft*~-Objekts auf Standardeinstellungen zurückgesetzt wird.

Implementierung des EQ-Interface

Das EQ-Interface wurde unter Nutzung von *jsui*-Objekten und den zugehörigen JavaScript-Codes entwickelt und über eine programmierte Objektlogik in Max zur Interaktion mit den anderen Steuerungselementen in die GUI-Steuerung integriert. Das Skript speichert außerdem die Filterparameter in einem Array *filter_data* mit für jedes Filter angelegten Dictionaries. Die Parameterwerte als *Values* werden in den Dictionaries den *Keys* als Parameterbezeichner zugeordnet. Diese sind für alle Filter identisch. Die Filterparameter sind über eine entsprechende Anweisung nach dem Schema *filter_data[FILTER-INDEX].FILTER-PARAMETER = NEUER-WERT* änderbar, was für die Interaktion des Anwenders mit dem GUI verknüpft wurde. Um eine intuitive Bedienbarkeit zu gewährleisten, wurde die Steuerung der Filter über Knotenpunkte, sog. *Nodes*, ermöglicht. Dies geschieht durch die Einbindung der in der Dokumentation [80] beschriebenen *Events*, welche z.B. Aktivitäten der Computermaus erfassen. Die exakte Anwendung durch den Nutzenden wird in Kapitel 5 beschrieben. Jeder Knotenpunkt symbolisiert ein Filter auf dem GUI. Diese Knotenpunkte verändern durch das Anwählen und Ziehen mit der Computermaus die Filterparameter. Das GUI visualisiert die sich bei Parametereinstellung ändernden Filterkurven in Echtzeit. Daneben können bis zu acht Knotenpunkte gleichzeitig vor-

handen sein, wobei sie über definierte Nutzerinteraktionen hinzugefügt oder entfernt werden können. Weiterhin grenzen die Variablen mit der Bezeichnung *NAME_range_param* den Wertebereich mit den Werten, welche die Filterparameter annehmen können, ein.

```
var gen_filter_data = new Buffer("---BUFFER_gen_eq_filter_stats");
var gen_filter_update = new Buffer("---BUFFER_gen_filter_update");
var filter_count_max = 8;
var filter_data = [{Type: "Highpass", Freq: 100, Gain: 0, Q: 1, Slope: 12,
    Enable: 1, ui_magn: new Array()},
                   {Type: "Bell", Freq: 1000, Gain: 0, Q: 1, Slope: 12,
                       Enable: 1, ui_magn: new Array()},
                   {Type: "Lowpass", Freq: 10000, Gain: 0, Q: 1, Slope: 6,
                       Enable: 1, ui_magn: new Array()}];
var filter_selected = 0;

var freq_range_param = [10, 22000];
var gain_range_param = [-20, 20];
var q_range_param = [0.05, 20];
var slope_range_param = [6, 96];
```

Quellcode 4.12: Filterparameter in der GUI-Steuerung

Über die *outlet()*-Funktion, ersichtlich in Quellcode 4.13, wurde die Verknüpfung mit der in Max programmierten Logik gewährleistet, indem die Filterparameter als Messages aus dem Outlet des *jsui*-Objekts ausgegeben werden. Dadurch ist die Steuerung und Anzeige der Filterparameter für den Anwender auch über die implementierten GUI-Objekte möglich. Die Variable *filter_selected* stellt den Index des ausgewählten Filters dar.

```
function updateMaxParams(){
    var fs1 = filter_selected + 1;
    outlet(0, "freq", fs1, filter_data[filter_selected].Freq);
    ...
    }
}
```

Quellcode 4.13: Kommunikation der GUI mit Max

Die Kommunikation mit dem beschriebenen Gen-Patcher aus Abschnitt 4.2.6 erfolgt über das angelegte und zuvor beschriebene *buffer~*-Objekt *—BUFFER_gen_eq_filter_stats*. Bei einer Änderung werden die Parameter des angewählten Filters an dieses Objekt weitergegeben und zur Verarbeitung freigegeben. Dies geschieht über den *poke()*-Operator.

```
function updateGenDSP(){
    switch(filter_data[filter_selected].Type){
        case "Highpass":
            gen_filter_data.poke(1, filter_selected, 1);
            break;
        case "Lowpass":
            gen_filter_data.poke(1, filter_selected, 2);
```

```
8               break;
9           ...
10      }
11      gen_filter_data.poke(2, filter_selected, filter_data[filter_selected].
            Freq);
12      ...
13      gen_filter_update.poke(1, 0, 1);
14  }
```

Quellcode 4.14: Übertragung der Parameter in Gen

Für die visuelle Darstellung der Filterkurven, den Gitterlinien und den Knotenpunkten wurde die MGraphics-API [81] in JavaScript genutzt. Diese ist neben der Sketch-API, die auf Befehlssätzen der Open Graphics Library (OpenGL) basiert, eine der beiden Varianten zur Darstellung von Grafiken mithilfe des *jsui*-Objekts. Hierfür war zur Initialisierung der Code der Zeilen 1-3 in Quellcode 4.15 notwendig. Über das Setzen von *mgraphics.relative_coords = 0* wurden die absoluten Pixelkoordinaten des *jsui*-Objekts genutzt. Diese Koordinaten werden auf die Filterparameter so umgerechnet, dass die x-Achse die Frequenz in Hertz und die y-Achse die Verstärkung in Dezibel bzw. den Gütefaktor Q widerspiegeln.

```
1   mgraphics.init();
2   mgraphics.relative_coords = 0;
3   mgraphics.autofill = 0;
4   ...
5   function paint(){
6       with(mgraphics){
7           set_line_width(thickness_eq_curve);
8           draw_color = isActive ? rgba_eq_curve_active :
                rgba_eq_curve_inactive;
9           set_source_rgba(draw_color);
10
11          for(var i = 0; i < filter_curve_freqs.length; i++){
12              if(i == 0){
13                  move_to(0.5, filter_curve_points[i] + 0.5);
14              } else {
15                  line_to(i + 0.5, filter_curve_points[i] + 0.5);
16              }
17          }
18          stroke();
19      ...
20      }
21  }
```

Quellcode 4.15: Zeichnen der EQ-Kurven

Da die Frequenzachse in EQs logarithmisch dargestellt wird, war hierfür ein Algorithmus zur logarithmischen Skalierung notwendig. Dieser basiert in dieser Implementierung auf

der Umstellung des Verhältnisses

$$\frac{x - x_{min}}{x_{max} - x_{min}} = \frac{\log y - \log y_{min}}{\log y_{max} - \log y_{min}} \tag{4.1}$$

nach den jeweiligen Variablen x bzw. y. Die *paint()*-Funktion wird sowohl bei der Initialisierung des Plugins als auch bei Parameteränderungen über die Funktion *mgraphics.redraw()* aufgerufen. Sie kann dann durch die Integration OpenGL-ähnlicher Befehle die Filterkurven als Vektorgrafik darstellen. Die grafische Darstellung der Kurven wird über die stückweise Aneinanderreihung kurzer geradliniger Segmente realisiert. Die x-Werte der Pixelkoordinaten werden in Frequenzwerte umgerechnet und die Betragsfunktionen der Filter an diesen Frequenzen in Dezibel kalkuliert. Dann werden die Liniensegmente auf Basis der auf die y-Achse skalierten Betragswerte gebildet. Über das in Abbildung 4.5 gezeigte *live.colors*-Objekt wird das eingestellte Farbschema aus der Live API abgefragt und die Filterkurven entsprechend gefärbt. Die Variable *isActive* sorgt für die Unterscheidung, ob sich das Plugin in Ableton Live im Bypass-Modus befindet oder nicht. Im Bypass-Modus werden die GUI-Objekte ausgegraut, was der visuellen Symbolisierung der Inaktivität des Plugins dient.

Da die Verarbeitung des Codes in *jsui*-Objekten im Prozess niedriger Priorität in Max erfolgt, war eine Optimierung des Programmcodes zur Einsparung an CPU-Ressourcen notwendig, um eine möglichst flüssige Darstellung zu gewährleisten. Statt alle Werte bei jedem Redrawing neu zu berechnen, wurden Arrays angelegt, die nicht-variable Werte wie die Frequenzwerte der x-Achse, die für die Berechnung der Filterkurven notwendig sind, bereits speichern. Die Berechnung der Werte erfolgt einmalig bei der Initialisierung des Plugins. Die Gitterlinien müssen ebenfalls nur einmalig gezeichnet werden, weshalb deren Berechnung in eine separate Datei *DRS_UI_EQ_grid_jsui.js* und in ein zusätzliches *jsui*-Objekt ausgelagert wurde. Die visuelle Transparenz dieser Objekte über die Einstellung des Alphakanals im RGBA-Format ermöglicht dann ein Übereinanderlegen der Filterkurven und Gitterlinien.

Animation der spektralen Modifikationen

Für die visuelle Nachvollziehbarkeit werden die spektralen Modifikationen auf dem GUI in Echtzeit animiert. Dies erfolgt über ein drittes *jsui*-Objekt und ebenfalls unter Nutzung der MGraphics-API. Die darzustellenden Gain-Reduction-Werte pro Frame werden in einem zusätzlichen *buffer~*-Objekt *—BUFFER_draw_reduction_filters* gespeichert, durch das *jsui*-Objekt im Code aufgegriffen und die Koordinaten der Segmente ähnlich zu den Filterkurven anhand der Bin-Frequenzen berechnet. Jedoch werden die Segmente nicht linear, sondern mithilfe von Catmull-Rom-Splines interpoliert. Dies dient einer visuellen Anpassung an das Design moderner Audio-Plugins. Da die MGraphics-API nur Funktionen zur Berechnung von Bézierkurven besitzt, erfolgt im Code eine Transformation der

Punktkoordinaten als Catmull-Rom- zu Bézier-Koordinaten nach den in [82] gegebenen Berechnungsvorschriften.

5 Test des Audio-Plugins in praktischen Applikationen

Die folgenden Abschnitte dienen dem Test des Plugins anhand von Messungen und Hörbeispielen und geben Aufschluss über mögliche Anwendungsszenarien in der Praxis. Die Funktion des Plugins wurde zuerst mithilfe geeigneter Messungen geprüft. Im Anschluss werden die Hörbeispiele als Anwendungsbeispiele herangezogen, anhand deren der auditive Effekt nachvollzogen werden kann.

Wenn Sie die gedruckte Variante dieser Arbeit nutzen, finden Sie die Plugin-Dateien, Hörbeispiele und Messdaten auf dem beiliegenden Datenträger im Ordner ***BA_52883_Kendy-Drechsler_Plugin_und_Hörbeispiele****. Bei Nutzung der PDF-Variante finden Sie die Daten unter folgendem Link:*

https://drive.google.com/drive/folders/1L_3FEI_cS7T_nnMijYoaTGeGGWxFjr2E?usp=sharing

5.1 Übersicht der Plugin-Parameter

Abbildung 5.1: Darstellung des finalen Plugins in Ableton Live (Eigene Darstellung)

Abbildung 5.1 zeigt die Anordnung der Parameter auf dem GUI. Diese wurden für einen entsprechenden Workflow von links nach rechts angeordnet. Dabei entspricht die Reihenfolge der Parameter nicht zwingend dem exakten Signalfluss der Implementierung. Dies ist damit zu begründen, dass für eine auditive und visuelle Wirkung einzelner Parameter erst die Einstellung anderer Parameter nötig ist. So ist es z.B. sinnvoll, bei der Bearbeitung breitbandiger Signale zuerst den Threshold bis auf ein Niveau herabzusenken, wo erste Ausschläge der Gain Reduction auf dem GUI sichtbar sind. Die erst anschließend eingestellte Frequenzbewertung mittels EQ und Bewertungskurven sowie die Feinjustierung des Thresholds ermöglichen dann die Fokussierung des Filtereffekts auf bestimmte Frequenzbereiche. Die Ausrichtung nach der Leserichtung von links nach rechts dient in diesem Zusammenhang einer intuitiven Bedienbarkeit. Eine vollständige

Übersicht der Plugin-Parameter als Bedienungsanleitung wird in Anhang B bereitgestellt.

5.2 Funktionstest des Plugins

Geeignete Testsignale sollten zunächst die Funktion des Plugins prüfen und den Effekt der spektralen Modifikationen einordnen. Dies ist für die Zielstellung dieser Arbeit nötig, da somit messtechnisch analysiert werden kann, inwiefern sich die in Max implementierte WOLA-Methode für den Filtereffekt eignet. Es bot sich deshalb eine Analyse im Zeit- und Frequenzbereich an.

Da die Amplitudenerkennung der einzelnen Frequency Bins in dem Algorithmus nicht voneinander abhängen, war ein Test anhand einer idealisierten Sinusfunktion sinnvoll. Es stellte sich heraus, dass die in Ableton Live gemessene Ausgangsamplitude einer einzelnen Sinusfunktion als Testsignal nicht mit der auf dem GUI gezeigten spektralen Modifikation korreliert. Dies bot Anlass, eine nähere Untersuchung des Effekts durchzuführen, da eine Revision des Codes zu keiner Fehlererkennung der programmierten mathematischen Beziehungen führte. Auch eine Rückführung auf ein Minimalbeispiel ohne Frequenzbewertungen und Glättung zeigten keine Verbesserungen. Eine zum Test durchgeführte Simulation in MATLAB hingegen zeigte die erwartete Rekonstruktion des idealen Sinustons. Auch eine Multiplikation aller Frequenzamplituden mit einem skalaren Faktor führt zu einem exakten Ergebnis in Max. Ein typisches Peak-Filter, z.B. mittels EQ in Ableton Live eingestellt, zeigt die gleiche Performance. Zudem waren mit sinkendem Schwellwert zunehmend Klickgeräusche am Beginn der Sinuskurve hörbar. Es wurde deshalb geprüft, inwiefern diese Phänomene in Verbindung mit der von der Bin abhängigen spektralen Modifikation mit den Parametern der STFT in Max zusammenhängen.

Für diesen Test wurden Faktoren im Vorfeld erörtert, welche die Ausgabeergebnisse beeinflussen könnten. Zum einen stellt die implementierte STFT einen zeitvariablen Prozess dar. Dies bedeutet, dass für nicht-monofrequente Signale jedes sukzessive Frame ein anderes Spektrum ausgibt, auf Basis dessen die spektralen Modifikationen gebildet werden. Um jedoch ein messbares Ergebnis anhand der Amplituden erhalten zu können, ist ein monofrequentes Testsignal besser geeignet, da dessen Spektrum, abgesehen von Quantisierungsfehlern, über die Zeit nicht schwangt. Zum anderen gilt es, den Leck-Effekt bei Messungen im Frequenzbereich zu berücksichtigen. Die Wahl einer Frequenz bei einem entsprechenden DFT-Rasterpunkt sollte dagegen Idealbedingungen schaffen, wonach der Leck-Effekt aufgrund der in Abschnitt 3.1.4 beschriebenen Verhältnisse nicht auftreten sollte.

5.2.1 Durchführung der Einzeltonmessung

Für die Messungen wurden zwei Tonspuren in Ableton Live angelegt. Als Abtastrate wurde $f_A = 44100\,\mathrm{Hz}$ gewählt. Die Testsignale wurden in MATLAB nach dem Waveform Audio File Format (WAVE) als *.wav*-Dateien generiert. Die erste Spur enthielt das gewählte Testsignal. Auf dieser Spur befand sich nur das Audio-Plugin als Effekt. Das verarbeitete Signal wurde mittels Resampling auf die zweite Spur aufgenommen. Die aufgenommenen Audio-Dateien wurden für eine präzise Analyse im Zeitbereich in MATLAB als WAVE-Dateien exportiert. Die Analyse der Amplituden fand dagegen mithilfe des integrierten Peak-Meters in Ableton Live statt.

Für die Analyse der Sinusfunktionen wurden folgende Parameter konstant eingestellt:
- Ratio: 64 : 1
- Knee: 0 dB
- Ac. Weighting: Flat
- EQ: alle Filter aus
- Resolution: Medium
- Attack: 0 ms
- Release: 0 ms
- Dry/Wet: 100 %
- Output Trim: 0.00 dB

Dies sollte eine von Frequenzbewertung und Glättung unabhängige Bewertung der tatsächlichen Amplituden ermöglichen. Die hier höchste Ratio sollte zu einer entsprechend starken Modifikation bzw. nahezu Clipping der Amplitude führen. Bei einem eingestellten Threshold von -24 dB bedeutet dies, dass eine Sinusfunktion, deren Frequenz exakt der Frequenz eines DFT-Rasterpunkts entspricht, nach der Resynthese unter idealen Bedingungen eine Amplitude von $-24\,\mathrm{dB} + 24/64\,\mathrm{dB} = -23.625\,\mathrm{dB}$ aufweisen muss. Der Schwellwert wurde von 0 dB bis -24 dB in 6dB-Schritten abgesenkt. Die Messungen wurden für je vier verschiedene FFT-Längen und Fensterfunktionen durchgeführt. Die Spitzenwerte wurden nach Ende der wahrgenommenen Transiente abgenommen, da das monofrequente Signal dann als stationär zu betrachten war. Dementsprechend waren die Werte auch als unabhängig vom Overlap-Faktor anzunehmen. Dieser wurde als konstant 8 eingestellt. Um eine mögliche Frequenzabhängigkeit der realen Amplituden zu prüfen, wurden die Messungen an Frequenzen für zwei entfernte DFT-Rasterpunkte durchgeführt.

5.2.2 Auswertung der Einzeltonmessung

Messung der Amplituden

In Tabelle 5.1 sind die real gemessenen Amplituden nach der Resynthese den Ideal-

werten gegenübergestellt. Die Idealwerte entsprechen dem eingestellten Schwellwert, verrechnet mit der angenommenen Ratio. Die Frequenzen f wurden so gewählt, sodass sie einen Rasterpunkt für alle gewählten FFT-Längen darstellen. Die Tabelle zeigt, dass

Tabelle 5.1: Real gemessene Amplitudenwerte nach der spektralen Modifikation mit einer Ratio von 64:1 für verschiedene Fenstertypen, FFT-Längen N und Frequenzen an DFT-Rasterpunkten. Zur einheitlichen Darstellung wurden die Werte auf zwei Nachkommastellen gerundet

Parameter		**gem. Amp. bei f = 172.265625 Hz für verschiedene Fensterfunktionen**				**gem. Amp. bei f = 1033.59375 Hz für verschiedene Fensterfunktionen**			
N [Samples]	Ideal [dB]	Hanning [dB]	Rechteck [dB]	Hamming [dB]	Blackman [dB]	Hanning [dB]	Rechteck [dB]	Hamming [dB]	Blackman [dB]
256	0.00	0.00	0.00	-0.01	0.00	0.00	0.00	-0.01	0.00
256	-5.91	-3.43	-3.45	-3.46	-3.72	-3.43	-3.45	-3.46	-4.08
256	-11.81	-7.45	-5.91	-7.19	-7.68	-9.34	-5.91	-8.68	-9.98
256	-17.72	-10.50	-7.49	-10.30	-10.70	-15.20	-7.49	-14.60	-15.90
256	-23.63	-12.60	-8.41	-12.50	-12.70	-21.10	-8.41	-20.50	-21.80
512	0.00	0.00	0.00	-0.02	0.00	0.00	0.00	-0.02	0.00
512	-5.91	-3.45	-3.46	-3.47	-4.09	-3.45	-3.46	-3.47	-4.09
512	-11.81	-9.35	-5.93	-8.69	-10.00	-9.35	-5.93	-8.69	-10.00
512	-17.72	-15.30	-7.51	-14.60	-15.90	-15.30	-7.51	-14.60	-15.90
512	-23.63	-21.20	-8.44	-20.50	-21.80	-21.20	-8.44	-20.50	-21.80
1024	0.00	0.00	0.00	-0.03	0.00	0.00	0.00	-0.03	0.00
1024	-5.91	-3.46	-3.46	-3.48	-4.10	-3.46	-3.46	-3.48	-4.10
1024	-11.81	-9.35	-5.94	-8.69	-10.00	-9.35	-5.94	-8.69	-10.00
1024	-17.72	-15.30	-7.52	-14.60	-15.90	-15.30	-7.52	-14.60	-15.90
1024	-23.63	-21.20	-8.45	-20.50	-21.80	-21.20	-8.45	-20.50	-21.80
2048	0.00	0.00	0.00	-0.03	0.00	0.00	0.00	-0.03	0.00
2048	-5.91	-3.46	-3.46	-3.49	-4.10	-3.46	-3.46	-3.49	-4.10
2048	-11.81	-9.36	-5.94	-8.70	-10.00	-9.36	-5.94	-8.70	-10.00
2048	-17.72	-15.30	-7.53	-14.60	-15.90	-15.30	-7.53	-14.60	-15.90
2048	-23.63	-21.20	-8.46	-20.50	-21.80	-21.20	-8.46	-20.50	-21.80

die gemessenen Amplituden selbst bei idealen Eigenschaften des Eingangssignals stark vom erwarteten Idealwert abweichen. Beim Vergleich der Werte fällt eine deutliche Abhängigkeit von dem gewählten Analysefenster auf. Während der Unterschied bei geringen Schwellwertsabsenkungen zwischen den Fensterfunktionen nur gering ist, so ist er für geringe Idealwerte umso größer. Das Rechteck-Fenster zeigt dabei die größten Abweichungen vom Idealwert. Hanning- und Blackman-Fenster hingegen zeigen hingegen eine ähnlich gute Annäherung an den Idealwert. Gleichzeitig sind die Abstände zwischen den Werten eines Fensters bei sinkendem Idealwert nicht-linear. Eine Frequenzabhängigkeit ist jedoch nur für die kleinste gewählte Fensterlänge erkennbar. Gleichzeitig ist eine eindeutige Abhängigkeit von der FFT-Länge nur für die kleinste Frequenz mit Ausnahme von Messabweichungen sichtbar.

Eine exakte Erklärung des Phänomens ist hier nicht möglich, da nicht bekannt ist, wie das in dieser Arbeit genutzte *pfft~*-Objekt in Max implementiert ist. Es ist naheliegend, dass es beim Processing im Spektralbereich zu Rundungsfehlern kommen kann, welche eine exakte Rekonstruktion des Signals nicht zulassen. Demnach ist es möglich, dass die Amplituden der Frequency Bins selbst bei Analyse einer Frequenz an DFT-Rasterpunkten nicht ganz 0 sind und deshalb vom Algorithmus nicht als *störend* erkannt werden. Im Umkehrschluss führe die Resynthese des Signals aufgrund der übrigen Signalenergie nicht zum erwarteten Ergebnis. Auch die gewählte Fensterfunktion und das Eingangssignal können von diesen Rundungsfehlern in Max betroffen sein. So zeigt das Blackman-Fenster, welches beim Vergleich mit 3.1 die höchste Side-Lobe-Absenkung der analysierten Fenster besitzt, die beste Annäherung an den Idealwert.

Messung der Transienten

Die Abbildungen 5.2 und 5.3 zeigen das wahrgenommene und hier analysierte Einschwingverhalten der vorher aufgenommenen Sinusschwingungen.

Das gleiche Prinzip lässt sich auf das Ausschwingen am Ende des Sinus übertragen. Die ideale Sinuskurve, welche durch das Plugin verarbeitet wurde, startete exakt bei Sekunde 1. Es ist ein deutliches Einschwingen zu erkennen, was durch das Processing verursacht wird. In Abbildung 5.2 wird deutlich, dass mit Ausnahme des Rechteckfensters alle gewählten Analysefenster ein ähnliches Einschwingverhalten zeigen. Der starke Unterschied der sichtbaren Amplituden zwischen dem Rechteckfenster und anderen Fensterfunktionen ist auf die in Tabelle 5.1 gemessenen Werte zurückzuführen. Dagegen verlängert sich die Einschwingkurve deutlich mit Zunahme der FFT-Länge.

Das Einschwingverhalten zeigt jedoch keine Abhängigkeit von den gewählten Frequenzen, da die Signale der oberen Darstellung in Abbildung 5.3 bei Betrachtung die gleiche Hüllkurve besitzen. Mit sinkenden Schwellwerten und damit einer deutlicheren spektralen Modifikation erhöht sich jedoch der maximale Ausschlag der Einschwingkurve.

Das Verhalten der Kurven lässt sich mit der Funktionsweise des WOLA-Verfahrens erklären. Da sich die Frames bei der Frequenzanalyse überlappen, liegen im Bereich des Starts der originalen Sinuskurve unterschiedlich hohe Anteile dieser in jedem Frame. Dies führt dazu, dass die gemessenen Amplituden in jedem Frame unterschiedlich hoch ausgeprägt sind. Diese fallen jedoch erst ab einem gewissen Wert über den Schwellwert. Dies bedeutet, dass ein reiner Ton am Eingang bei plötzlichem Einschalten erst nach und nach in seiner Amplitude reduziert wird. Die FFT-Länge bestimmt dabei die zeitliche Auflösung der Resynthese. Je größer diese gewählt wurde, desto länger und ausgeprägter ist das hier dargestellte Einschwingen, wobei gleichzeitig die spektrale Auflösung steigt. Das Verhalten ist ähnlich zu linearphasigen FIR-Filtern, welche mit steigender Filterord-

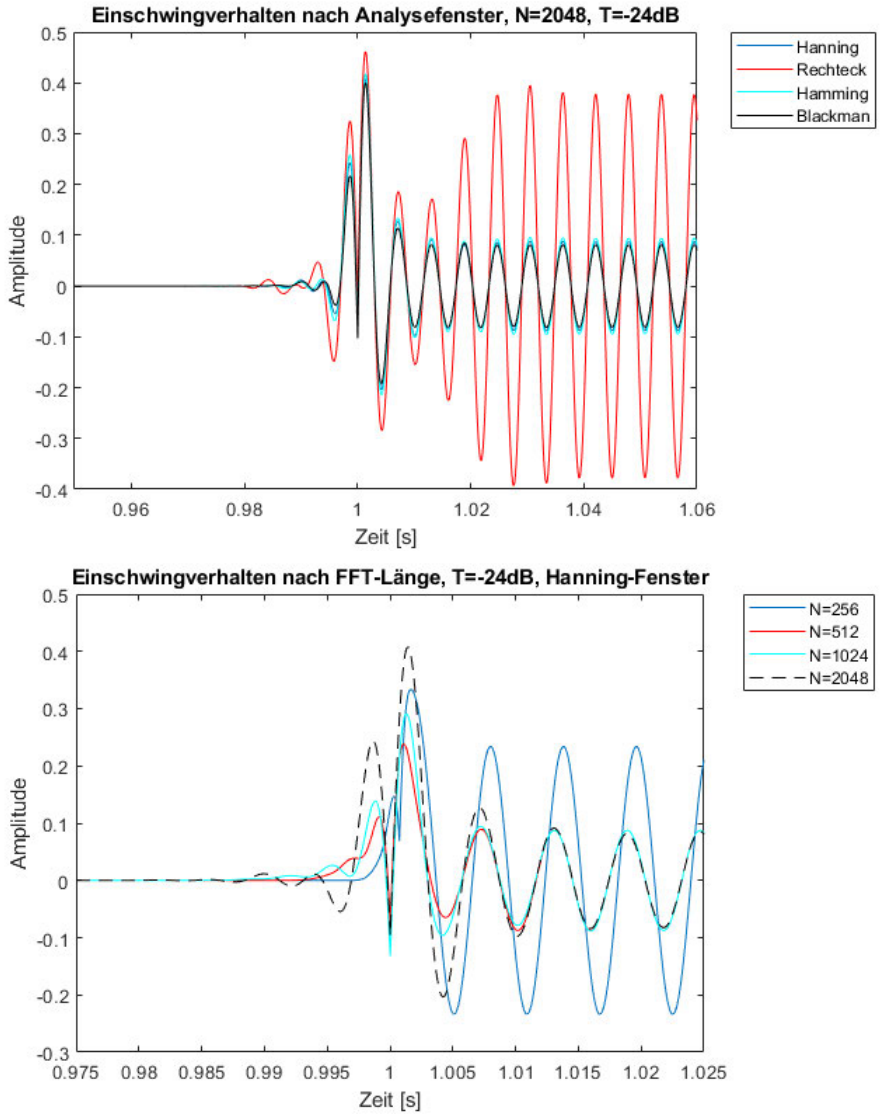


Abbildung 5.2: Darstellung des Einschwingverhaltens des Plugins, abhängig von der Fensterfunktion und FFT-Länge (Eigene Darstellung)

nung schmalbandigere Filter realisieren können, dafür aber ein deutliches Einschwingen nach sich ziehen. Ein Einfluss des Overlap-Faktors konnte beim Test jedoch nicht festgestellt werden und wird deshalb hier nicht visualisiert.

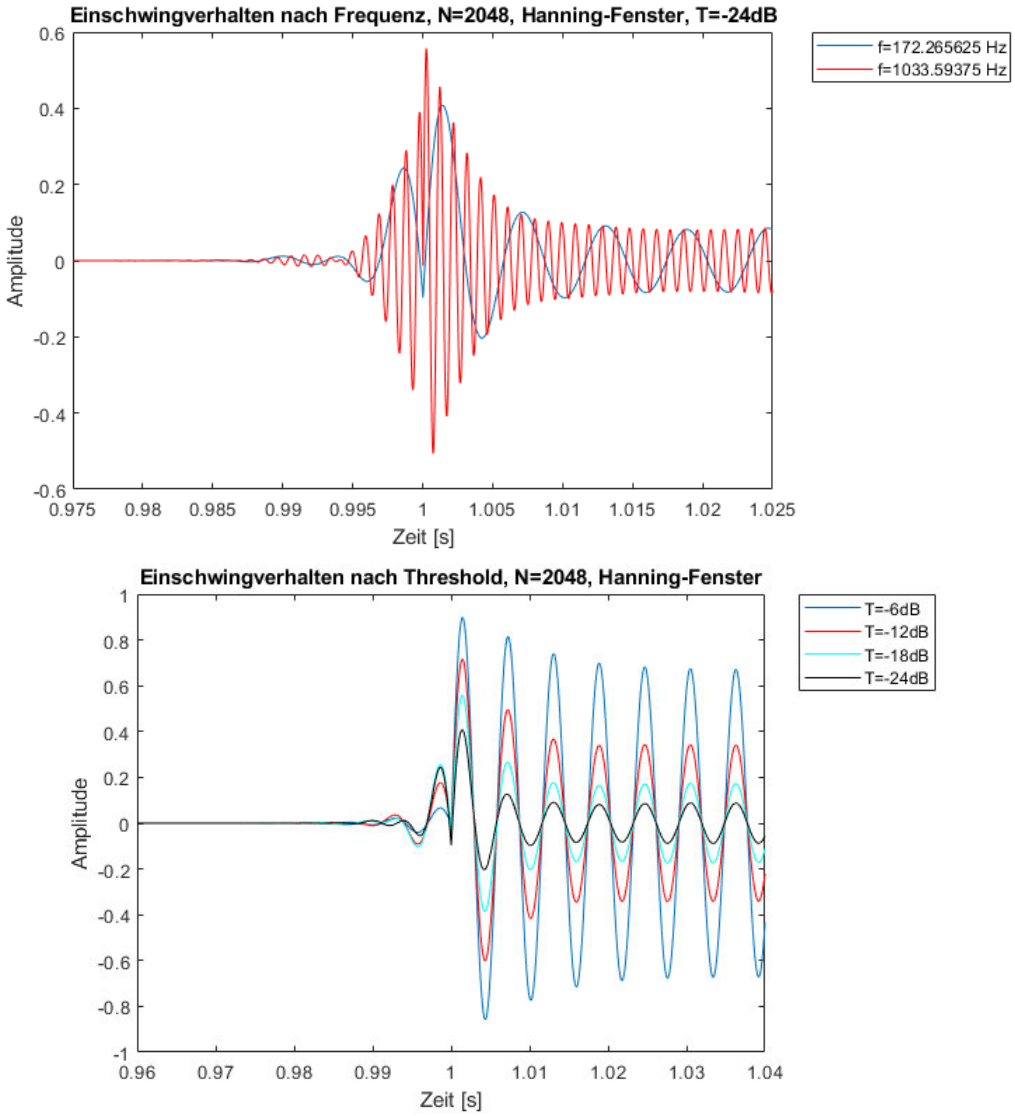


Abbildung 5.3: Darstellung des Einschwingverhaltens des Plugins, abhängig von der Frequenz des Signals und dem Schwellwert (Eigene Darstellung)

5.2.3 CPU-Last des Plugins

Ableton Live bietet zur Bestimmung der CPU-Last eine integrierte Metering-Option. Da die STFT und die Berechnungen im Spektralbereich hier den größten Anteil der Rechenschritte ausmachen, wurde die CPU-Last unter Berücksichtigung der FFT-Länge und dem Overlap-Faktor in Tabelle 5.2 ausgewertet. Diese Zahlen sind jedoch in hohem Maße abhängig vom Gesamtsystem des Anwenders. Hier wurden die Werte unter Nutzung des CPU-Modells Intel Core i7-8750H aufgenommen, wobei in Ableton Live eine Grundlast von 2% gemessen wurde. Erkennbar ist ein deutlicher Anstieg der Last unter Erhöhung des Overlap-Faktors, auf dem GUI als *Resolution* bezeichnet. Dies ist auf die Anzahl der zu berechnenden FFTs pro Zeitabschnitt zurückzuführen. Die spektralen Berechnungen pro Bin in jedem Frame scheinen jedoch nur einen geringfügigen Einfluss mit steigenden FFT-Längen zu haben, obwohl die Anzahl der Bins ebenfalls stark wächst.

Tabelle 5.2: CPU-Last bei Nutzung des Plugins in Ableton Live

	FFT-Länge in Samples			
Overlap-Faktor	256	512	1024	2048
4	6%	6%	6%	6%
8	8%	8%	8%	8%
16	11%	11%	12%	12%
32	18%	18%	18%	19%

5.3 Hörbeispiele und Messung anwendungsbezogener Kenngrößen

Die erläuterten Hörbeispiele sollen Aufschluss über die klanglichen Eigenschaften des Plugins geben. Die rohen WAVE-Dateien wurden unter Anwendung des Plugins nach den hier beschriebenen Kriterien auf einer Tonspur in Ableton Live bearbeitet. Die Ergebnisse wurden über die *Resampling*-Option in Ableton Live für die weitere Analyse in der Software MATLAB aufbereitet.

Um eine messtechnische Vergleichbarkeit zwischen den Hörbeispielen zu schaffen, musste eine standardisierte Parametrisierung der Plugin-Parameter für jedes Beispiel gefunden werden. Die gebotenen Einstellungen wurden so getroffen, sodass das Audiomaterial nicht stark verfremdet wurde, ein Effekt jedoch hörbar und ein messtechnischer Vergleich möglich ist. Es sei anzumerken, dass die Parameter in der Praxis bei bestimmten Applikationen stark variieren, je nach Abhörsituation individuell gewählt und die hör- und messbaren Effekte demnach extremer oder subtiler ausfallen können. Dazukommen letztendlich die hohe Subjektivität des Anwenders sowie individuelle Anforderungen und ggf. Kreativität in der Audiopraxis, welche eine eineindeutige Parametrisierung nicht zulassen. Die hier beschriebenen Beispiele dienen demnach als Anhaltspunkt für praktische Applikationen.

Eine Kategorisierung der Hörbeispiele nach "klingt gut/besseröder "klingt schlecht/schlechter"kann mit den hier beschriebenen Tests jedoch nicht erfolgen. Dies würde umfassende psychoakustische Untersuchungen unter festgelegten Bedingungen und unter Einbezug von Probanden erfordern, um eine Aussage treffen zu können.

5.3.1 Anwendungsfall 1: Ausgleich der spezifischen Lautheit

Im Folgenden sollte anhand des Hörbeispiels geprüft werden, ob sich das Plugin zur Reduktion der als störend empfundenen Frequenzamplituden eignet, wobei sich der Begriff *störend* in dieser Applikation auf als *zu laut* wahrgenommene Frequenzanteile in einer Studioanwendung bezieht. Bei dem zu analysierenden Hörbeispiel handelt es sich um eine kurze Instrumentalsequenz, welche für die Weiterverwendung in der Postproduktion

simulativ aufbereitet werden soll.

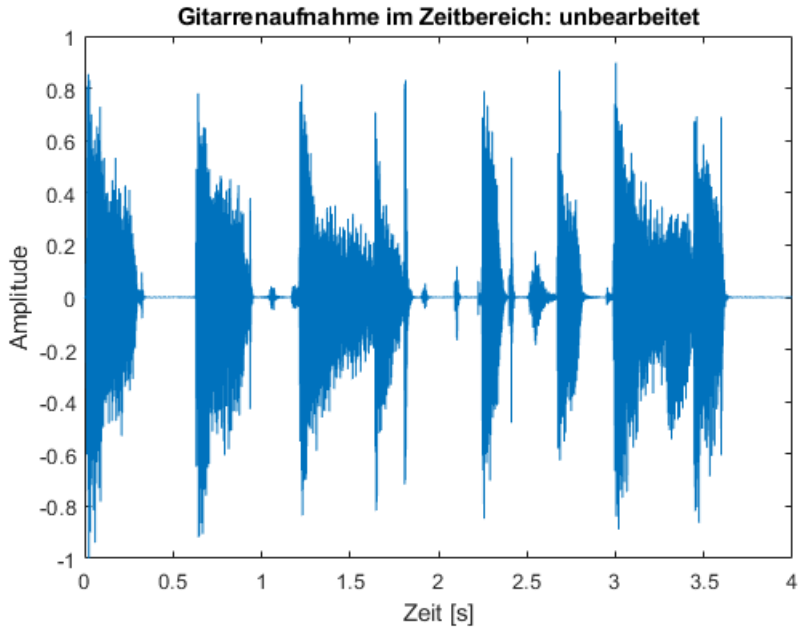


Abbildung 5.4: Darstellung der Gitarrensequenz im Zeitbereich (Eigene Darstellung)

Um den Charakter der tatsächlichen *Wahrnehmung* zu berücksichtigen, ist eine Spektralanalyse mittels Spektrogramm für die Untersuchung des psychoakustischen Effekts unzureichend, da sie nicht die Wahrnehmung des Menschen berücksichtigt. Stattdessen eignet sich hierfür die spezifische Lautheit als Indikatorgröße für die frequenzbandabhängige Wahrnehmung der Lautheit. Es sollte messtechnisch gezeigt werden, ob und wie sich diese unter Anwendung des Plugins durch die spektralen Modifikationen ändert. Hierfür eigneten sich die in MATLAB nutzbaren Funktionen zur Untersuchung. Die spezifische Lautheit der Gitarrensequenz kann aufgrund des instationären Charakters des Signals über die Zeit stark variieren. Die in DIN 45631/A1:2010-03 [63] verankerte und in MATLAB implementierte Methodik eignete sich zur Analyse, da sie die zeitliche Komponente der Lautheit berücksichtigt.

Abbildung 5.4 dient im Folgenden dem Vergleich mit den Messungen als Referenz. Erkennbar sind die Saitenanschläge im Zeitbereich an den deutlichen Transienten. Als Vorbereitung wurde das Signal mithilfe der in der jeweils oberen Darstellung in Abbildung 5.5 gezeigten Analyse auf die enthaltenen Frequenzanteile sowie die Lautheit und spezifische Lautheit überprüft. Es ist erkennbar, dass das unbearbeitete Signal prominente Amplituden im Bereich von 0 bis 5 kHz aufweist. Die Messergebnisse der spezifischen Lautheit zeigen dagegen, dass die Frequenzbänder zwischen etwa 14 und 17 Bark (2320 bis 3700 Hz) besonders zum Lautheitseindruck beitragen. Es lässt sich schlussfolgern, dass die Frequenzgruppen oberhalb und unterhalb als weniger laut wahrgenommen werden. Dies verdeutlicht die Notwendigkeit der implementierten Frequenzbewertungsmöglichkeiten. Da der Algorithmus diese Faktoren ohne einer Frequenzbewertung nicht be-

rücksichtigt, kann er mittels EQ in diesem Anwendungsfall stattdessen auf die Frequenzbereiche fokussiert werden, welche vom Anwender als *zu laut* wahrgenommen werden.

Das Signal wurde deshalb für einen qualitativen Vergleich in zwei Ausführungen bearbeitet. Die erste Version wurde ohne eingestellter Frequenzbewertungskurve erstellt, die zweite mit einem Hochpassfilter vierter Ordnung als individuelle Frequenzbewertungskurve mit einer Grenzfrequenz von 2 kHz. Für beide Varianten wurden folgende Standardparameter zum messtechnischen Vergleich gewählt:

- Threshold: -36 dB
- Ratio: 10 : 1
- Knee: 6 dB
- Ac. Weighting: Flat
- Block Size: 2048
- Resolution: Medium
- Window Function: Hanning
- Envelope: Log
- Attack: 50 ms
- Release: 500 ms
- Dry/Wet: 100 %
- Output Trim: 0.00 dB

Die zeitliche Glättung durch Attack und Release dienen der Vorbeugung von hörbaren Artefakten, verursacht durch abrupte Änderungen der Werte einzelner Frames.

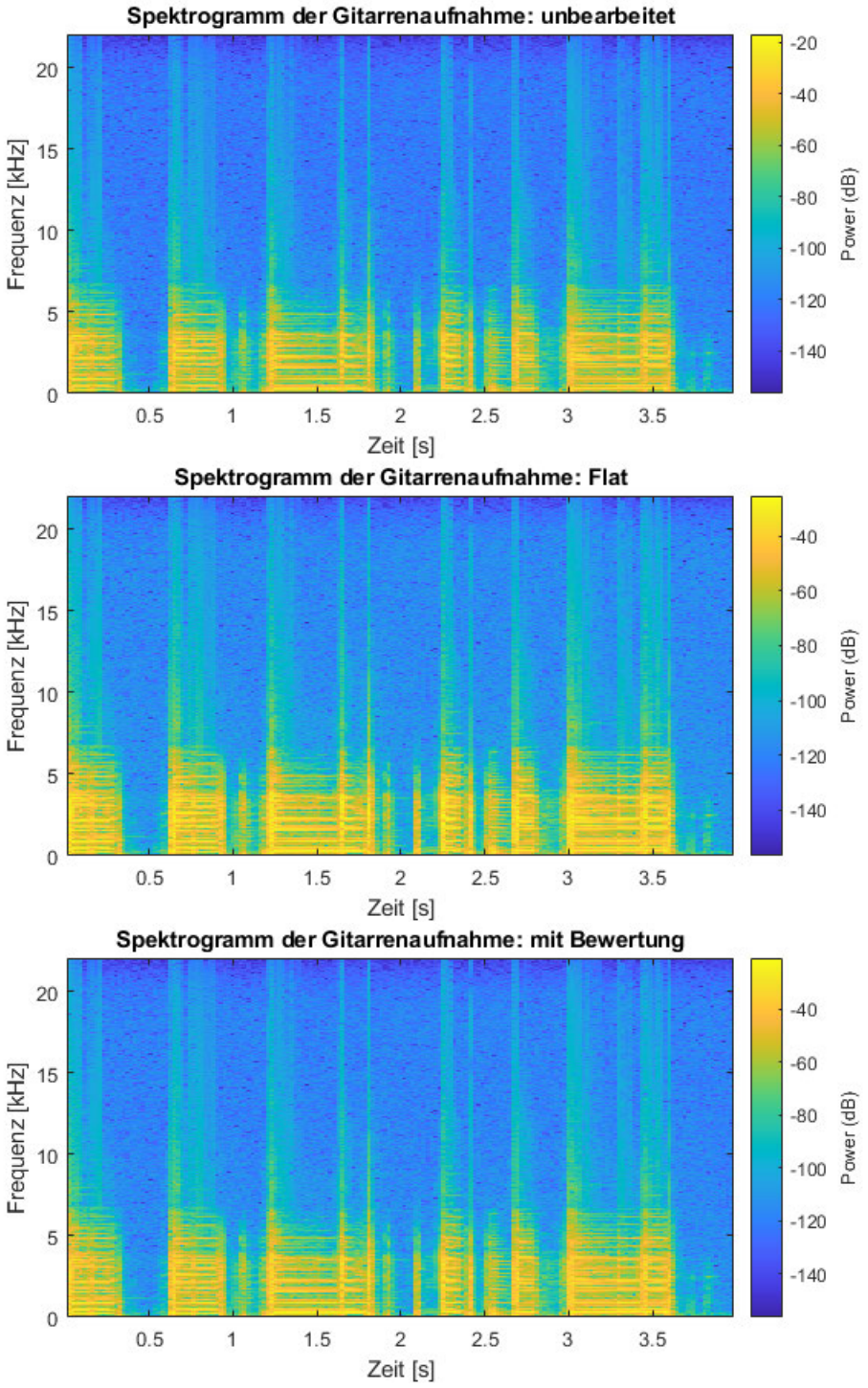


Abbildung 5.5: Vergleich der Spektrogramme der Gitarrenaufnahme. Oben ist die unbearbeitete Aufnahme. Die Aufnahme in der Mitte wurde ohne eingestellter Frequenzbewertung bearbeitet, die untere Aufnahme hingegen mit einem Hochpassfilter 4. Ordnung bei 2 kHz als Frequenzbewertungskurve. (Eigene Darstellung)

Zum auditiven Verständnis des Hörbeispiels werden dem Leser die folgenden drei WAVE-Dateien zur Verfügung gestellt:

1) *BA_Drechsler_HBsp_SpezLautheit_1_Rohaufnahme.wav*
2) *BA_Drechsler_HBsp_SpezLautheit_1_Flat.wav*
3) *BA_Drechsler_HBsp_SpezLautheit_1_CustomWeighting.wav*

Datei 1 beinhaltet hierbei das unbearbeitete Signal, Datei 2 das mittels Plugin bearbeitete Signal ohne eingestellter Frequenzbewertung und Datei 3 das bearbeitete Signal mit mittels EQ eingestellter Frequenzbewertung. Die Rohaufnahme stammt aus [83].

Nach der Audiobearbeitung war in beiden Fällen ein Pegelverlust von etwa 6 dB gegenüber dem Eingangssignal messbar. Für einen Vergleich wurden die Signale in den Abbildungen auf die Vollaussteuerung normalisiert. Abbildung 5.6 ermöglicht einen Vergleich der gemessenen Lautheit der Signale. Gegenüber dem unbearbeiteten Signal ist ein Verlust der Maximallautheit für beide Bearbeitungsmöglichkeiten sichtbar. Im Vergleich zeigt die bearbeitete Variante ohne eingestellter Frequenzbewertungskurve eine leichte Reduktion gegenüber der Variante mit eingestellter Frequenzbewertung. Bei Betrachtung von Abbildung 5.7 fällt unter Beachtung der Skala eine Reduktion der Spitzenwerte der spezifischen Lautheit nach der Bearbeitung auf. Der hellblaue Bereich hingegen bleibt auf der Skala weitestgehend konstant. Es lässt sich ein Kompressionseffekt der spezifischen Lautheit beim Vergleich feststellen. Zudem ist eine Zunahme der spezifischen Lautheit zwischen 0 und 5 Bark (0 bis 510 Hz) bei der bearbeiteten Variante mit individueller Frequenzbewertung zu erkennen. Dagegen nimmt diese bei der Variante ohne Frequenzbewertung in diesem Bereich ab.

Es zeigt sich, dass die Frequenzbewertung der gemessenen Amplituden mittels EQ zur Wahrnehmbarkeit der gefilterten Signalanteile beiträgt. Die spektrale Amplitudenreduktion ist an diesen Stellen etwas geringer. Für praktische Applikationen zeigt das Beispiel einen Nutzen des Plugins beim spektralen Ausbalancieren der wahrgenommenen spezifischen Lautheiten. So war erkenntlich, dass insbesondere tiefe Frequenzen weniger zur Lautheit beitragen als Frequenzen, die schon in den Isophonenkurven aus Abschnitt 3.3.1 als empfindlich für das menschliche Hören wirken. Eine unbewertete Erkennung der als *zu laut* empfundenen Frequenzamplituden ergibt sich daher als wenig sinnvoll. Die Bedeutung der implementierten standardisierten Frequenzbewertungskurven lässt sich demnach vor allem für Situationen ableiten, in denen der anwendende Toningenieur unter Nutzung eines eingemessenen Wiedergabesystems arbeitet. Es ist denkbar, dass das Plugin einen Nutzen z.B. in der Beschränkung der Wahrnehmung von lauten Knallgeräuschen in Kinoproduktionen unter Anwendung der in Abschnitt 3.3.2 beschriebenen Variante der ITU-468-Kurve finden könnte. Hierfür sind weitere Untersuchungen notwendig, um die tatsächliche Effektivität der Frequenzbewertungskurven in tontechnischen

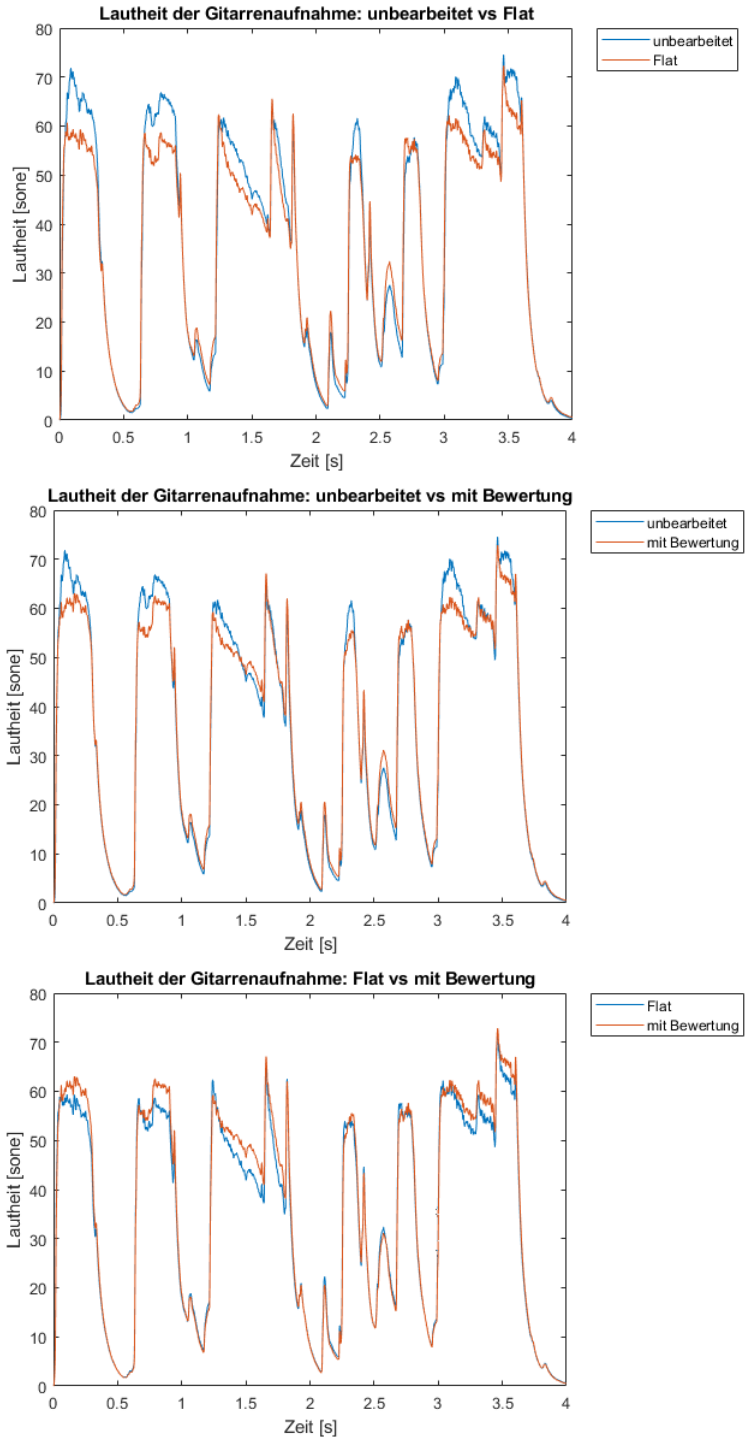


Abbildung 5.6: Vergleich der Lautheiten der Gitarrenaufnahme. Unbearbeitet, mit dem Plugin ohne eingestellter Frequenzbewertung und mit individueller Frequenzbewertung bearbeitet. (Eigene Darstellung)

Applikationen zu überprüfen und erfordert Messungen in spezialisierten Tonstudios.

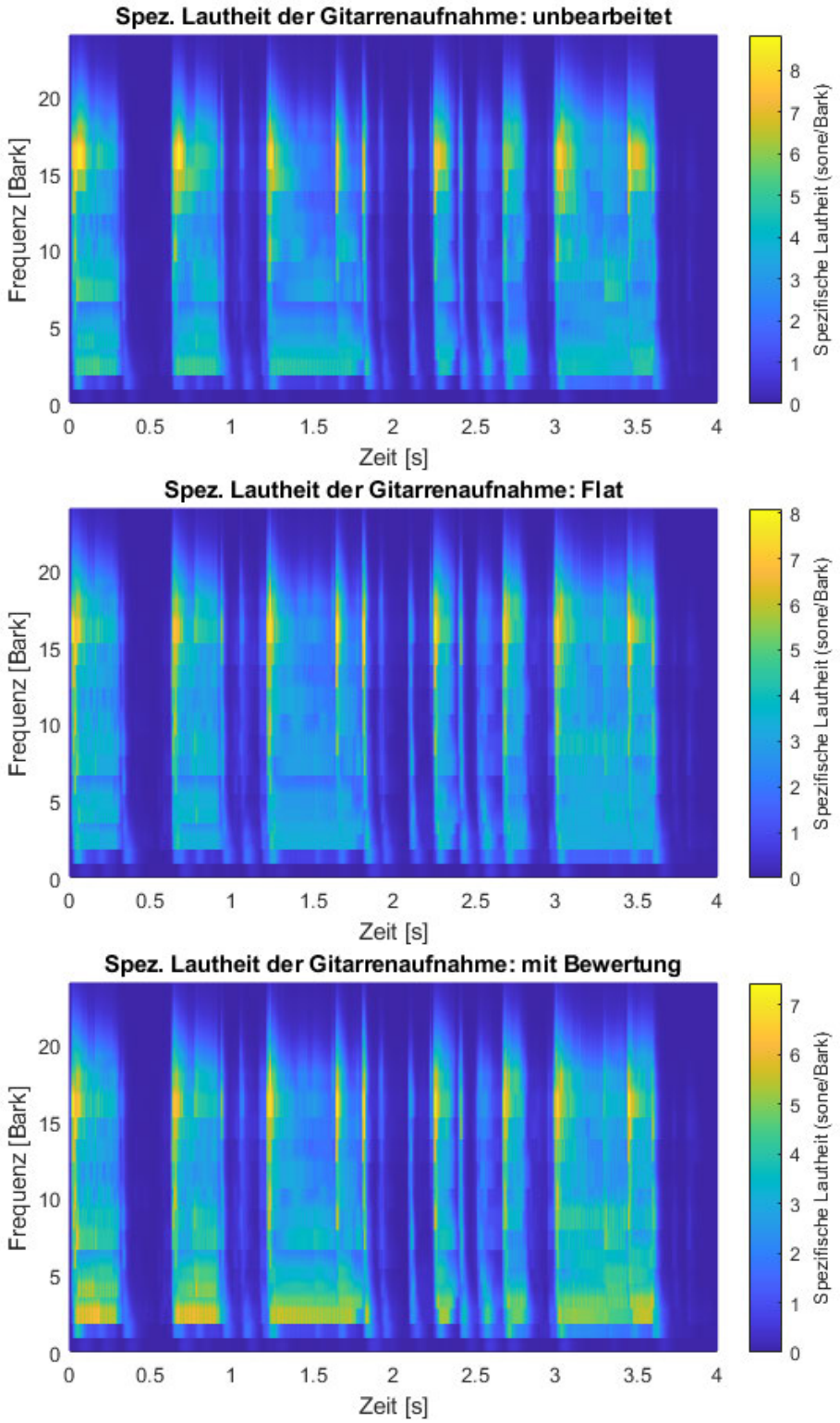


Abbildung 5.7: Vergleich der Spezifischen Lautheiten der Gitarrenaufnahme. Oben ist die unbearbeitete Aufnahme. Die Aufnahme in der Mitte wurde ohne eingestellter Frequenzbewertung bearbeitet, die untere Aufnahme hingegen mit einem Hochpassfilter 4. Ordnung bei 2 kHz als Frequenzbewertungskurve. (Eigene Darstellung)

5.3.2 Anwendungsfall 2: Reduktion der Schärfe in Zischlauten

Das zweite Beispiel zeigt die Applikation des Plugins zur Verringerung der wahrgenommenen Schärfe in Zischlauten. In der Tonstudiopraxis können als *zu scharf* beschriebene Signalanteile, oft auch als *Harshness* bezeichnet, störend wirken. Oft ist dies bei Sprach- und Gesangsaufnahmen der Fall. Im speziellen Fokus sind hier Zischlaute, da sich deren Spektralenergie besonders in den hohen Frequenzen des menschlichen Hörbereichs konzentriert. Da in der Praxis oft Filter in Form von EQs zur gezielten Bearbeitung zum Einsatz kommen, empfahl sich ein Vergleich zwischen dem mess- und hörbaren Effekt des Plugins und dem EQ. Zur Analyse der Schärfe kommt die in MATLAB implementierte Variante aus der DIN 45692 zum Einsatz. Diese berücksichtigt auch die Zeitvariabilität dieser Kenngröße.

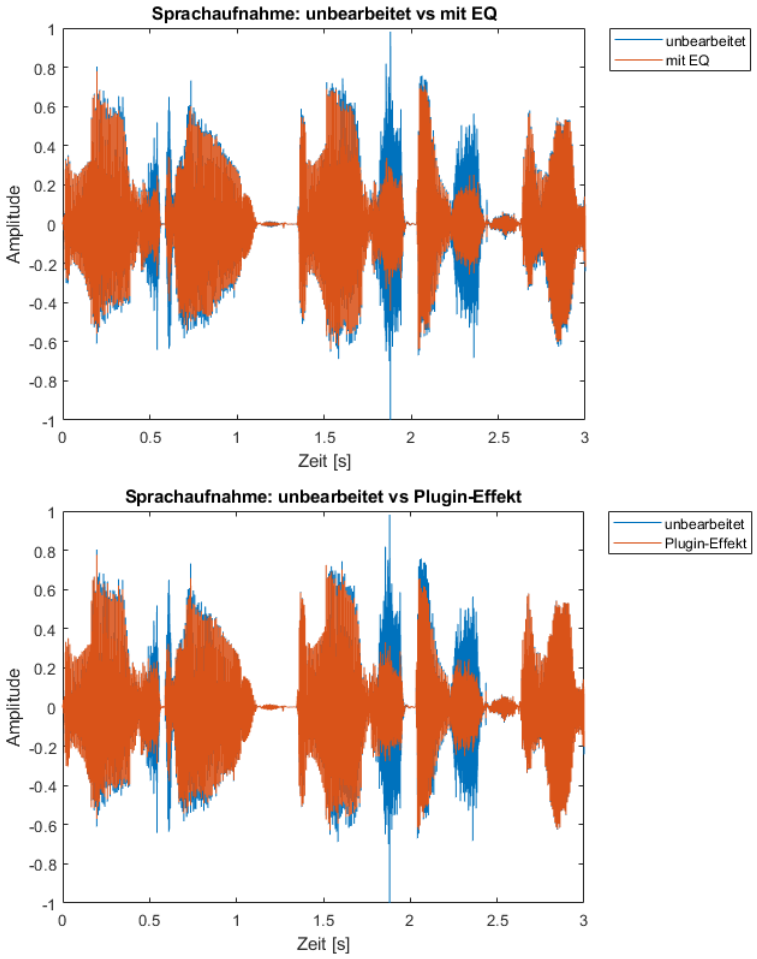


Abbildung 5.8: Vergleich der Zeitdarstellungen der Sprachaufnahme. Oben ist der Vergleich der rohen Aufnahme mit der mittels Peak-Filter (-10 dB Gain bei 7 kHz) bearbeiteten Aufnahme. Unten ist der Vergleich der rohen Aufnahme und der mittels Plugin bearbeiteten Aufnahme. (Eigene Darstellung)

Als Hörbeispiel werden dem Leser auch hier drei WAVE-Dateien zur Verfügung gestellt:

1) *BA_Drechsler_HBsp_Schaerfe_1_Rohaufnahme.wav*

2) *BA_Drechsler_HBsp_Schaerfe_1_EQ.wav*
3) *BA_Drechsler_HBsp_Schaerfe_1_Plugin.wav*

Datei 1 beinhaltet das unbearbeitete Signal, Datei 2 das mittels EQ bearbeitete Signal und Datei 3 das mittels Plugin bearbeitete Signal. Die Rohaufnahme stammt aus [84]. Auch hier erfolgte im Vorfeld eine Spektralanalyse zur Identifikation der Zischlaute im Frequenzbereich. Die obere Darstellung aus Abbildung 5.9 zeigt das Spektrogramm der ersten 3 Sekunden der Aufnahme. Dieses bezieht sich auf die in Englisch gesprochenen Worte *"The mind is time, the mind is space"*. Bei Betrachtung der Spektrogramme fällt auf, dass das Signal in Höhe der Sibilanten prominente Frequenzanteile zwischen 5 und 10 kHz aufweist. Zur Bearbeitung mit dem Ziel der Reduktion der wahrgenommenen Schärfe wurde ein statisches Peak-Filter des in Ableton Live zugänglichen Plugins *EQ Eight* bei 7 kHz und einer Verstärkung von -10 dB eingerichtet. Das entwickelte Plugin hingegen wurde wie folgt parametrisiert.

- Threshold: -36 dB
- Ratio: 10 : 1
- Knee: 6 dB
- Ac. Weighting: Flat
- EQ:
  - Highpass 24 dB/Oct bei 2 kHz
  - Peak-Filter mit 10 dB Gain bei 7 kHz
- Block Size: 2048
- Resolution: Medium
- Window Function: Hanning
- Envelope: Log
- Attack: 50 ms
- Release: 500 ms
- Dry/Wet: 100 %
- Output Trim: 0.00 dB

Die Einstellungen wurden so getroffen, sodass die auf dem GUI angezeigte Gain Reduction der Frequenzen um 7 kHz im Durchschnitt bei etwa -10 dB lag. Gleichzeitig sollte das eingerichtete Hochpassfilter verhindern, dass die hohen Frequenzanteile der gesprochenen Vokale zwischen den Sibilanten fälschlich modifiziert werden.

Das Ergebnis der beiden Bearbeitungsmöglichkeiten wird im Zeitbereich in Abbildung 5.8 deutlich. Beide Darstellungen weisen beim Vergleich mit dem originalen Signal eine deutliche Absenkung der Signalamplitude in Höhe der Sibilanten auf. Ein Unterschied zwischen Plugin und EQ lässt sich jedoch erst bei Betrachtung der in Abbildung 5.10 ausgewerteten Schärfe erkennen. Beim EQ ist gegenüber dem originalen Signal ein na-

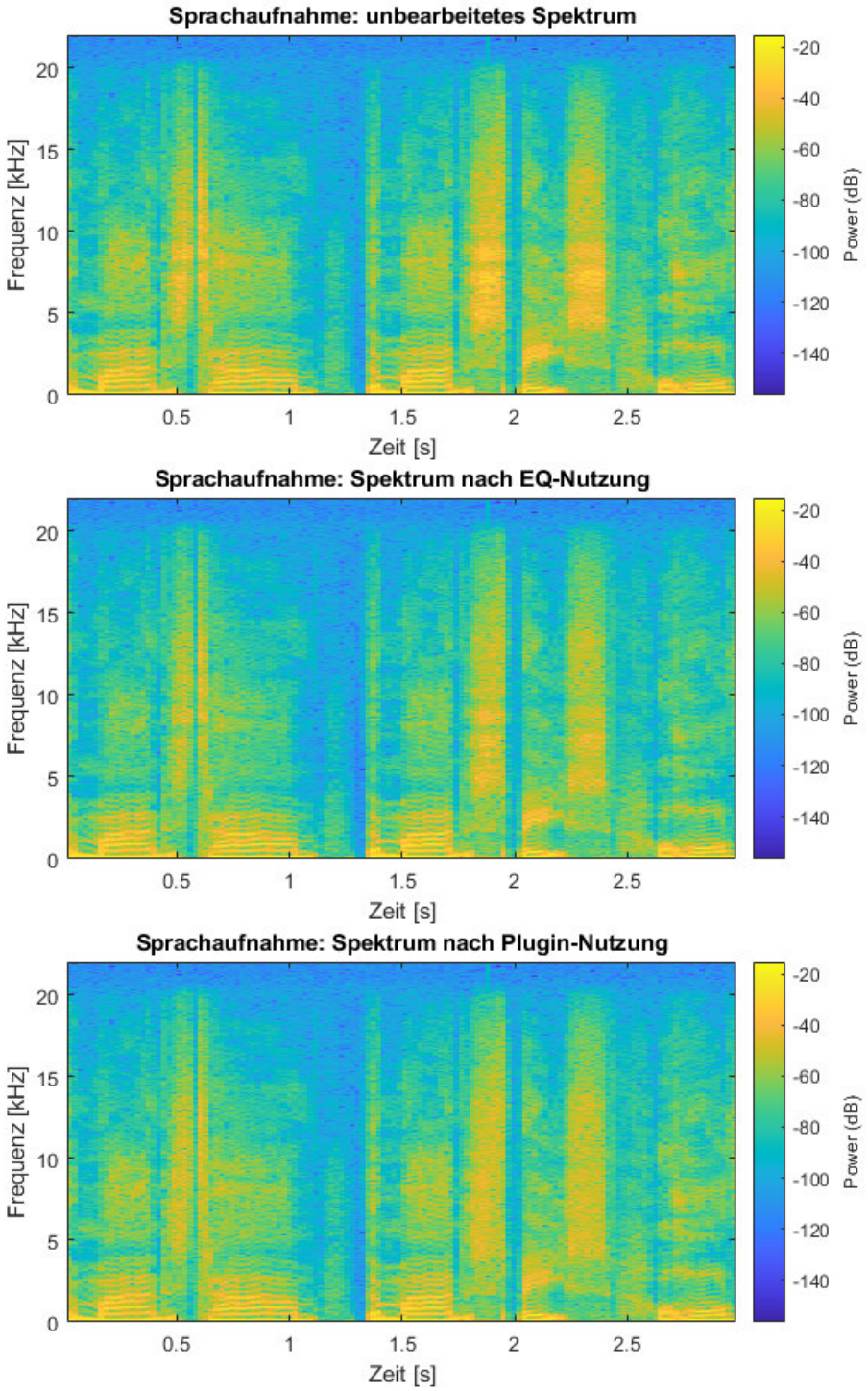


Abbildung 5.9: Vergleich der Spektren der Sprachaufnahme. Unbearbeitet, mit EQ und mit Plugin bearbeitet. (Eigene Darstellung)

hezu konstanter Reduktionseffekt aller Schärfewerte über die Zeitachse erkennbar. Im Gegensatz dazu führt der Plugin-Effekt zu einer Verminderung der hohen Schärfewerte, beeinflusst die geringen Schärfewerte in den dazwischenliegenden Zeitabschnitten allerdings nur minimal. Der Effekt des Plugins führt zu einer sichtbaren Komprimierung der Schärfewerte. Beim Vergleich mit den Spektrogrammen in Abbildung 5.9 wird deutlich, dass die spektralen Modifikationen des Plugins im Gegensatz zum EQ zur Kompressi-

on der hohen Frequenzanteile zwischen etwa 5 und 10 kHz führt. Beim EQ ist dagegen eine höhere Dynamik bei den Sibilanten ersichtlich. Beim Vergleich der beiden unteren Abbildungen ist sichtbar, dass diese Frequenzanteile in den Bereichen zwischen den Sibilanten beim Plugin etwas prägnanter sind als beim EQ.

Das entwickelte Plugin zeigt eine gute Performance bei der Einschränkung der Schärfe in der gegebenen Sprachaufnahme. Zeitgleich wird der adaptive Charakter der zeitvariablen Spektralbearbeitung deutlich. Während ein EQ auch einen Einfluss auf Passagen mit geringen Schärfewerten hat, so fällt der Effekt des Plugins an diesen Stellen eher gering aus. Für praktische Applikationen lässt sich demnach ein Nutzen bei der Beschränkung von als *zu scharf* wahrgenommenem Audiomaterial wie Zischlauten in Sprachaufnahmen ableiten, wo eine konstante Reduktion nicht immer sinnvoll ist. So kann ein Signal durch den EQ-Effekt als *dumpf* wahrgenommen werden, da die Schärfe auch einen maßgeblichen Indikator für die Wahrnehmung hoher Frequenzanteile durch den Menschen darstellt. Es bleibt Aufgabe des anwendenden Toningenieurs, zu entscheiden, bis zu welchem Grad eine wahrnehmbare Schärfe als vertretbar gilt. Bei den gegebenen Hörbeispielen ist es hierbei dem Leser überlassen, welche Aufnahme letztendlich aus eigener Perspektive *besser* oder *schlechter* klingt. Weitere Hörbeispiele und zugehörige Messungen sind im Anhang zu finden.

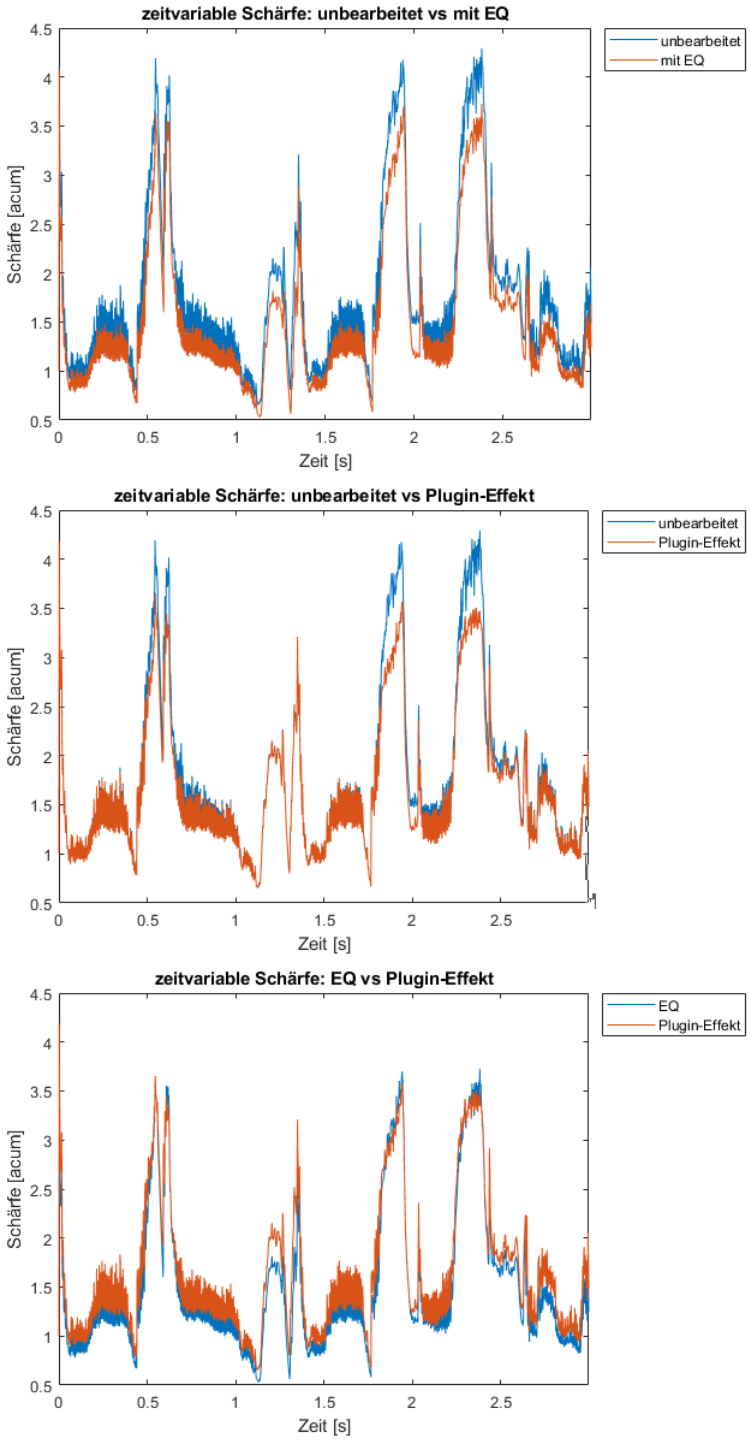


Abbildung 5.10: Vergleich der Schärfen der bearbeiteten Sprachaufnahme (Eigene Darstellung)

6 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit bestand in der Entwicklung eines Audio-Plugins zur dynamischen Anpassung von subjektiv als störend empfundenen Frequenzamplituden in der Entwicklungsumgebung Max/MSP. In einer initialen Analyse wurden die für das Vorhaben relevanten Anforderungen sowie praxisnahe Design-Ansätze beleuchtet. Im Theorieteil wurde im Anschluss die für die Entwicklung notwendigen fachlichen Grundlagen erläutert. Schwerpunkte wurden dabei auf die Themen Signalanalyse im Frequenzbereich, Filterung von Signalen und ausgewählte Aspekte des menschlichen Hörempfindens gesetzt. Letztere dienten im Verlauf der Arbeit als messtechnische Indikatoren für die Prüfung des entwickelten Plugins hinsichtlich seiner Eignung in der tontechnischen Praxis. Als ersten Schritt der praktischen Umsetzung wurde die Umgebung Max/MSP detailiert betrachtet. Ein besonderes Augenmerk lag dabei auf der Arbeitsweise mit dieser Programmierumgebung. Dabei fiel auf, dass sich die darin nativ implementierte STFT nur bedingt für das geplante Entwicklungsvorhaben eignet. Die datenstrombasierte Signalverarbeitung in Max/MSP erwies sich demnach als nachteilig bei der Implementierung eines zeitvariablen, linearen Faltungsprozesses. Die programmierte Funktionsweise basiert demzufolge auf der spektralen Modifikation. Das fertige Plugin vereint Konzepte traditioneller Audiobearbeitungs-Tools. Die Funktionsweise lässt sich schrittweise wie folgt beschreiben: Das Eingangssignal wird in kurze überlappende Zeitabschnitte zerlegt. Es folgt eine Spektralanalyse dieser. Über standardisierte Frequenzbewertungskurven oder einem implementierten Equalizer kann eine Frequenzbewertung über die gemessenen Spektren gelegt werden. Jede Frequenzamplitude wird individuell durch ein dem Dynamikkompressor ähnlichen Verfahren angepasst. Die Rücktransformation der sich überlappenden Zeitabschnitte ergibt das Ausgangssignal. Den Abschluss der Arbeit bilden neben Messungen zur Funktionsweise auch Untersuchungen ausgewählter psychoakustischer Parameter, genauer der spezifischen Lautheit und Schärfe anhand von Hörbeispielen. Die abschließenden Tests ergaben Unstimmigkeiten zwischen den gemessenen und erwarteten Amplituden von verarbeiteten idealisierten Sinuskurven als Testsignale. Eine definitive Erklärung für dieses Ergebnis bleibt unklar. Die Messergebnisse zu spezifischer Lautheit und Schärfe zeigen einen frequenz- und zeitabhängigen Kompressionseffekt von getesteten Hörbeispielen. Zudem wurde deutlich, dass die implementierten Frequenzbewertungskurven zur effizienten Fokussierung bestimmter Frequenzbereiche bei der frequenzabhängigen Kompression beitragen. Es ist ein erster Hinweis dafür, dass dieser Effekt zu einer ausgeglicheneren Wahrnehmung des Frequenzspektrums in der Tonpraxis führen kann. In Bezug auf die Forschungsfrage lässt sich diese in Hinblick auf vorangegangenen Aspekten wie folgt beantworten: Eine Implementierung des Plugins ist mit Abstrichen möglich. Diese beziehen sich im Speziellen auf eine Nicht-Implementierbarkeit der linearen Faltungsoperation unter Nutzung der in Max/MSP verfügbaren Objekte, wobei das Filterprofil auf Basis zeitpunktabhängiger Eingangsspektren gebildet wird; sowie

das eingeschränkte Processing der einzelnen Frequenzamplituden in Abhängigkeit zueinander aufgrund der datenstromorientierten Programmierweise. Eine effektive Funktionsweise, die der linearen Filterung nahekommt, konnte unter Nutzung spektraler Modifikationen implementiert werden.

Die Entwicklung des Plugins zeigt das Potenzial der Schaffung innovativer Werkzeuge für die Audiobranche. So verfolgen moderne Audio-Plugins die effektive Klangverbesserung mit adaptiven Algorithmen fernab von simplen EQ-Filtern und Tools. Ein besonderes Potenzial in aktueller Zeit bietet das Thema Künstliche Intelligenz. So werben Firmen wie iZotope [85] mit der Integration dieser Technologien in ihren Algorithmen. Es wäre denkbar, auch das entwickelte Plugin mit entsprechenden Modellen zu erweiteren, welche auf das Eingabematerial trainiert werden. In Kombination mit der Applikation psychoakustischer Erkenntnisse erscheinen autonome EQ-Systeme zur Schaffung einer speziellen Klangästhetik als plausibel.

Während die beschriebene Erkennung der störenden Frequenzamplituden auf der DFT beruht, so ist die Anwendung anderer Verfahren für spezifische Applikationen denkbar. Es gibt Plugin-EQs, welche die mit der Bark-Skala verwandte Mel-Skala als Basis für eine Filterbank zur Modellierung der kritischen Bänder nutzen [86]. In anderen Situationen, wo eine genaue Detektion einzelner störender Frequenzen gefordert ist, können sog. Polyphasen-FFT-Filterbänke effektiver bei der Reduktion des Leck-Effekts wirken. Um das schwache Auflösungsvermögen der DFT in den tiefen Frequenzen auszugleichen, erscheint eine Applikation der sog. Constant-Q-Transformation insbesondere bei musikalischem Material als sinnvoll. Auch Verfahren wie die Wavelet-Transformation könnten herangezogen werden. Während Max/MSP diese Analyseformen nativ nicht unterstützt, so erscheint eine manuelle Implementierung mittels Externals als plausibel. Dies erfordert jedoch ein spezialisiertes Wissen über effiziente Implementierungsmethoden solcher Algorithmen und bietet weiteren Forschungsstoff auf diesem Gebiet.

Es ist denkbar, einen oft mit Herausforderungen der Stabilität und Effizienz verbundenen Ansatz der Filterung über IIR-Filter auf Basis der Frequenzanalyse anzugehen. Insbesondere bei sehr schmalbandigen Filtern zur Erfassung einzelner Frequenzen wäre jedoch ein Modell für eine ressourcenschonende Implemetierung notwendig. Auch das Potenzial für Rundungsfehler müsste hierbei berücksichtigt werden. Ob dies mit einer wahrnehmbaren Klangverbesserung gegenüber dem implementierten Modell einherginge, bliebe zu untersuchen.

Es bietet sich auch an, einen Algorithmus zum automatischen Ausgleich der Lautheit zwischen Eingangs- und Ausgangsmaterial zu implementieren. Dies würde in der Praxis dem Phänomen entgegenwirken, dass *lauter* auf den Zuhörer gleichzeitig *besser* klingt. So ist eine effizientere Bewertung durch den Toningenieur möglich.

Literaturverzeichnis

[1] Steinberg. *Unsere Technologien*. Adresse: https://www.steinberg.net/de/technology/ (Zugriff am 29.12.2024).

[2] Cycling '74. *Products: Max*. Adresse: https://cycling74.com/products/max (Zugriff am 16.01.2025).

[3] Ableton AG. *Ableton Live*. Adresse: https://www.ableton.com/de/live/ (Zugriff am 16.01.2025).

[4] D. Stotz, "Bearbeitung von Sampling-Dateien," in *Computergestützte Audio- und Videotechnik*, D. Stotz, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, S. 99–132, ISBN: 978-3-662-58872-7. DOI: 10.1007/978-3-662-58873-4_3.

[5] H.-J. Maempel, "Audiobearbeitung," in *Handbuch der Audiotechnik*, S. Weinzierl, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, S. 1–30, ISBN: 978-3-662-60357-4. DOI: 10.1007/978-3-662-60357-4_29-1.

[6] FabFilter. *Pro-Q 4 Equalizer Plugin*. Adresse: https://www.fabfilter.com/products/pro-q-4-equalizer-plug-in (Zugriff am 13.01.2025).

[7] FabFilter. *Pro-C 2 Compressor Plugin*. Adresse: https://www.fabfilter.com/products/pro-c-2-compressor-plug-in (Zugriff am 12.01.2025).

[8] Tokyo Dawn Records. *TDR Prism*. Adresse: https://www.tokyodawn.net/tdr-prism/ (Zugriff am 12.01.2025).

[9] J.-F. Charles, "A Tutorial on Spectral Sound Processing Using Max/MSP and Jitter," *Computer Music Journal*, Jg. 32, Nr. 3, S. 87–102, 2008, ISSN: 0148-9267. DOI: 10.1162/comj.2008.32.3.87. Adresse: https://direct.mit.edu/comj/article/32/3/87/94223/A-Tutorial-on-Spectral-Sound-Processing-Using-Max.

[10] A. Mertins, "Zeitkontinuierliche Signale und Systeme," in *Signaltheorie*, A. Mertins, Hrsg., Wiesbaden: Springer Fachmedien Wiesbaden, 2023, S. 53–112, ISBN: 978-3-658-41528-0. DOI: 10.1007/978-3-658-41529-7_3.

[11] Ulrich, *Laplace-, Fourier- und z-Transformation*. Wiesbaden: Springer Fachmedien Wiesbaden, 2017, ISBN: 978-3-658-03449-8. DOI: 10.1007/978-3-658-03450-4.

[12] S. Weinzierl, "Audiosignale und -systeme," in *Handbuch der Audiotechnik*, S. Weinzierl, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, S. 1–22, ISBN: 978-3-662-60357-4. DOI: 10.1007/978-3-662-60357-4_1-1. Adresse: https://link.springer.com/referenceworkentry/10.1007/978-3-662-60357-4_1-1.

[13] S. Goebbels und S. Ritter, "Fourier-Transformation," in *Mathematik Verstehen und Anwenden*, S. R. S. GOEBBELS, Hrsg., [S.l.]: SPRINGER SPEKTRUM, 2024, S. 335–354, ISBN: 978-3-662-68368-2. DOI: 10.1007/978-3-662-68369-9_12.

[14] T. Westermann, “Fourier-Transformation,” in *Mathematik für Ingenieure*, T. Westermann, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, S. 623–656, ISBN: 978-3-662-61322-1. DOI: 10.1007/978-3-662-61323-8_16.

[15] K.-D. Kammeyer und K. Kroschel, “Diskrete Signale und Systeme,” in *Digitale Signalverarbeitung*, K.-D. Kammeyer und K. Kroschel, Hrsg., Wiesbaden: Vieweg+Teubner Verlag, 2002, S. 9–52, ISBN: 978-3-519-46122-7. DOI: 10.1007/978-3-663-09805-8_2.

[16] A. Mertins, “Diskrete Signale und Systeme,” in *Signaltheorie*, A. Mertins, Hrsg., Wiesbaden: Springer Fachmedien Wiesbaden, 2023, S. 113–182, ISBN: 978-3-658-41528-0. DOI: 10.1007/978-3-658-41529-7_4.

[17] K. Bressler und J. Endres, “Digital-Analog (DA)-, Analog-Digital (AD)-Wandler und digitale Filter,” in *Elektronik für Ingenieure und Naturwissenschaftler*, Hering, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 467–502, ISBN: 978-3-662-54213-2. DOI: 10.1007/978-3-662-54214-9_9.

[18] S. Goebbels und S. Ritter, “Diskrete Fourier-Transformation,” in *Mathematik Verstehen und Anwenden*, S. R. S. GOEBBELS, Hrsg., [S.l.]: SPRINGER SPEKTRUM, 2024, S. 383–439, ISBN: 978-3-662-68368-2. DOI: 10.1007/978-3-662-68369-9_14.

[19] J. Gomes und L. Velho, “The Fourier Transform,” in *From fourier analysis to wavelets*, J. Gomes und L. Velho, Hrsg., New York NY: Springer Science+Business Media, 2015, S. 29–46, ISBN: 978-3-319-22074-1. DOI: 10.1007/978-3-319-22075-8_3.

[20] S. Palani, “Discrete Fourier Transform and Computation,” in *Discrete Time Systems and Signal Processing*, S. Palani, Hrsg., Cham: Springer International Publishing, 2023, S. 291–452, ISBN: 978-3-031-32420-8. DOI: 10.1007/978-3-031-32421-5_3.

[21] H. Ulrich und S. Ulrich, “Diskrete Fourier Transformation (DFT),” in *Laplace-Transformation, diskrete Fourier-Transformation und z-Transformation*, Ser. Lehrbuch, H. Ulrich und S. Ulrich, Hrsg., Wiesbaden: Springer Vieweg, 2023, S. 203–229, ISBN: 978-3-658-31876-5. DOI: 10.1007/978-3-658-31877-2_6.

[22] K.-D. Kammeyer und K. Kroschel, “Die diskrete Fourier-Transformation (DFT),” in *Digitale Signalverarbeitung*, K.-D. Kammeyer und K. Kroschel, Hrsg., Wiesbaden: Vieweg+Teubner Verlag, 2002, S. 219–258, ISBN: 978-3-519-46122-7. DOI: 10.1007/978-3-663-09805-8_6.

[23] D. Sundararajan, “The Discrete Fourier Transform,” in *Signals and Systems*, D. Sundararajan, Hrsg., Cham: Springer Nature Switzerland, 2023, S. 125–160, ISBN: 978-3-031-19376-7. DOI: 10.1007/978-3-031-19377-4_5.

[24] A. Mertins, “Diskrete Blocktransformationen,” in *Signaltheorie*, A. Mertins, Hrsg., Wiesbaden: Springer Fachmedien Wiesbaden, 2023, S. 183–208, ISBN: 978-3-658-41528-0. DOI: 10.1007/978-3-658-41529-7_5.

[25] D. Sundararajan, “The Discrete-Time Fourier Transform,” in *Signals and Systems*, D. Sundararajan, Hrsg., Cham: Springer Nature Switzerland, 2023, S. 197–240, ISBN: 978-3-031-19376-7. DOI: 10.1007/978-3-031-19377-4_7.

[26] K.-D. Kammeyer und K. Kroschel, “Spektralanalyse determinierter Signale,” in *Digitale Signalverarbeitung*, K.-D. Kammeyer und K. Kroschel, Hrsg., Wiesbaden: Vieweg+Teubner Verlag, 2002, S. 259–293, ISBN: 978-3-519-46122-7. DOI: 10.1007/978-3-663-09805-8_7.

[27] U. Karrenberg, “Lineare und nichtlineare Prozesse,” in *Signale - Prozesse - Systeme*, Springer, Berlin, Heidelberg, 2010, S. 189–230, ISBN: 978-3-642-01864-0. DOI: 10.1007/978-3-642-01864-0_8. Adresse: https://link.springer.com/chapter/10.1007/978-3-642-01864-0_8.

[28] A. V. Oppenheim, J. R. Buck und R. W. Schafer, *Discrete-time signal processing* (Prentice Hall signal processing series), 2. ed., internat. ed. Upper Saddle River, NJ: Prentice-Hall Internat, 1999, ISBN: 0130834432.

[29] A. Ukil, Y. M. Yeap und K. Satpathi, “Frequency-Domain Based Fault Detection: Application of Short-Time Fourier Transform,” in *Fault Analysis and Protection System Design for DC Grids*, Ser. Power Systems, A. Ukil, Y. M. Yeap und K. Satpathi, Hrsg., Singapore: Springer Singapore, 2020, S. 195–221, ISBN: 978-981-15-2976-4. DOI: 10.1007/978-981-15-2977-1_6.

[30] R. C. Maher, “Forensische Audioanalyse,” in *Handbuch der Audiotechnik*, S. Weinzierl, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, S. 1–16, ISBN: 978-3-662-60357-4. DOI: 10.1007/978-3-662-60357-4_12-1.

[31] J. Gomes und L. Velho, “Windowed Fourier Transform,” in *From fourier analysis to wavelets*, J. Gomes und L. Velho, Hrsg., New York NY: Springer Science+Business Media, 2015, S. 47–60, ISBN: 978-3-319-22074-1. DOI: 10.1007/978-3-319-22075-8_4.

[32] W. S. Gan, “Fast Fourier Transform,” in *Signal Processing and Image Processing for Acoustical Imaging*, Springer, Singapore, 2020, S. 17–20, ISBN: 978-981-10-5550-8. DOI: 10.1007/978-981-10-5550-8_5. Adresse: https://link.springer.com/chapter/10.1007/978-981-10-5550-8_5.

[33] H. Sorensen, D. Jones, M. Heideman und C. Burrus, “Real-valued fast Fourier transform algorithms,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Jg. 35, Nr. 6, S. 849–863, 1987, ISSN: 0096-3518. DOI: 10.1109/TASSP.1987.1165220.

[34] H. Ulrich und S. Ulrich, “Anwendungen der Laplace-Transformation,” in *Laplace-Transformation, diskrete Fourier-Transformation und z-Transformation*, Ser. Lehrbuch, H. Ulrich und S. Ulrich, Hrsg., Wiesbaden: Springer Vieweg, 2023, S. 111–178, ISBN: 978-3-658-31876-5. DOI: 10.1007/978-3-658-31877-2_4.

[35] D. Sundararajan, “Time-Domain Analysis of Discrete Systems,” in *Signals and Systems*, D. Sundararajan, Hrsg., Cham: Springer Nature Switzerland, 2023, S. 65–95, ISBN: 978-3-031-19376-7. DOI: 10.1007/978-3-031-19377-4_3.

[36] M. Vorländer, “Digitale Signalverarbeitung in der Messtechnik,” in *Digitale Signalverarbeitung in der Messtechnik*, Ser. Fachwissen Technische Akustik, M. Möser, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, S. 1–28, ISBN: 978-3-662-56612-1. DOI: 10.1007/978-3-662-56613-8_1.

[37] O. Beucher, “Analoge Signale und Systeme,” in *Signale und Systeme: Theorie, Simulation, Anwendung*, O. Beucher, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, S. 33–210, ISBN: 978-3-662-58043-1. DOI: 10.1007/978-3-662-58044-8_2.

[38] K.-D. Kammeyer und K. Kroschel, “Nichtrekursive Filter,” in *Digitale Signalverarbeitung*, K.-D. Kammeyer und K. Kroschel, Hrsg., Wiesbaden: Vieweg+Teubner Verlag, 2002, S. 157–217, ISBN: 978-3-519-46122-7. DOI: 10.1007/978-3-663-09805-8_5.

[39] S. Palani, “Representation of Discrete Signals and Systems,” in *Discrete Time Systems and Signal Processing*, S. Palani, Hrsg., Cham: Springer International Publishing, 2023, S. 1–114, ISBN: 978-3-031-32420-8. DOI: 10.1007/978-3-031-32421-5_1.

[40] K.-D. Kammeyer und K. Kroschel, “Die Z-Transformation,” in *Digitale Signalverarbeitung*, K.-D. Kammeyer und K. Kroschel, Hrsg., Wiesbaden: Vieweg+Teubner Verlag, 2002, S. 53–76, ISBN: 978-3-519-46122-7. DOI: 10.1007/978-3-663-09805-8_3.

[41] S. Palani, “The z-transform Analysis of Discrete Time Systems,” in *Discrete Time Systems and Signal Processing*, S. Palani, Hrsg., Cham: Springer International Publishing, 2023, S. 115–290, ISBN: 978-3-031-32420-8. DOI: 10.1007/978-3-031-32421-5_2.

[42] S. Palani, “Design of Finite Impulse Response (FIR) Digital Filters,” in *Discrete Time Systems and Signal Processing*, S. Palani, Hrsg., Cham: Springer International Publishing, 2023, S. 591–732, ISBN: 978-3-031-32420-8. DOI: 10.1007/978-3-031-32421-5_5.

[43] M. Werner, “Schnelle Fourier-Transformation,” in *Digitale Signalverarbeitung mit MATLAB®*, M. Werner, Hrsg., Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 113–141, ISBN: 978-3-658-18646-3. DOI: 10.1007/978-3-658-18647-0_6.

[44] A. Mertins, “Die Kurzzeit-Fourier-Transformation,” in *Signaltheorie*, A. Mertins, Hrsg., Wiesbaden: Springer Fachmedien Wiesbaden, 2023, S. 313–328, ISBN: 978-3-658-41528-0. DOI: 10.1007/978-3-658-41529-7_8.

[45] J. O. Smith, *Spectral audio signal processing / Julius O. Smith III, Center for Computer Research in Music and Acoustics (CCRMA), Department of Music, Stanford University, Stanford, California 94305 USA*. [Stanford, Calif.?]: W3K, 2011, ISBN: 9780974560731.

[46] J. B. Allen und L. R. Rabiner, “A unified approach to short-time Fourier analysis and synthesis,” *Proceedings of the IEEE*, Jg. 65, Nr. 11, S. 1558–1564, 1977, ISSN: 0018-9219. DOI: 10.1109/PROC.1977.10770.

[47] J. Allen, “Short term spectral analysis, synthesis, and modification by discrete Fourier transform,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Jg. 25, Nr. 3, S. 235–238, 1977, ISSN: 0096-3518. DOI: 10.1109/TASSP.1977.1162950.

[48] M. Portnoff, “Time-scale modification of speech based on short-time Fourier analysis,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Jg. 29, Nr. 3, S. 374–390, 1981, ISSN: 0096-3518. DOI: 10.1109/TASSP.1981.1163581.

[49] S. Ruiz, T. Dietzen, T. van Waterschoot und M. Moonen, “A comparison between overlap-save and weighted overlap-add filter banks for multi-channel Wiener filter based noise reduction,” in *2021 29th European Signal Processing Conference (EUSIPCO)*, IEEE, 8/23/2021 - 8/27/2021, S. 336–340, ISBN: 978-9-0827-9706-0. DOI: 10.23919/EUSIPCO54536.2021.9616352.

[50] R. Crochiere, “A weighted overlap-add method of short-time Fourier analysis/Synthesis,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Jg. 28, Nr. 1, S. 99–102, 1980, ISSN: 0096-3518. DOI: 10.1109/TASSP.1980.1163353.

[51] P. Zeller, “Psychoakustik,” in *Handbuch Fahrzeugakustik*, P. Zeller, Hrsg., Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 251–265, ISBN: 978-3-658-18519-0. DOI: 10.1007/978-3-658-18520-6_8.

[52] E. Zwicker, Hrsg., *Psychoakustik*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, ISBN: 978-3-540-11401-7. DOI: 10.1007/978-3-642-68510-1.

[53] A. Schick, Hrsg., *Schallbewertung: Grundlagen der Lärmforschung*. Berlin und New York: Springer, 1990, ISBN: 978-3-540-52922-4. DOI: 10.1007/978-3-642-50267-5.

[54] W. Ellermeier und J. Hellbrück, “Psychoakustik,” in *Handbuch der Audiotechnik*, S. Weinzierl, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, S. 1–17, ISBN: 978-3-662-60357-4. DOI: 10.1007/978-3-662-60357-4_4-2.

[55] ISO 226:2023-03, Acoustics — Normal equal-loudness-level contours.

[56] H. Bernstein, “Spracherzeugung und Wahrnehmung,” in *Elektroakustik*, Springer Vieweg, Wiesbaden, 2019, S. 47–83, ISBN: 978-3-658-25174-1. DOI: 10.1007/978-3-658-25174-1_2. Adresse: https://link.springer.com/chapter/10.1007/978-3-658-25174-1_2.

[57] IEC, *Elektroakustik - Schallpegelmesser - Teil 1: Anforderungen*.

[58] AES. *Audio-Topics: Loudness | Learn More*. Adresse: https://aes2.org/resources/audio-topics/loudness-project/learn-more/ (Zugriff am 04.01.2025).

[59] ITU-R BS.468-4:1986-07, Measurement of audio-frequency noise voltage level in sound broadcasting.

[60] ISO 21727:2016-05, Cinematography — Method of measurement of perceived loudness of short duration motion-picture audio material.

[61] EBU R 128-2023-11, Loudness normalisation and permitted maximum level of audio signals.

[62] ITU-R BS.1770-5:2023-11, Algorithms to measure audio programme loudness and true-peak audio level.

[63] DIN 45631/A1:2010-03, Berechnung des Lautstärkepegels und der Lautheit aus dem Geräuschspektrum - Verfahren nach E. Zwicker - Änderung 1: Berechnung der Lautheit zeitvarianter Geräusche; mit CD-ROM.

[64] E. Zwicker, “Lautheit zeitabhängiger Schalle,” in *Psychoakustik*, E. Zwicker, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, S. 102–105, ISBN: 978-3-540-11401-7. DOI: 10.1007/978-3-642-68510-1_10.

[65] E. Zwicker, “Verdeckung,” in *Psychoakustik*, E. Zwicker, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, S. 35–53, ISBN: 978-3-540-11401-7. DOI: 10.1007/978-3-642-68510-1_3.

[66] E. Zwicker, “Funktionsschema der Lautheit,” in *Psychoakustik*, E. Zwicker, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, S. 128–145, ISBN: 978-3-540-11401-7. DOI: 10.1007/978-3-642-68510-1_15.

[67] B. C. J. Moore, B. R. Glasberg, A. Varathanathan und J. Schlittenlacher, “A Loudness Model for Time-Varying Sounds Incorporating Binaural Inhibition,” *Trends in Hearing*, Jg. 20, S. 2 331 216 516 682 698, 2016. DOI: 10.1177/2331216516682698.

[68] E. Zwicker, “Schärfe,” in *Psychoakustik*, E. Zwicker, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, S. 84–85, ISBN: 978-3-540-11401-7. DOI: 10.1007/978-3-642-68510-1_6.

[69] DIN 45692:2009-08, Messtechnische Simulation der Hörempfindung Schärfe.

[70] Cycling ’74. *About the Company*. Adresse: https://cycling74.com/company (Zugriff am 19.12.2024).

[71] Cycling ’74. *Object & Operator Reference | Cycling ’74 Documentation*. Adresse: https://docs.cycling74.com/reference/?page=1 (Zugriff am 20.12.2024).

[72] Cycling ’74. *Abstractions | Cycling ’74 Documentation*. Adresse: https://docs.cycling74.com/userguide/abstractions/ (Zugriff am 20.12.2024).

[73] Cycling ’74. *Scheduler and Priority | Cycling ’74 Documentation*. Adresse: https://docs.cycling74.com/userguide/scheduler/ (Zugriff am 22.12.2024).

[74] Cycling ’74. *GenExpr | Cycling ’74 Documentation*. Adresse: https://docs.cycling74.com/userguide/gen/gen_genexpr/ (Zugriff am 22.12.2024).

[75] Cycling ’74. *js - Max Reference | Cycling ’74 Documentation*. Adresse: https://docs.cycling74.com/reference/js/ (Zugriff am 06.01.2025).

[76] SIEMENS Community. *Window Correction Factors*. Adresse: https://community.sw.siemens.com/s/article/window-correction-factors (Zugriff am 28.12.2024).

[77] Cycling ’74. *pfft~ - MSP Reference | Cycling ’74 Documentation*. Adresse: https://docs.cycling74.com/reference/pfft~/ (Zugriff am 27.12.2024).

[78] Robert Bristow-Johnson. *Cookbook formulae for audio EQ biquad filter coefficients*. Adresse: https://webaudio.github.io/Audio-EQ-Cookbook/Audio-EQ-Cookbook.txt (Zugriff am 14.01.2025).

[79] D. Giannoulis, M. Massberg und J. Reiss, “Digital Dynamic Range Compressor Design—A Tutorial and Analysis,” *AES: Journal of the Audio Engineering Society*, Jg. 60, Juli 2012.

[80] Cycling ’74. *The jsui Object - Max 8 Documentation | Cycling ’74 Documentation*. Adresse: https://docs.cycling74.com/legacy/max8/vignettes/jsuiobject (Zugriff am 01.01.2025).

[81] Cycling ’74. *MGraphics - Max JS API | Cycling ’74 Documentation*. Adresse: https://docs.cycling74.com/apiref/js/mgraphics/ (Zugriff am 01.01.2025).

[82] S. Tayebi Arasteh und A. Kalisz, “Conversion Between Cubic Bezier Curves and Catmull–Rom Splines,” *SN Computer Science*, Jg. 2, Nr. 5, 2021, ISSN: 2662-995X. DOI: 10.1007/s42979-021-00770-x.

[83] Dillon Bastan. *Inspired by Nature Max for Live Pack | Ableton*. Adresse: https://www.ableton.com/de/packs/inspired-nature/ (Zugriff am 16.01.2025).

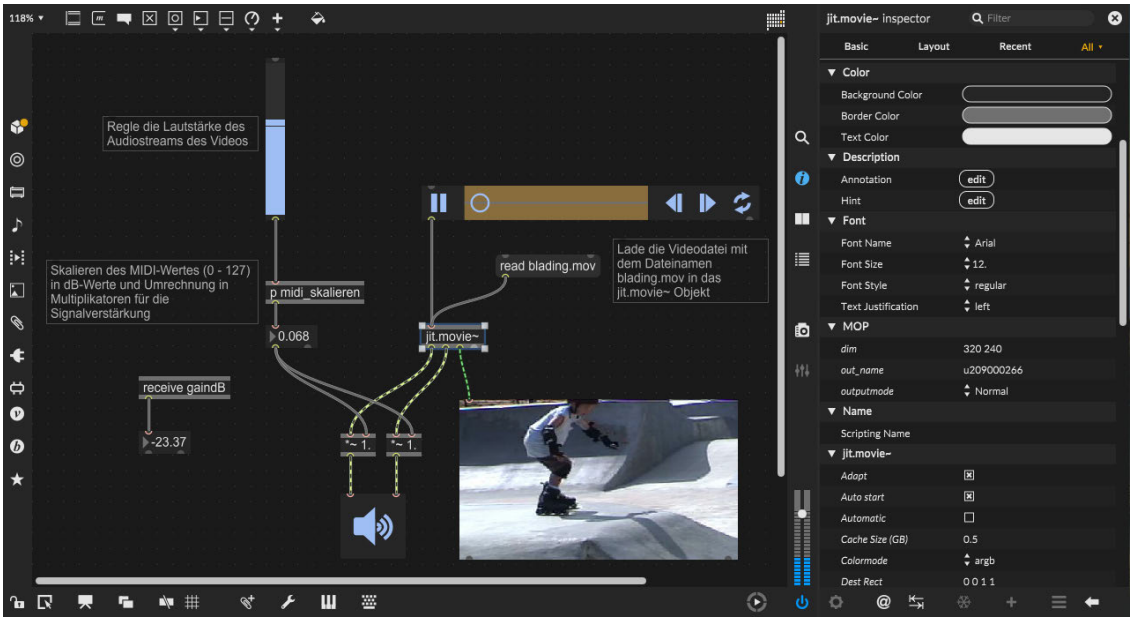
[84] Loopmasters. *Loopmasters Mixtape Sample Pack | Ableton*. Adresse: https://www.ableton.com/en/packs/loopmasters-mixtape/ (Zugriff am 16.01.2025).

[85] iZotope. *Plugins for Audio Restoration, Mixing, Mastering and More*. Adresse: https://www.izotope.com (Zugriff am 14.01.2025).

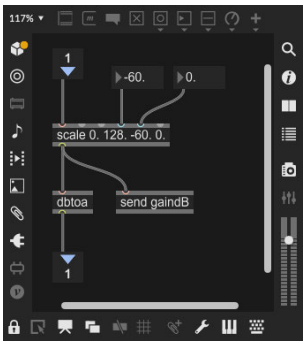
[86] Newfangled Audio. *EQuivocate Mixing & Mastering EQ*. Adresse: https://www.newfangledaudio.com/equivocate (Zugriff am 13.01.2025).

Anlagen

A Visualisierung der Arbeitsweise in Max/MSP



(a) Main-Patch des Beispiel-Patches



(b) Subpatcher *midi_skalieren* aus dem Beispiel-Patch

Abbildung A.1: Darstellung eines Beispiel-Patches in Max/MSP (Eigene Darstellung)

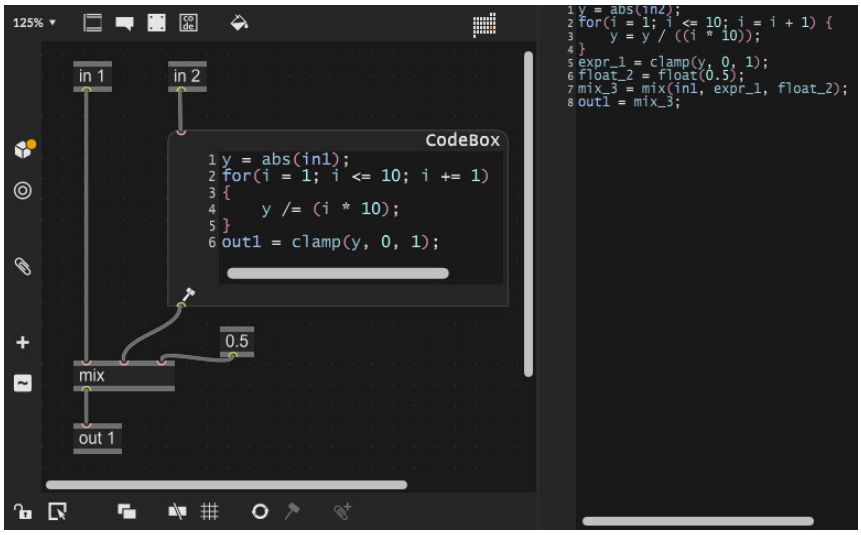


Abbildung A.2: Ansicht eines Gen-Patchers (Eigene Darstellung)

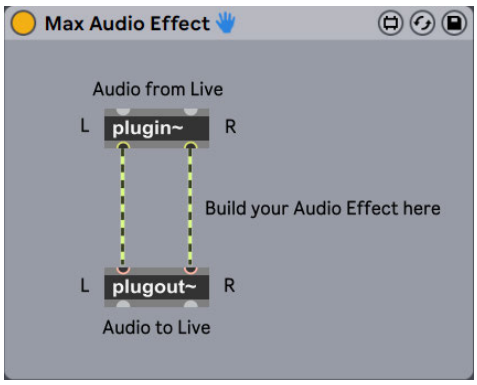


Abbildung A.3: Ausgangspunkt eines M4L-Plugins (Eigene Darstellung)

B Übersicht der Plugin-Parameter

Kompressionsspezifische Parameter

Die Parameter dienen der Einstellung des Kompressionseffekts im Spektralbereich. Sie bestimmen den Grad, mit dem die Kompression pro Frequency Bin auftritt.

- **Threshold**
  Er stellt den spektralen Schwellwert dar, welchen die Amplituden der individuellen Frequency Bins überschreiten müssen, um vom System als *störend* erkannt zu werden. Der Parameter kann vom Anwender in der Dezibelskala eingestellt werden.

- **Ratio**
  Dieser Wert definiert die Intensität, mit welcher als *störend* erkannte Amplituden reduziert werden. Er wird als Verhältnis angegeben. Der höchste einstellbare Wert von 64:1 bedeutet, dass bei einer Amplitudenüberschreitung des Schwellwerts um 64 dB lediglich eine Überschreitung von 1 dB am Ausgang messbar ist.

- **Knee**
  Der Knee-Parameter definiert einen glatten Übergang zwischen den Amplitudenbereichen, wo eine Kompression auftritt und wo die Amplituden linear übertragen werden. Er wird in der Dezibelskala angegeben. Ein Wert von 6 dB bedeutet, dass auch Amplituden 3 dB unter dem Schwellwert eine geringe Kompression erfahren, während der volle Effekt erst 3 dB über dem Schwellwert eintritt.

Sidechain-Parameter

Die Parameter dienen dem Abgreifen eines externen Analysesignals. Dieses kann genutzt werden, um als Steuersignal den Effekt in Anwendung auf ein anderes Signal zu steuern.

- **External Sidechain**
  Das Einschalten dieser Option legt den externen Sidechain-Eingang als Signalpfad zur Frequenzanalyse, auf welche die Amplitudenanalyse angewandt wird, fest.

- **Audio From**
  Die Dropdown-Menüs legen die Tonspur in Ableton Live fest, von der das Signal für den externen Sidechain-Eingang abgegriffen wird.

- **SC Trim**
  Der Parameter als Lautstärkeregler des Analysesignals bzw. Sidechain-Signals dient

der Kompensation zu hoher oder zu geringer Pegel. Er wird in Dezibel angegeben.

Parameter der Frequenzbewertung

Diese Parameter dienen der Frequenzbewertung der in Echtzeit gemessenen Amplituden. Neben erläuterten Bewertungskurven steht dem Anwender ein parametrischer EQ zur individuellen Bewertung zur Verfügung.

- **Acoustic Weighting**
  Über das Dropdown-Menü ist eine Auswahl standardisierter Frequenzbewertungskurven möglich. Sie beschränkt sich auf die in Abschnitt 3.3.2 erwähnten Kurven. Daneben sind sog. Tilt-Funktionen, welche in herkömmlichen Spectrum Analyzern zu finden sind, implementiert.

- **EQ-Interface**
  Über die nummerierten Knotenpunkte lassen sich die einzelnen Filter in Frequenz, Verstärkung und Güte anpassen. Ziel dieser Steuerung war eine intuitive Bedienbarkeit der EQ-Filter, was einer zeitsparenden Anwendung in der Praxis zugutekommt. Der gelb hervorgehobene Knotenpunkt 2 in Abbildung 5.1 zeigt das angewählte Filter. Die Filterparameter unter dem EQ-Interface passen sich dem angewählten Filter an und können auch darüber spezifiziert werden. Grau dargestellte Knotenpunkte symbolisieren ein inaktives Filter. Die Filter können über den nebem dem *Freq*-Parameter sichtbaren, gelben Button einzeln aktiviert oder deaktiviert werden. Es können bis zu acht Filter gleichzeitig genutzt werden. Das Hinzufügen weiterer Filter geschieht durch das Schweben mit dem Mauszeiger über die EQ-Kurve und dem anschließenden Gedrückthalten der linken Maustaste und Ziehen in eine beliebige Richtung. Alternativ kann ein Knotenpunkt durch Doppelklick an der gewünschten Stelle eingefügt werden. Je nach Frequenzbereich, in dem das Hinzufügen erfolgt, werden unterschiedliche Filtertypen standardmäßig hinzugefügt, die nachträglich geändert werden können. So sind hinzugefügte Filter unterhalb von 30 Hz standardmäßig Hochpässe zweiter Ordnung, zwischen 30 Hz und 80 Hz Lowshelf-Filter, zwischen 80 Hz und 8000 Hz Peak-Filter, zwischen 8000 Hz und 14000 Hz Highshelf-Filter und oberhalb von 14000 Hz Tiefpässe zweiter Ordnung. Durch das Gedrückthalten der ALT-Taste und Doppelklick auf einen Knotenpunkt werden Filter dagegen entfernt. Es ist vom gewählten Filtertyp abhängig, ob ein vertikales Ziehen des Knotenpunkts eine Änderung der Verstärkung oder der Güte bewirkt. Bei Peak-, Highshelf- und Lowshelf-Filtern ist dies grundsätzlich die Verstärkung. Durch Gedrückthalten der ALT-Taste und Ziehen eines Knotenpunkts kann stattdessen die Güte verändert werden, bei Hoch- und Tiefpässen hingegen die Flankensteilheit.

- **Freq**
  Die Mitten- bzw. Grenzfrequenz (*Cutoff*-Frequenz) des angewählten Filters wird über diesen Parameter bestimmt.

- **Q**
  Der Wert bestimmt den Gütefaktor des angewählten Filters.

- **Gain**
  Dieser Parameter bestimmt die Verstärkung bei Shelving- und Peak-Filtern. Er ist zwischen -20 dB und 20 dB wählbar.

- **Slope**
  Der Wert bestimmt die Steilheit eines Hoch- oder Tiefpassfilters. Er ist stufenweise als ein Vielfaches von 6 bis zu 96 dB/Oktave wählbar.

- **Type**
  Über das Dropdown-Menü kann der Filtertyp geändert werden. Es stehen dem Anwender die Filtertypen Peak-, Notch-, Bandpass-, Highshelf- und Lowshelf-Filter zweiter Ordnung sowie Hoch- und Tiefpassfilter zur Verfügung.

Parameter der STFT

- **Block Size**
  Der Wert als Zweierpotenz bestimmt die Länge der mittels FFT analysierten Zeitabschnitte in Samples.

- **Resolution**
  Dies bestimmt den Overlap-Faktor der STFT. Er ist als Faktor 4 (*Low*) bis maximal 32 (*Ultra*) wählbar.

- **Zero Pad**
  Es wird die FFT-Länge um den angegebenen Faktor erhöht, wobei die benötigten Samples mit Nullen aufgefüllt werden.

- **Window Type**
  Für das Analysesignal stehen verschiedene Fensterfunktionen zur Verfügung. Es kann zwischen Hanning-, Rechteck-, Hamming-, Blackman-, Blackman-Harris-, Flat-Top-, Dreieck- und Bartlett-Fenstern gewählt werden.

Kompressionshüllkurve

Diese Parameter steuern das Glättungsverhalten der spektralen Modifikationen über die

Zeit.

- **Envelope**
  Es kann zwischen linearen und logarithmischen Attack- und Release-Kurven umgeschaltet werden.

- **Attack**
  Die Attack-Zeit bestimmt, wie schnell der vollständige Kompressionseffekt über die Zeit bei Überschreiten des Schwellwerts greift.

- **Release**
  Die Release-Zeit hingegen gibt an, wie lange der Kompressionseffekt nach Unterschreiten des Schwellwerts noch ausklingt.

Zusätzliche Parameter

- **Stereo mode**
  Es kann zwischen der klassischen Links-Rechts-Verarbeitung (*Stereo*) und der Verarbeitung von Summen- und Differenzsignal (*Mid/Side*) umgeschaltet werden.

- **Output Trim**
  Der Parameter dient der nachträglichen Kompensation der empfundenen Lautstärke des Ausgangssignals, um einen subjektiven Vorher-Nachher-Vergleich zwischen unbearbeitetem und bearbeitetem Signal ziehen zu können.

- **Dry/Wet**
  Der Wert stellt den prozentualen Gesamteffekt des Plugins ein. 0 % bedeutet, dass keine spektralen Modifikationen auftreten.

C Weitere Hörbeispiele zur Spezifischen Lautheit

Die Rohaufnahmen der folgenden Beispiele wurden zur Illustration aus [84] entnommen.

C.1 Hörbeispiel Spezifische Lautheit 2: Groove

1) *BA_Drechsler_HBsp_SpezLautheit_2_Rohaufnahme.wav*
2) *BA_Drechsler_HBsp_SpezLautheit_2_A-Weighting.wav*
3) *BA_Drechsler_HBsp_SpezLautheit_2_CustomWeighting.wav*

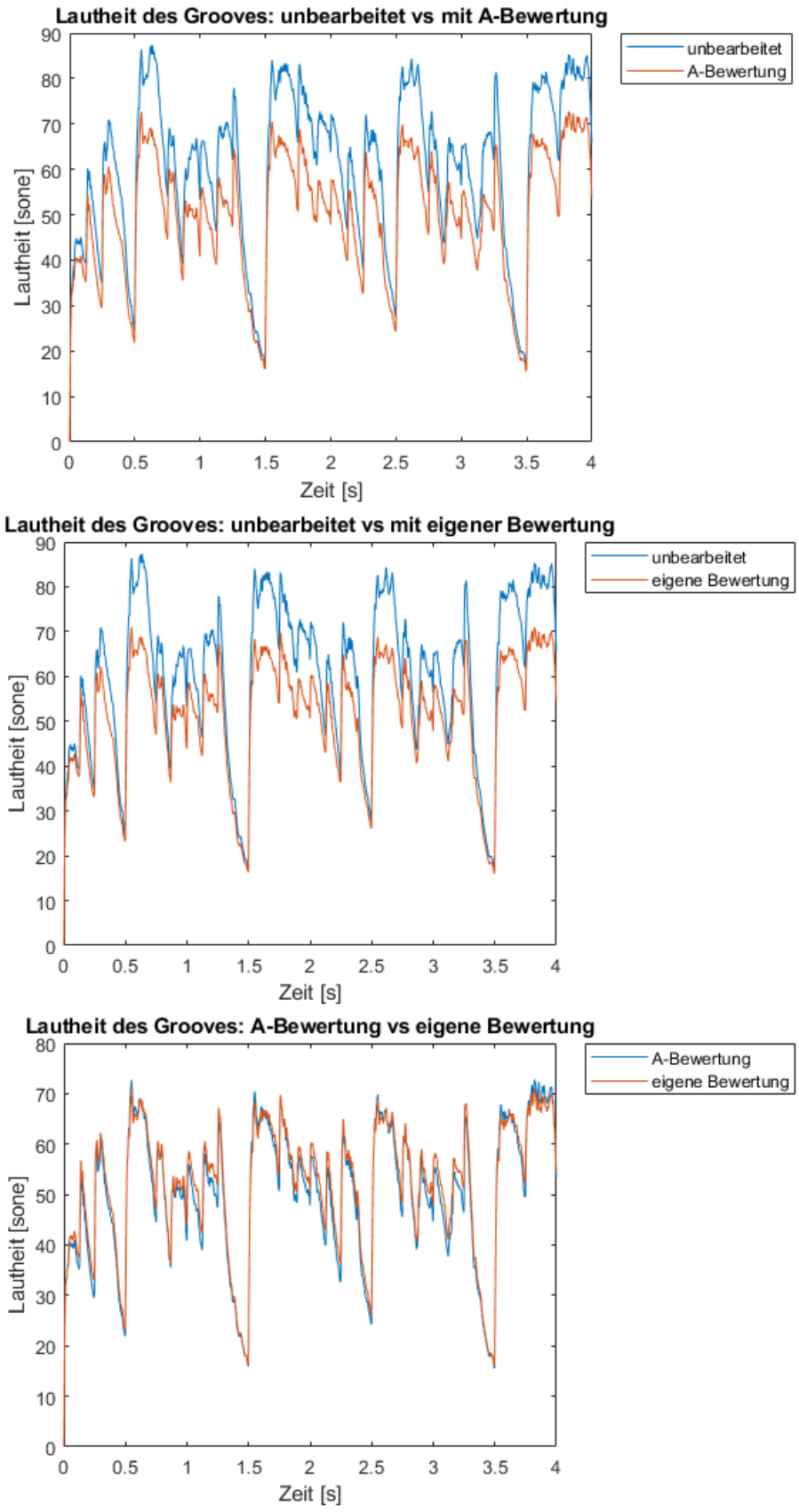


Abbildung C.1: Vergleich der Lautheiten des Grooves. Unbearbeitet, mit Bearbeitung unter A-Bewertung und unter eigener Bewertung (Eigene Darstellung)

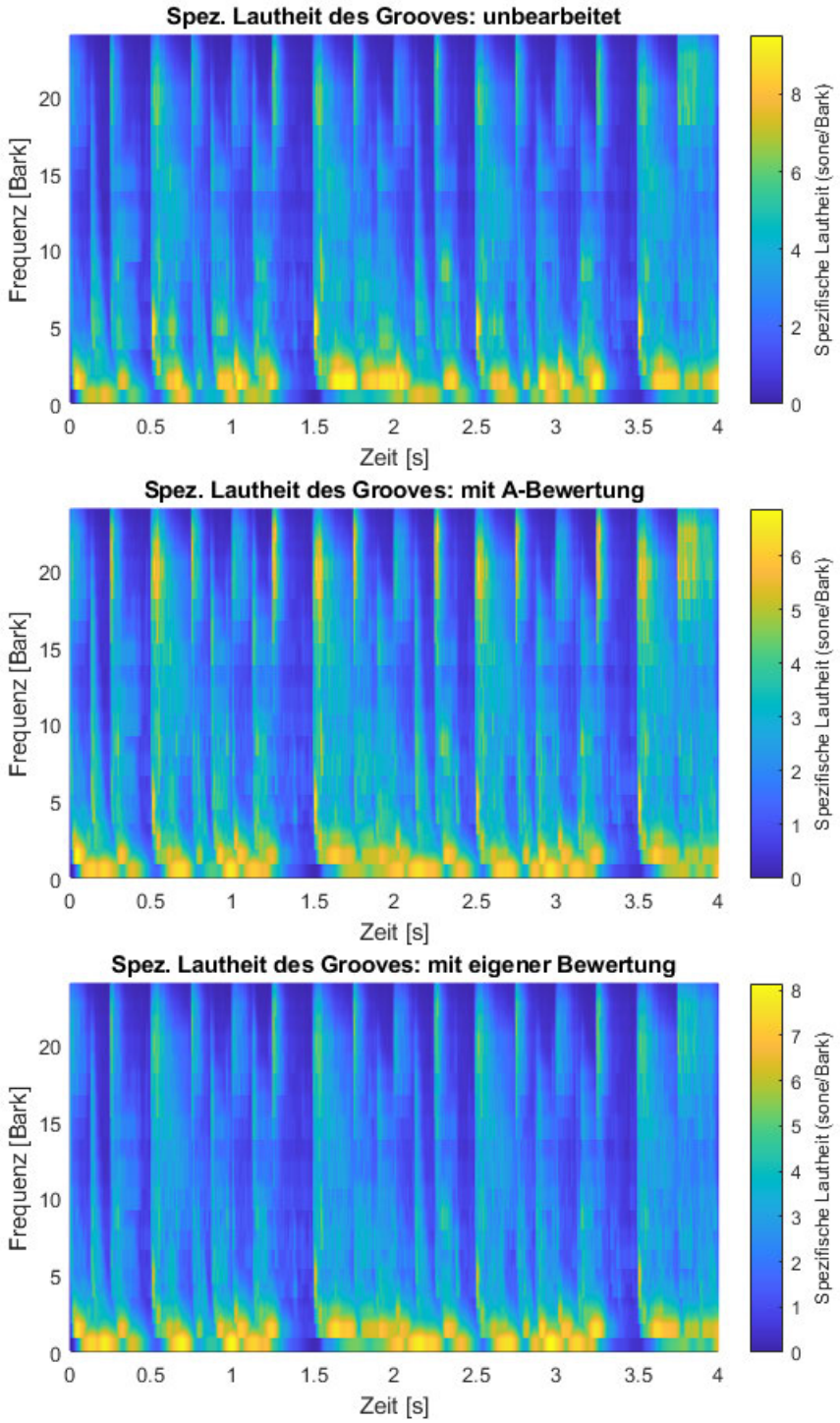


Abbildung C.2: Vergleich der Spezifischen Lautheiten des Grooves. Oben ist die unbearbeitete Aufnahme. Die Aufnahme in der Mitte wurde mit eingestellter A-Bewertung bearbeitet, die untere Aufnahme hingegen mit einer eigenen Bewertungsfunktion. (Eigene Darstellung)

C.2 Hörbeispiel Spezifische Lautheit 3: Gitarren-Loop

1) *BA_Drechsler_HBsp_SpezLautheit_3_Rohaufnahme.wav*
2) *BA_Drechsler_HBsp_SpezLautheit_3_A-Weighting.wav*
3) *BA_Drechsler_HBsp_SpezLautheit_3_CustomWeighting.wav*

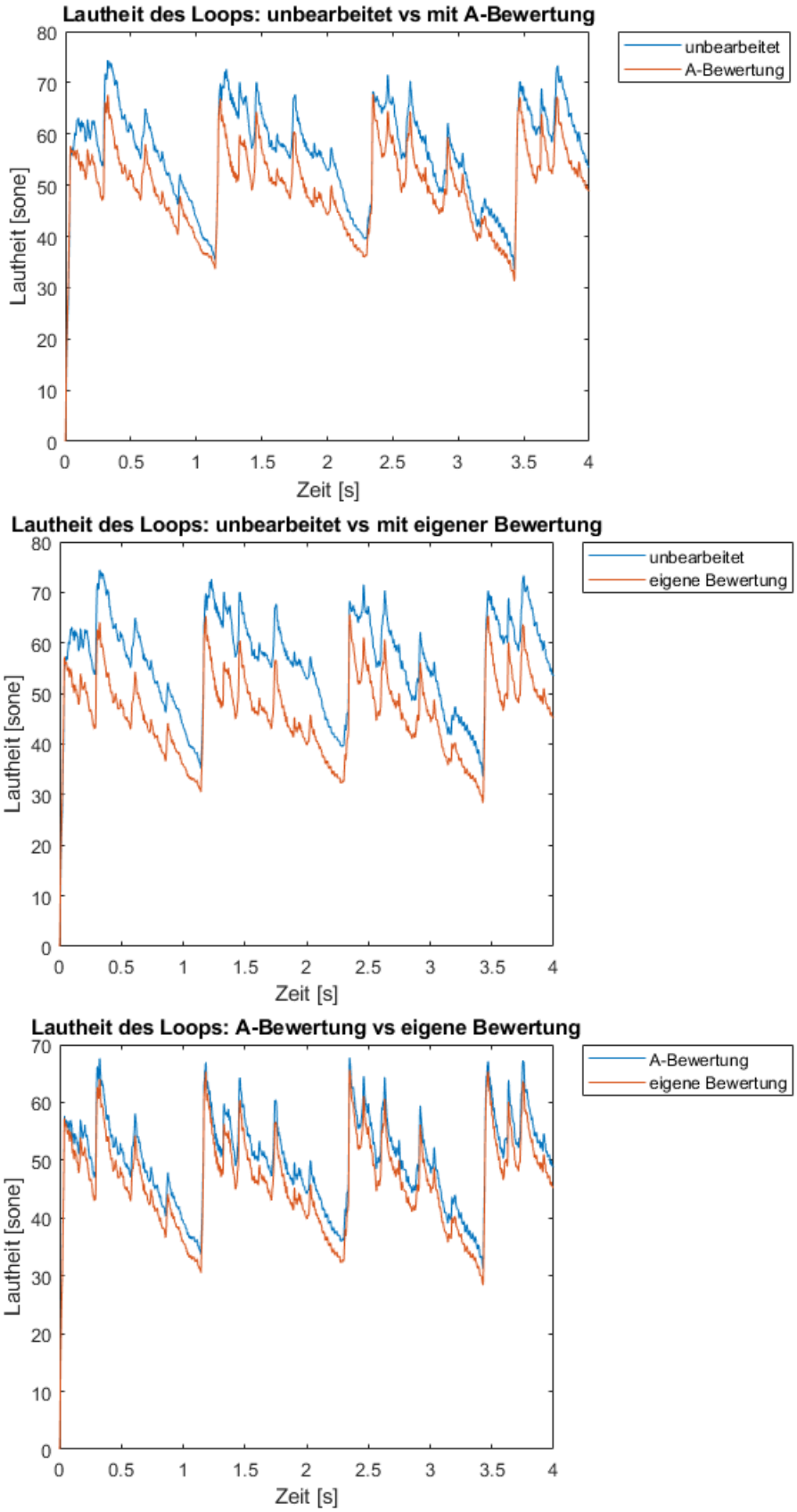


Abbildung C.3: Vergleich der Lautheiten des Loops. Unbearbeitet, mit Bearbeitung unter A-Bewertung und unter eigener Bewertung. (Eigene Darstellung)

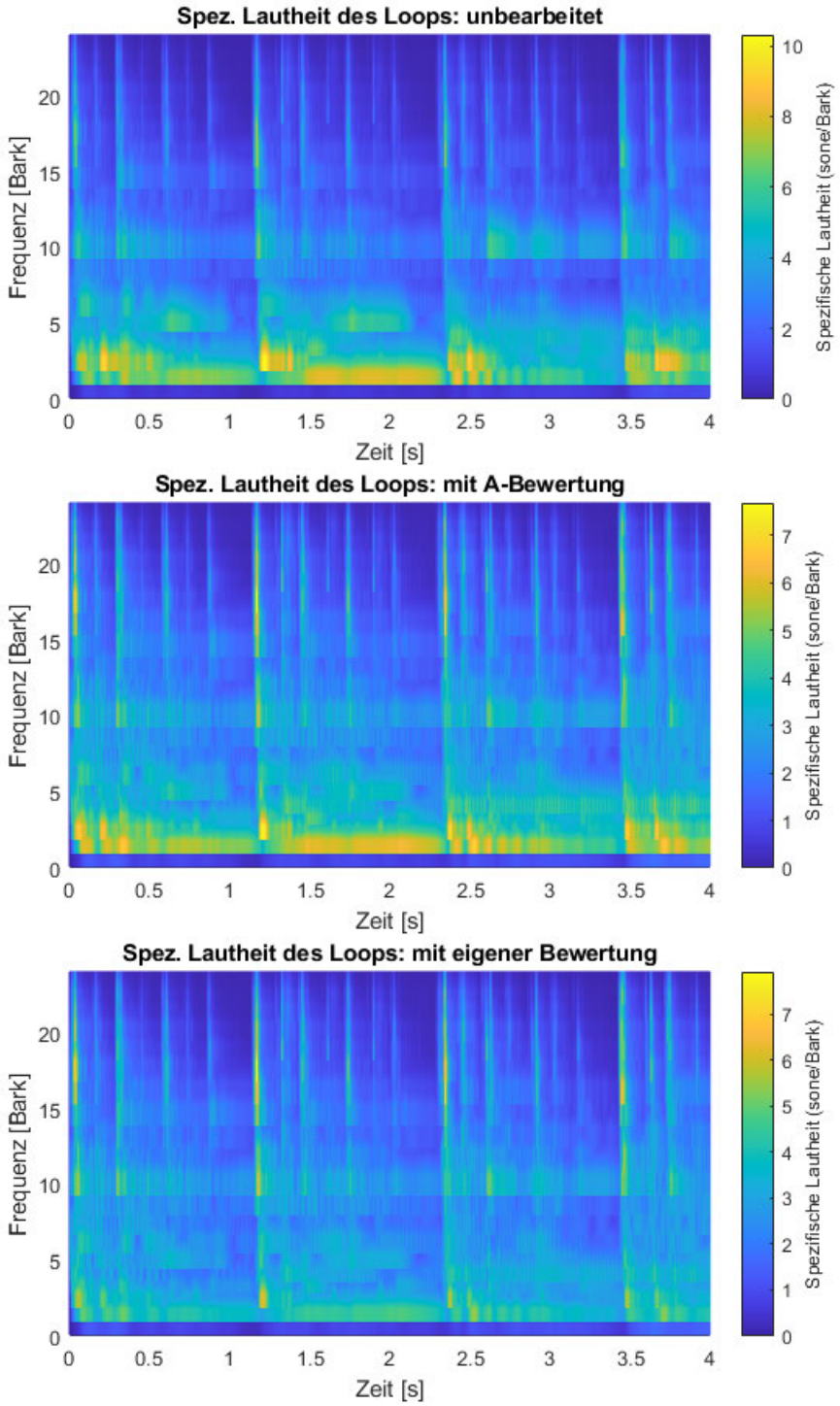


Abbildung C.4: Vergleich der Spezifischen Lautheiten des Loops. Oben ist die unbearbeitete Aufnahme. Die Aufnahme in der Mitte wurde mit eingestellter A-Bewertung bearbeitet, die untere Aufnahme hingegen mit einer eigenen Bewertungsfunktion. (Eigene Darstellung)

C.3 Hörbeispiel Spezifische Lautheit 4: Extremeinstellungen

1) *BA_Drechsler_HBsp_SpezLautheit_4_Rohaufnahme.wav*

2) *BA_Drechsler_HBsp_SpezLautheit_4_FlatExtreme.wav*

3) *BA_Drechsler_HBsp_SpezLautheit_4_A-WeightingExtreme.wav*

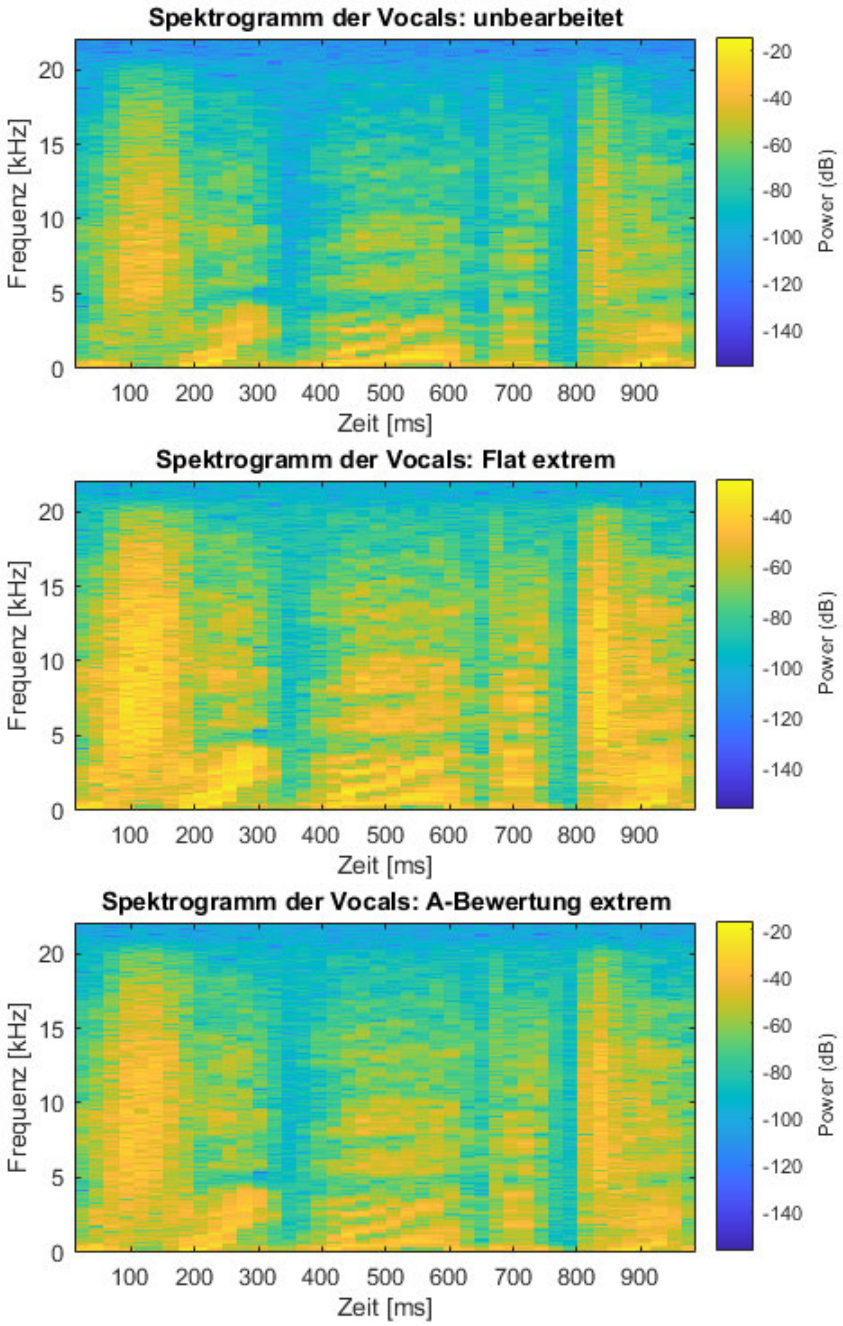


Abbildung C.5: Vergleich der Spektren der Vocals. Oben ist die unbearbeitete Aufnahme. Die Aufnahme in der Mitte wurde ohne Bewertung und mit Extremeinstellungen bearbeitet, die untere Aufnahme hingegen wurde mit eingestellter A-Bewertung und Extremeinstellungen bearbeitet. (Eigene Darstellung)

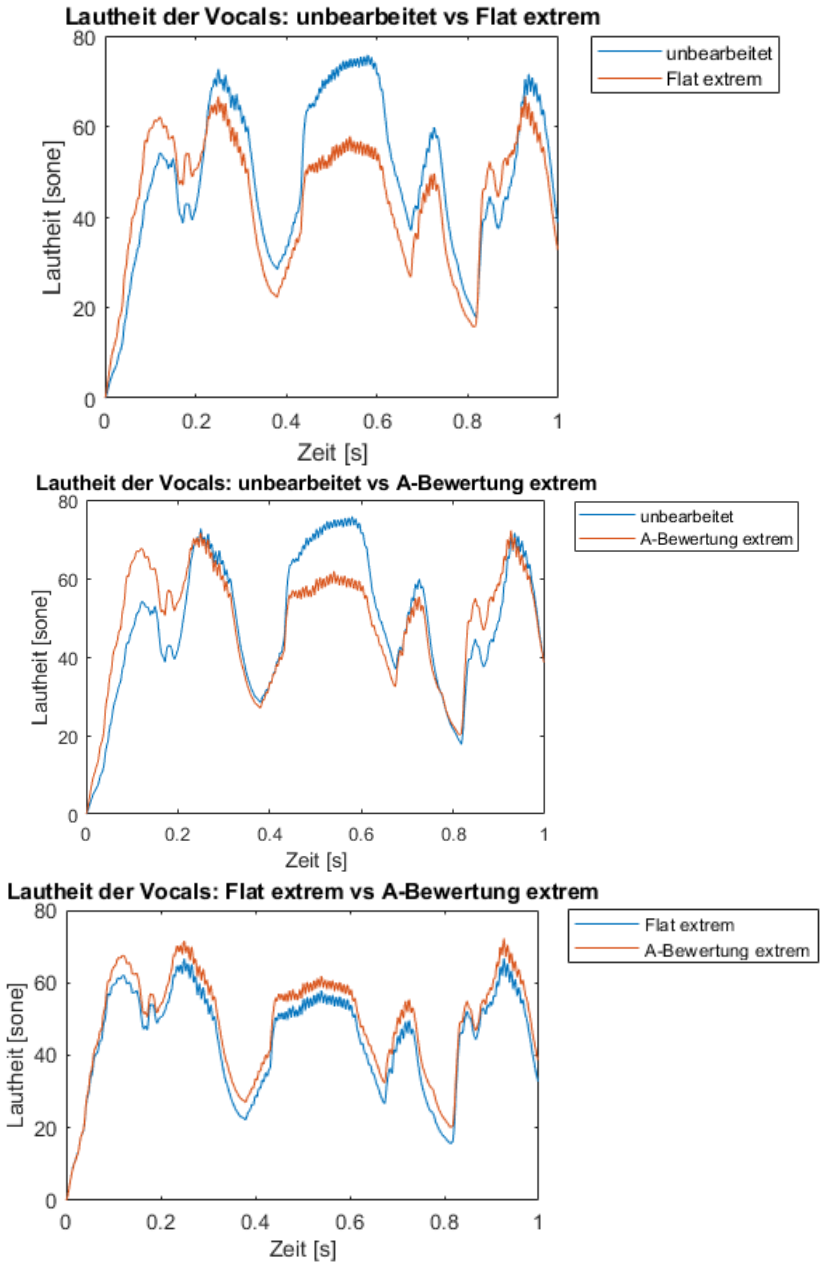


Abbildung C.6: Vergleich der Lautheiten der Vocals. Unbearbeitet, unter Extremeinstellungen ohne Frequenzbewertung und mit A-Bewertung. (Eigene Darstellung)

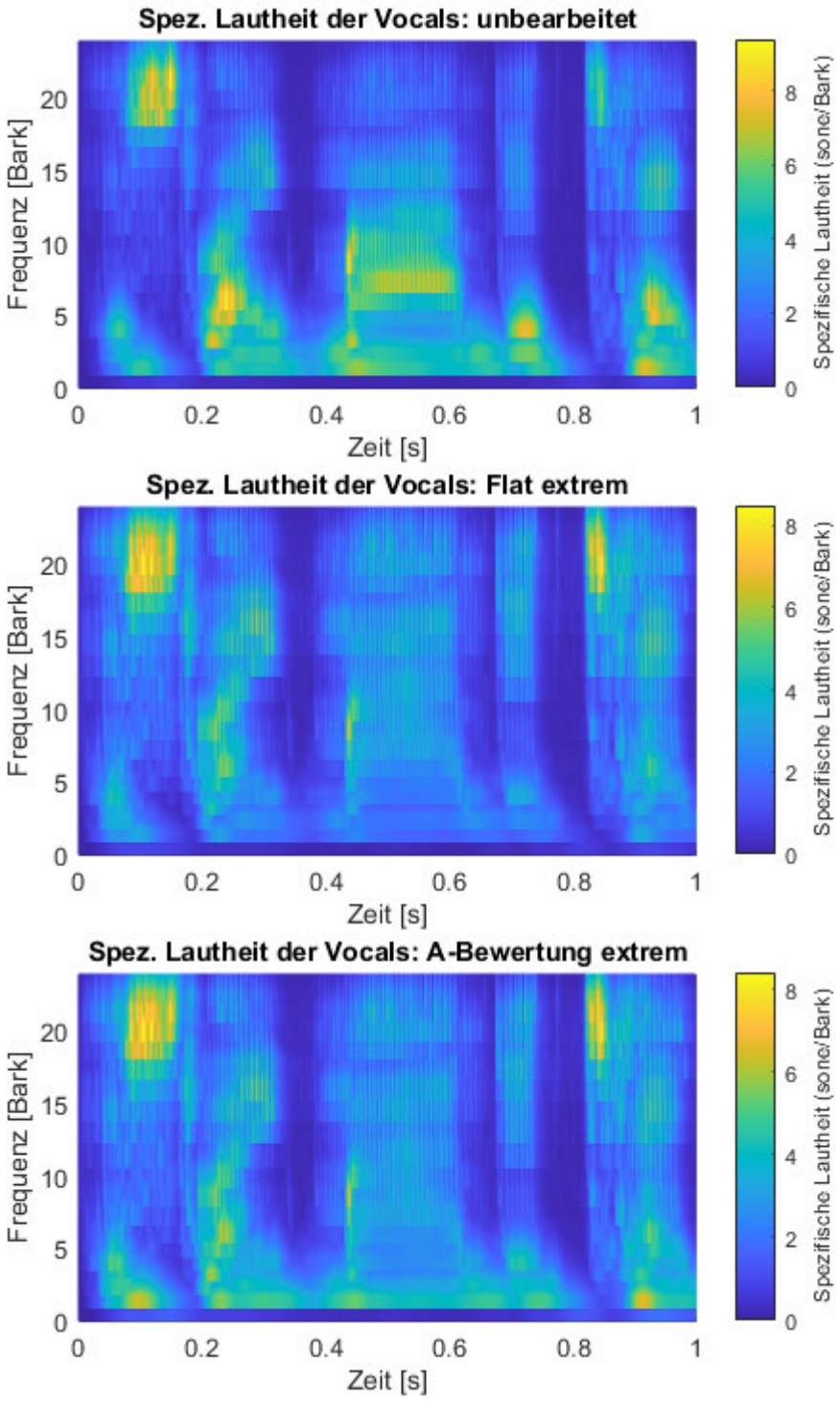


Abbildung C.7: Vergleich der Spezifischen Lautheiten der Vocals. Oben ist die unbearbeitete Aufnahme. Die Aufnahme in der Mitte wurde ohne Bewertung und mit Extremeinstellungen bearbeitet, die untere Aufnahme hingegen wurde mit eingestellter A-Bewertung und Extremeinstellungen bearbeitet. (Eigene Darstellung)

D Weitere Hörbeispiele zur Schärfe

Bei den hier gezeigten Beispielen handelt es sich um eigene Aufnahmen.

D.1 Hörbeispiel Schärfe 2: Frauenstimme

1) *BA_Drechsler_HBsp_Schaerfe_2_Rohaufnahme.wav*
2) *BA_Drechsler_HBsp_Schaerfe_2_EQ.wav*
3) *BA_Drechsler_HBsp_Schaerfe_2_Plugin.wav*

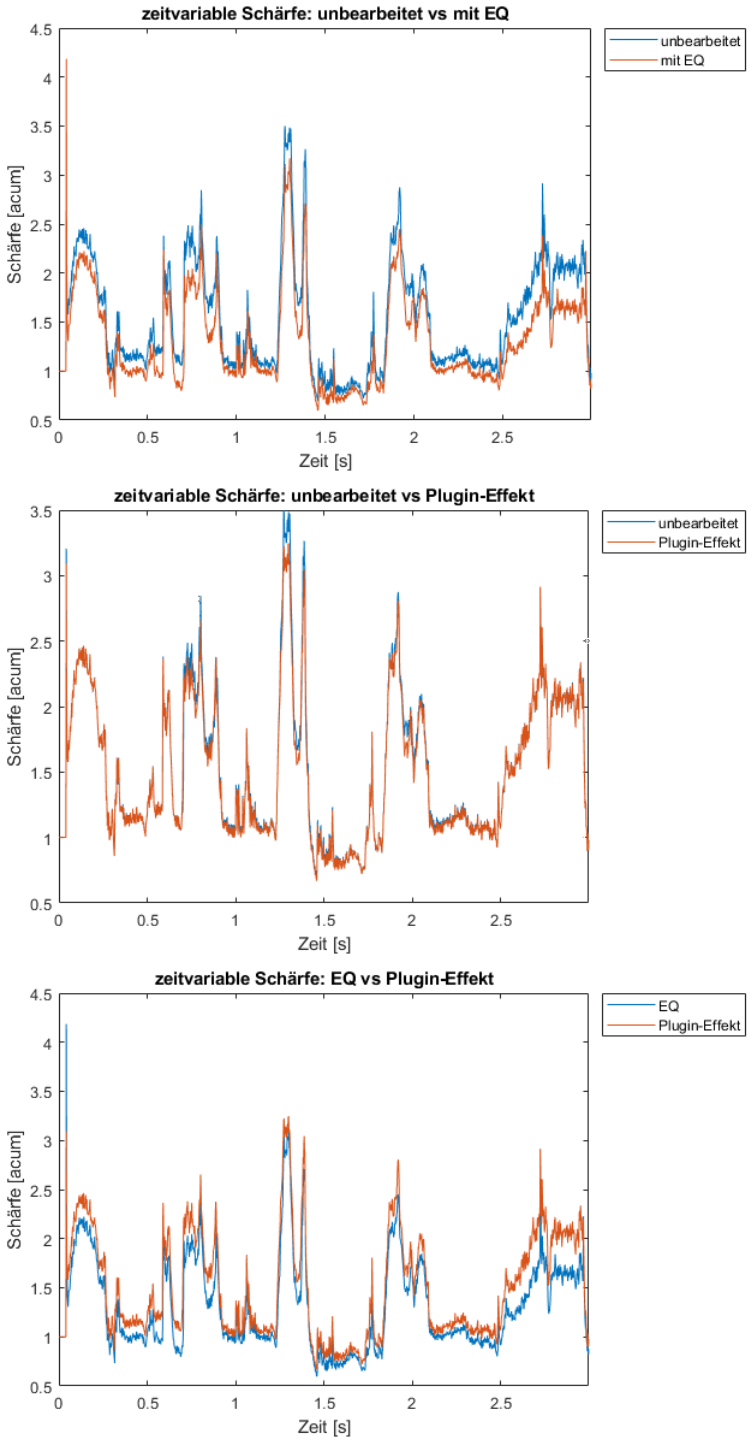


Abbildung D.1: Vergleich der Schärfen der bearbeiteten Sprachaufnahme (Frau). unbearbeitet, mit EQ und mit Plugin. (Eigene Darstellung)

D.2 Hörbeispiel Schärfe 3: Männerstimme

1) *BA_Drechsler_HBsp_Schaerfe_3_Rohaufnahme.wav*
2) *BA_Drechsler_HBsp_Schaerfe_3_EQ.wav*
3) *BA_Drechsler_HBsp_Schaerfe_3_Plugin.wav*

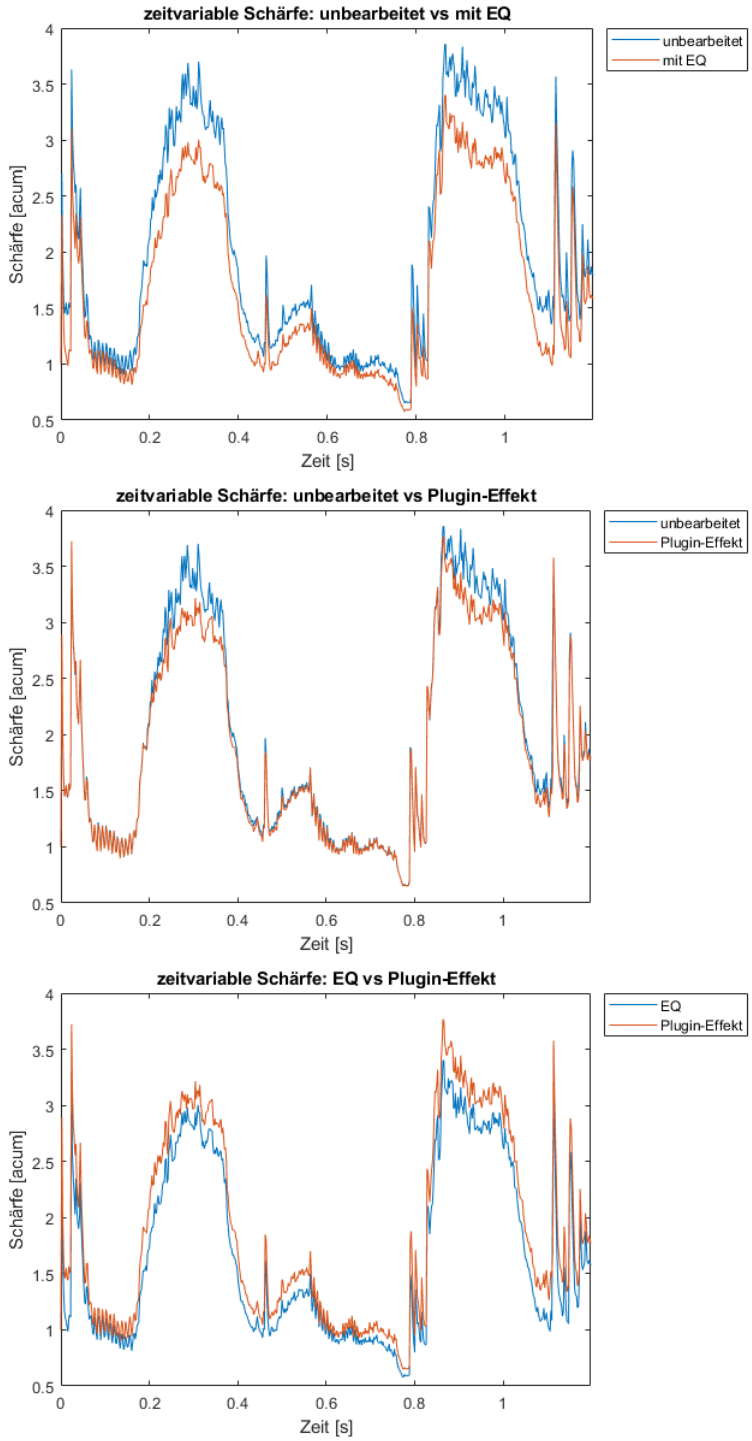


Abbildung D.2: Vergleich der Schärfen der bearbeiteten Sprachaufnahme (Mann). unbearbeitet, mit EQ und mit Plugin. (Eigene Darstellung)

E Gesamter GenExpr-Code im Frequenzbereich

```
Data DAT_filtcurve(fftsize / 2, 1);
Data DAT_freqwght(fftsize / 2, 7);
Data DAT_prevwindow(fftsize / 2, 2);

Buffer BUF_filtstats("---BUFFER_gen_eq_filter_stats");
Buffer BUF_filtupdate("---BUFFER_gen_filter_update");
Buffer BUF_drawredfilt("---BUFFER_draw_reduction_filters");
Buffer BUF_recordred("---recorded_frames_freqdomain1");
Buffer BUF_recordspec("---recorded_frames_freqdomain2");
Buffer BUF_correctedspec("---recorded_frames_freqdomain3");
Buffer BUF_resultspec("---recorded_frames_freqdomain4");
Buffer BUF_ampcorrcurve("---amp_correction_curve");
Buffer BUF_startrecord("---start_record");

History reset(1);
History filtrecalc(1);
History recording(0);

Param weightfct(3, min = 0, max = 6);
Param threshold(0., min = -48., max = 48.);
Param ratio(4., min = 1., max = 64.);
Param knee(6., min = 0., max = 36.);

Param attack(100., min = 0., max = 3000.);
Param release(1000., min = 0., max = 10000.);
Param env_mode(0, min = 0, max = 1);

Param dry_wet(1., min = 0., max = 1.);
Param winfct_index(0., min = 0., max = 7.);

// Initialize

CONST_bin_width = samplerate / fftsize;
CONST_wcompensation = 1.;

if(winfct_index == 0)
{
  CONST_wcompensation = 2.;
}
else if(winfct_index == 1)
{
  CONST_wcompensation = 1.;
}
else if(winfct_index == 2)
{
  CONST_wcompensation = 1.8625;
}
```

```
48 else if(winfct_index == 3)
49 {
50   CONST_wcompensation = 2.365;
51 }
52 else if(winfct_index == 4)
53 {
54   CONST_wcompensation = 2.8;
55 }
56 else if(winfct_index == 5)
57 {
58   CONST_wcompensationc = 4.56;
59 }
60 else if(winfct_index == 6)
61 {
62   CONST_wcompensation = 2.;
63 }
64 else if(winfct_index == 7)
65 {
66   CONST_wcompensation = 2.;
67 }
68
69
70 if(reset == 1)
71 {
72   for(i = 0; i < (fftsize / 2); i += 1)
73   {
74     bin_freq = i * CONST_bin_width;
75
76     wght_fct = 0.;
77
78     if(i == 0)
79     {
80       bin_freq = 10.;
81     }
82     bin_freq2 = bin_freq * bin_freq;
83     bin_freq3 = bin_freq2 * bin_freq;
84     bin_freq4 = bin_freq3 * bin_freq;
85     bin_freq5 = bin_freq4 * bin_freq;
86     bin_freq6 = bin_freq5 * bin_freq;
87
88     // Z-Weighting (No Weighting)
89     poke(DAT_freqwght, wght_fct, i, 0);
90
91     // Pink noise function
92     wght_fct = -0.5 * atodb(1. / bin_freq) - 30.;
93     poke(DAT_freqwght, wght_fct, i, 1);
94
95     // Tilt function
96     wght_fct = 4.5 * (log2(bin_freq) - log2(1000.));
```

```
97      poke(DAT_freqwght, wght_fct, i, 2);
98
99      // A-weighting function
100     wght_fct = atodb((12194. * 12194. * bin_freq4) / ((bin_freq2 + 20.6 *
            20.6) * sqrt((bin_freq2 + 107.7 * 107.7) * (bin_freq2 + 737.9 *
            737.9)) * (bin_freq2 + 12194 * 12194))) + 2.;
101     poke(DAT_freqwght, wght_fct, i, 3);
102
103     // C-Weighting
104     wght_fct = atodb((12194. * 12194. * bin_freq2) / ((bin_freq2 + 20.6 *
            20.6) * (bin_freq2 + 12194. * 12194.))) + 0.06;
105     poke(DAT_freqwght, wght_fct, i, 4);
106
107     // ITU-468 noise weighting
108     h1f = -4.737338981378384 * pow(10, -24) * bin_freq6 + 2.043828333606125
             * pow(10, -15) * bin_freq4 - 1.363894795463638 * pow(10, -7) *
            bin_freq2 + 1.;
109     h2f = 1.306612257412824 * pow(10, -19) * bin_freq5 - 2.118150887518656
            * pow(10, -11) * bin_freq3 + 5.559488023498642 * pow(10, -4) *
            bin_freq;
110     wght_fct = atodb((1.246332637532143 * pow(10, -4) * bin_freq) / sqrt(
            h1f * h1f + h2f * h2f)) + 18.2;
111     poke(DAT_freqwght, wght_fct, i, 5);
112
113     // K-Weighting
114     wght_fct = 0.;
115     k_phi = TWOPI * bin_freq / 48000.;
116     k_a0 = 1.;
117     k_a1 = -1.69065929318241;
118     k_a2 = 0.73248077421585;
119     k_b0 = 1.53512485958697;
120     k_b1 = -2.69169618940638;
121     k_b2 = 1.19839281085285;
122     k_A = k_b0 + k_b1 * cos(k_phi) + k_b2 * cos(2. * k_phi);
123     k_B = k_b1 * sin(k_phi) + k_b2 * sin(2. * k_phi);
124     k_C = k_a0 + k_a1 * cos(k_phi) + k_a2 * cos(2. * k_phi);
125     k_D = k_a1 * sin(k_phi) + k_a2 * sin(2. * k_phi);
126     k_denom = k_C * k_C + k_D * k_D;
127     k_Re = (k_A * k_C + k_B * k_D) / k_denom;
128     k_Im = (k_A * k_D - k_B * k_C) / k_denom;
129     wght_fct += atodb(k_Re * k_Re + k_Im * k_Im) / 2.;
130
131     k_a0 = 1.;
132     k_a1 = -1.99004745483398;
133     k_a2 = 0.99007225036621;
134     k_b0 = 1.;
135     k_b1 = -2.;
136     k_b2 = 1.;
137     k_A = k_b0 + k_b1 * cos(k_phi) + k_b2 * cos(2. * k_phi);
```

```
138     k_B = k_b1 * sin(k_phi) + k_b2 * sin(2. * k_phi);
139     k_C = k_a0 + k_a1 * cos(k_phi) + k_a2 * cos(2. * k_phi);
140     k_D = k_a1 * sin(k_phi) + k_a2 * sin(2. * k_phi);
141     k_denom = k_C * k_C + k_D * k_D;
142     k_Re = (k_A * k_C + k_B * k_D) / k_denom;
143     k_Im = (k_A * k_D - k_B * k_C) / k_denom;
144     wght_fct += atodb(k_Re * k_Re + k_Im * k_Im) / 2.;
145     poke(DAT_freqwght, wght_fct, i, 6);
146   }
147   reset = 0;
148 }
149
150 SIG_resg_inL = in1;
151 SIG_imsg_inL = in2;
152 SIG_resg_inR = in3;
153 SIG_imsg_inR = in4;
154 SIG_resc_inL = in5;
155 SIG_imsc_inL = in6;
156 SIG_resc_inR = in7;
157 SIG_imsc_inR = in8;
158
159 bin = in9;
160 w   = CONST_bin_width * bin; // frequency at the specified bin
161 if(bin == 0)
162 {
163   w = 10.;
164 }
165
166 // compensation for fftsize and analysis window energy loss
167 SIG_ampsc_dbL = atodb(CONST_wcompensation * 2. * sqrt(SIG_resc_inL *
      SIG_resc_inL + SIG_imsc_inL * SIG_imsc_inL) / fftsize);
168 SIG_ampsc_dbR = atodb(CONST_wcompensation * 2. * sqrt(SIG_resc_inR *
      SIG_resc_inR + SIG_imsc_inR * SIG_imsc_inR) / fftsize);
169
170 // Frequency weighting
171 ampcorrection_db = peek(DAT_freqwght, bin, weightfct);
172
173 Hs_abs_square   = 1.;   // filter transfer curve at the specified frequency
174 Hs_dB       = 0.;   // filter transfer curve at the specified frequency in
      Dezibel
175
176 // Additional EQ
177
178
179 if(bin == 0)
180 {
181   if(peek(BUF_filtupdate, 0, 0) == 1)
182   {
183     poke(BUF_filtupdate, 0, 0, 0);
```

```
    filtrecalc = 1;
  }
}

if(filtrecalc == 1)
{
  for(i = 0; i < dim(BUF_filtstats); i += 1.)
  {
    type   = peek(BUF_filtstats, i, 0);
    w0     = peek(BUF_filtstats, i, 1);
    gain   = peek(BUF_filtstats, i, 2);
    q_val    = peek(BUF_filtstats, i, 3);
    slope    = peek(BUF_filtstats, i, 4);
    enable   = peek(BUF_filtstats, i, 5);

    if(enable == 1)
    {
      wdw0   = w / w0;
      wdw02  = wdw0 * wdw0;
      A      = pow(10., (gain / 40.));
      N      = slope / 6.;
      uneven = round(N % 2.);

      if(type == 1)      // Highpass
      {
        Q = atan(0.1 * q_val) / atan(0.1);
        a = wdw02 - 1.;
        for(n = 0; n <= floor((N/2) - 1); n += 1.)
        {
          angle = 2. * PI * ((2. * n + N + 1.) / (4. * N));
          b = 2. * wdw0;

          if(n == 0)
          {
            b *= (cos(angle) / Q);
          }
          else
          {
            b *= cos(angle);
          }
          Hs_abs_square = (wdw02 * wdw02) / (a * a + b * b);
          Hs_dB += (10. * log10(Hs_abs_square));
        }
        if(uneven == 1)
        {
          Hs_abs_square = (wdw02) / (1. + wdw02);
          Hs_dB += (10. * log10(Hs_abs_square));
        }
      }
```

```
233       else if(type == 2)    // Lowpass
234       {
235         Q = atan(0.1 * q_val) / atan(0.1);
236         a = wdw02 - 1.;
237         for(n = 0; n <= floor((N/2) - 1); n += 1)
238         {
239           angle = 2. * pi * ((2. * n + N + 1) / (4. * N));
240           b = 2. * wdw0;
241
242           if(n == 0)
243           {
244             b *= (cos(angle) / Q);
245           }
246           else
247           {
248             b *= cos(angle);
249           }
250           Hs_abs_square = 1. / (a * a + b * b);
251           Hs_dB += 10. * log10(Hs_abs_square);
252         }
253
254         if(uneven == 1)
255         {
256           Hs_abs_square = 1. / (1. + wdw02);
257           Hs_dB += 10. * log10(Hs_abs_square);
258         }
259       }
260       else if(type == 3)    // Lowshelf
261       {
262         Q = atan(0.2 * q_val) / (atan(0.2) * SQRT2);
263         sqrtA = sqrt(A);
264         sqrtA2 = sqrtA * sqrtA;
265         a = wdw02 - sqrtA2;
266         b = wdw02 * sqrtA2 - 1.;
267         c = wdw0 * (sqrt(A) / Q);
268         Hs_abs_square = A * A * (a * a + c * c) / (b * b + c * c);
269         Hs_dB += 10. * log10(Hs_abs_square);
270       }
271       else if(type == 4)    // Highshelf
272       {
273         Q = atan(0.2 * q_val) / (atan(0.2) * SQRT2);
274         sqrtA = sqrt(A);
275         sqrtA2 = sqrtA * sqrtA;
276         a = wdw02 - sqrtA2;
277         b = wdw02 * sqrtA2 - 1.;
278         c = wdw0 * (sqrt(A) / Q);
279         Hs_abs_square = A * A * (b * b + c * c) / (a * a + c * c);
280         Hs_dB += 10. * log10(Hs_abs_square);
281       }
```

```
282       else if(type == 5)     // Bell
283       {
284         Q = q_val / SQRT2;
285         a = wdw02 - 1.;
286         b = wdw0 * (A / Q);
287         c = wdw0 / (A * Q);
288         Hs_abs_square = (a * a + b * b) / (a * a + c * c);
289         Hs_dB += 10. * log10(Hs_abs_square);
290       }
291       else if(type == 6)     // Notch
292       {
293         Q = q_val / SQRT2;
294         a = wdw02 - 1.;
295               b = wdw0 / Q;
296               Hs_abs_square = (a * a) / (a * a + b * b);
297         Hs_dB += 10. * log10(Hs_abs_square);
298       }
299       else if(type == 7)     // Bandpass
300       {
301         Q = q_val / SQRT2;
302         a = wdw02 - 1.;
303               b = wdw0 / Q;
304               Hs_abs_square = (b * b) / (a * a + b * b);
305         Hs_dB += 10. * log10(Hs_abs_square);
306       }
307     }
308   }
309   poke(DAT_filtcurve, Hs_dB, bin, 0);
310   if(bin == (fftsize/2 - 1))
311   {
312     filtrecalc = 0;
313   }
314 }
315 else
316 {
317   Hs_dB = peek(DAT_filtcurve, bin, 0);
318 }
319
320 weightedamp_dbL = SIG_ampsc_dbL + ampcorrection_db + Hs_dB;
321 weightedamp_dbR = SIG_ampsc_dbR + ampcorrection_db + Hs_dB;
322
323 // Compression algorithm
324
325 x_dbL = weightedamp_dbL;
326 x_dbR = weightedamp_dbR;
327 if(bin == 0)
328 {
329   x_dbL = -200.;
330   x_dbR = -200.;
```

```
331 }
332 x_scL = 0.;
333 x_scR = 0.;
334 g_outL = 0.;
335 g_outR = 0.;
336
337 t = threshold;
338 k = knee;
339 r = ratio;
340
341 if(k == 0.)
342 {
343   if(x_dbL < t)
344   {
345     x_scL = x_dbL;
346   }
347   else if(x_dbL >= t)
348   {
349     x_scL = t + ((x_dbL - t) / r);
350   }
351
352   if(x_dbR < t)
353   {
354     x_scR = x_dbR;
355   }
356   else if(x_dbR >= t)
357   {
358     x_scR = t + ((x_dbR - t) / r);
359   }
360 }
361 else if(k > 0.)
362 {
363   if(x_dbL < (t - (k / 2.)))
364   {
365     x_scL = x_dbL;
366   }
367   else if((x_dbL >= (t - (k / 2.))) && (x_dbL <= (t + (k / 2.))))
368   {
369     x_scL = x_dbL + ((((1. / r) - 1.) * (x_dbL - t + (k / 2.)) * (x_dbL - t
          + (k / 2.))) / (2. * k));
370   }
371   else if(x_dbL > (t + (k / 2.)))
372   {
373     x_scL = t + ((x_dbL - t) / r);
374   }
375
376   if(x_dbR < (t - (k / 2.)))
377   {
378     x_scR = x_dbR;
```

```
379   }
380   else if((x_dbR >= (t - (k / 2.))) && (x_dbR <= (t + (k / 2.))))
381   {
382     x_scR = x_dbR + ((((1. / r) - 1.) * (x_dbR - t + (k / 2.)) * (x_dbR - t
          + (k / 2.))) / (2. * k));
383   }
384   else if(x_dbR > (t + (k / 2.)))
385   {
386     x_scR = t + ((x_dbR - t) / r);
387   }
388 }
389
390 g_outL = x_scL - x_dbL;
391 g_outR = x_scR - x_dbR;
392
393 // envelope smoothing algorithm
394
395 g_inL = g_outL;
396 g_inR = g_outR;
397
398 g_sL = 0.;
399 g_sR = 0.;
400 g_prevL = peek(DAT_prevwindow, bin, 0);
401 g_prevR = peek(DAT_prevwindow, bin, 1);
402
403 Ta = (attack / 1000.) * (samplerate / (4. * ffthop));
404 Tr = (release / 1000.) * (samplerate / (4. * ffthop));
405 alpha_a = 1.;
406 alpha_r = 1.;
407
408 if(bin == 0)
409 {
410   g_sL = 0.;
411   g_sR = 0.;
412 }
413 else
414 {
415   if(env_mode == 0)              // Linear envelope
416   {
417     // prev = prev + in2 * (in1 - prev);
418     alpha_a = Ta >= 1. ? Ta : 1.;
419     alpha_r = Tr >= 1. ? Tr : 1.;
420
421     if(g_inL <= g_prevL)
422     {
423       g_sL = g_prevL + ((1. / alpha_a) * (g_inL - g_prevL));
424     }
425     else if(g_inL > g_prevL)
426     {
```

```
427       g_sL = g_prevL + ((1. / alpha_r) * (g_inL - g_prevL));
428     }
429
430     if(g_inR <= g_prevR)
431     {
432       g_sR = g_prevR + ((1. / alpha_a) * (g_inR - g_prevR));
433     }
434     else if(g_inR > g_prevR)
435     {
436       g_sR = g_prevR + ((1. / alpha_r) * (g_inR - g_prevR));
437     }
438   }
439   else if(env_mode == 1)            // Logarithmic envelope
440   {
441     alpha_a = Ta > 0. ? exp(-log10(9.) / Ta) : 0.;
442     alpha_r = Tr > 0. ? exp(-log10(9.) / Tr) : 0.;
443
444     if(g_inL <= g_prevL)
445     {
446       g_sL = (alpha_a * g_prevL) + ((1. - alpha_a) * g_inL);
447     }
448     else if(g_inL > g_prevL)
449     {
450       g_sL = (alpha_r * g_prevL) + ((1. - alpha_r) * g_inL);
451     }
452
453     if(g_inR <= g_prevR)
454     {
455       g_sR = (alpha_a * g_prevR) + ((1. - alpha_a) * g_inR);
456     }
457     else if(g_inR > g_prevR)
458     {
459       g_sR = (alpha_r * g_prevR) + ((1. - alpha_r) * g_inR);
460     }
461   }
462 }
463
464 poke(DAT_prevwindow, g_sL, bin, 0);
465 poke(DAT_prevwindow, g_sR, bin, 1);
466
467 poke(BUF_drawredfilt, ((g_sL + g_sR) * dry_wet / 2.), bin, 0);
468
469 redfilt_amplinL = dbtoa(g_sL * dry_wet);
470 redfilt_amplinR = dbtoa(g_sR * dry_wet);
471
472 if(bin == 0)
473 {
474   redfilt_amplinL = 1.;
475   redfilt_amplinR = 1.;
```

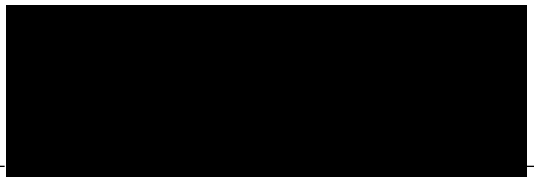
```
476 }
477
478 SIG_resg_outL = SIG_resg_inL * redfilt_amplinL;
479 SIG_imsg_outL = SIG_imsg_inL * redfilt_amplinL;
480 SIG_resg_outR = SIG_resg_inR * redfilt_amplinR;
481 SIG_imsg_outR = SIG_imsg_inR * redfilt_amplinR;
482
483 SIG_ampfinal_dbL = atodb(redfilt_amplinL * 4. * sqrt(SIG_resg_inL *
        SIG_resg_inL + SIG_imsg_inL * SIG_imsg_inL) / fftsize);
484 SIG_ampfinal_dbR = atodb(redfilt_amplinR * 4. * sqrt(SIG_resg_inR *
        SIG_resg_inR + SIG_imsg_inR * SIG_imsg_inR) / fftsize);
485
486 if(bin == 0)
487 {
488   if(peek(BUF_startrecord, 0, 0) == 1)
489   {
490     poke(BUF_startrecord, 0, 0, 0);
491     recording = 1;
492   }
493 }
494 if(recording == 1)
495 {
496   if((Hs_dB < -200.) || (ampcorrection_db < -200.))
497   {
498     poke(BUF_ampcorrcurve, -200., bin, 0);
499   }
500   else
501   {
502     poke(BUF_ampcorrcurve, ampcorrection_db + Hs_dB, bin, 0);
503   }
504 }
505 if(recording >= 1)
506 {
507   ovlp = int(fftsize / ffthop);
508   if(((recording - 1) % ovlp) == 0)
509   {
510     poke(BUF_recordred, (g_sL + g_sR) * dry_wet / 2., bin, (recording - 1)
          / ovlp);
511     poke(BUF_recordspec, (SIG_ampsc_dbL + SIG_ampsc_dbR) / 2., bin, (
          recording - 1) / ovlp);
512     poke(BUF_correctedspec, (weightedamp_dbL + weightedamp_dbR) / 2, bin, (
          recording - 1) / ovlp);
513     poke(BUF_resultspec, (SIG_ampfinal_dbL + SIG_ampfinal_dbR) / 2, bin, (
          recording - 1) / ovlp);
514   }
515   if(bin == (fftsize/2 - 1))
516   {
517     recording = recording + 1;
518     if(recording >= 4 * ovlp)
```

```
519     {
520       recording = 0;
521     }
522   }
523 }
524 
525 out1 = SIG_resg_outL;
526 out2 = SIG_imsg_outL;
527 out3 = SIG_resg_outR;
528 out4 = SIG_imsg_outR;
```

Quellcode E.1: GenExpr-Code des Patchers *DRS_DSP_SpectralComp.gendsp*

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.



Ort, Datum Kendy Drechsler