HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Adrian Alexander

Entwicklung eines Verzerrungs-Plugins mit MATLAB

2021

Fakultät: Medien

BACHELORARBEIT

Entwicklung eines Verzerrungs-Plugins mit MATLAB

Autor:
Herr Adrian Alexander

Studiengang:
Media and Acoustical Engineering

Seminargruppe:
MG-17-w-c

Erstprüfer:
**Prof. Dr.-Ing.
Jörn Hübelt**

Zweitprüfer:
**Dr. rer. nat.
Detlef Schulz**

Einreichung:
Ort, Datum

Faculty of Media

BACHELOR THESIS

Developement of a Distortion-Plugin with MATLAB

author:
Mr. Adrian Alexander

course of studies:
Media and Acoustical Engineering

seminar group:
MG-17-w-c

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

second examiner:
Dr. rer. nat.
Detlef Schulz

submission:
Ort, Datum

Bibliografische Angaben

Alexander, Adrian:

Entwicklung eines Verzerrungs-Plugins mit MATLAB

Developement of a Distortion-Plugin with MATLAB

60 Seiten, Hochschule Mittweida, University of Applied Sciences,
Fakultät Medien, Bachelorarbeit, 2021

Abstract

Ziel der Arbeit ist es den aktuellen Stand des Wissens zum Thema: „Erschaffung von Verzerrung als kreativen Effekt in Echtzeit-Audioprogrammen“ dazustellen, zu analysieren und anzuwenden. Es sollen Verzerrungsalgorithmen aus Fachliteratur kompiliert werden. Diese werden analysiert mit MATLAB. Die Analyse soll sich auf den Zeitbereich und den Frequenzbereich beziehen. Das Oberton verhalten soll charakterisiert werden. Die Analyse des Obertonverhaltens soll dazu dienen den Alias-Effekt des Algorithmus einzuschätzen und den Klang der verschiedenen Verzerrungsalgorithmen zu differenzieren. Es soll festgestellt werden, ob der Algorithmus für eine Echtzeit-Anwendung brauchbar ist und wie man ihn in ein Audioplugin implementieren kann. Ein Fokus liegt bei der Ansteuerung des Algorithmus mit verschiedenen Parametern wie z.B. Drive. Es soll ein VST-Plugin mit MATLAB erstellt werden. Dieses Plugin soll die für brauchbar befundenen Algorithmen anwenden. Es werden für die Anwendung spezifische Konzepte, wie z.B. die Überabtastung von Signalen, die Filterung von Signalen und andere Aspekte der Digitalen Signalverarbeitung vorgestellt und angewandt. Ein weiteres Ziel der Arbeit soll es sein, dem Leser das Objektorientierte Programmieren mit MATLAB näher zu bringen und brauchbaren Code zur Verfügung zu stellen. Der Leser wird in der Laage sein anhand dieser Arbeit ein Verzerrungs-plugin zu erstellen. Es soll besprochen werden, wie dieser Ansatz der Plugin-Erstellung weitergeführt werden kann. Die Motivation dieser Arbeit ist es einen Einstieg in die Digitale Signalverarbeitung mit MATLAB zu erlangen, anzuwenden und zu vermitteln.

Inhaltsverzeichnis

Inhaltsverzeichnis 2

Abkürzungsverzeichnis 4

Formelverzeichnis 7

Abbildungsverzeichnis 8

Tabellenverzeichnis 9

Einführung in das Thema „Verzerrung von Audiosignalen“ 10

1.1 Was ist Verzerrung? 10

1.2 Bedeutung von Verzerrung in der Musik 12

1.3 Stand der Verzerrungs-Technik 13

2 Analyse von Verzerrungsalgorithmen 15

2.1 Einleitung 15

2.2 Überblick des Analyseverfahrens 15

2.2.1 Analyse des Zeit- und Frequenzbereichs 15
2.2.2 Analyse des Aliasing-Effekts 16

2.3 Algorithmen 21

2.3.1 Hard Clipping 21
2.3.2 Gleichrichtung 24
2.3.3 Araya und Suyama 25
2.3.4 Diodic et al. 27
2.3.5 Bendiksen 29
2.3.6 Diode 31
2.3.7 kubisches Softclipping 31
2.3.8 tangentiales Softclipping 34

2.4 Vergleich der Verzerrungsalgorithmen 36

2.5 Fazit der Analyse der Algorithmen 39

3 Erstellung von Audio-Plugins in MATLAB 41

3.1 Einführung MALTAB objektorientierte Programmierung 41

3.1.1 Warum objektorientierte Programmierung? 41
3.1.2 Verwendete Toolboxen und Versionsnummern 42
3.1.3 Grundgerüst für ein Audio-Plugin 43

3.2 Grundkonzepte der Audioprogrammierung 47

3.2.1 Umgang mit variablen Eingangsgrößen 47
3.2.2 Elementweise Signalverarbeitung 48

3.2.3 Implementierung der Überabtastung 50
3.2.4 Implementierung von simplen Filtern 52
3.2.5 Anpassung der Signalstärke 53
3.2.6 Mischen von zwei Signalen 54
3.3 Vorstellung des finalen Verzerrungs-Plugins 55
3.3.1 Schaltplan 55
3.3.2 Das finale User-Interface 56
3.3.3 Der Klang 57
3.3.4 Leistungsoptimierung 58
3.3.5 Verbesserungsansätze 59
3.3.6 Wie könnte es weiter gehen? 59
4 Fazit 60
Literaturverzeichnis XIV
Anlagen XVII
1. verwendeter Code aus Quellen XVII
1.1. Verzerrungs-Algorithmen XVII
1.1.1. Hard-Clip-Algorithmus (Tarr 2018) XVII
1.1.2. Limitierung von positiven Halbwellen XVIII
1.1.3. Röhrenemulation (Araya und Suyama 1996) XVIII
1.1.4. Röhrenemulation (Doidic et al. 1998) XIX
1.1.5. Röhrenemulation (Zölzer et al. 2002) XX
1.1.6. Diode (Shockley 1949; Tarr 2018) XXI
1.1.7. Gleichrichter XXI
1.1.8. Kubisches Softclipping (Smith III 2008; Sullivan 1990) XXII
1.1.9. Tangentiales Softclipping XXIII
1.2. Verzerrungs-Plugin XXIII
1.2.1. Oversampling XXIII
1.2.2. Finales Plugin XXV
Eigenständigkeitserklärung XXXVI

Abkürzungsverzeichnis

DAW (Digital Audio Workstation)

Die DAW ist eine Software in der Audiospuren, aufgezeichnet, arrangiert und bearbeitet werden können. Es kann externe Software in der DAW geladen werden. Diese Software ist als Plugin bekannt und erweitert den Funktionsumfang der DAW.

Samples

Samples sind numerische Daten, welche durch die Abtastung eines Signals erzeugt werden. Wenn ein Signal beispielsweise mit 44,1 kHz abgetastet wird, so entstehen 44100 Samples pro Sekunde. Diese numerischen Daten sind die Werte des Amplitudenganges des Zeitsignals. Samples existieren nur in der digitalen Domäne. Ihr Wert muss sich zwischen -1 und 1 am Ausgang eines Audioplugins befinden. Eine Sinusschwingung mit einem Maximum bei 1 und einem Minimum bei -1 würde einen Pegel von 0 dBFS erzeugen.

dBFS

Decibles relative to full scale (engl.) ist die Einheit, welche in einer DAW verwendet wird, um den Pegel anzugeben. 0 dBFS ist der maximale Pegel. Jedes Signal, welches diesen Pegel überschreitet, wird digital verzerrt, diese Verzerrung wird auch Hard-Clipping genannt. In einer DAW wird immer die Differenz aus Signalpegel und Maximalpegel verwendet. Beispielsweise könnte ein Instrument einen Pegel von -6 dBFS aufweisen.

Oversampling (Überabtastung)

Die Abtastfrequenz wird um einen ganzzahligen Faktor erhöht. Dadurch entstehen mehr Samples im Zeitsignal und es können höhere Frequenzen wiedergegeben werden. Die Vergrößerung der Bandbreite des Frequenzbereichs verringert den Alias-Effekt. Nach dem eine digitale Signalverarbeitung stattgefunden hat, wird das Signal durch Unterabtastung wieder auf seine ursprüngliche Datenrate, Samples pro Sekunde, reduziert.

Distortion (Verzerrung)

Durch eine Veränderung des Zeitsignales werden dem Signal harmonische Obertöne hinzugefügt.

Alias-Effekt

Frequenzinhalte die die Nyquist-Frequenz überschreiten werden an dieser gespiegelt. Dies resultiert in einer Erschaffung neuer Frequenzkomponenten, die nicht in harmonischen Zusammenhang mit dem Ursprungsmaterial stehen müssen. Diese neuen Frequenzkomponenten werden deswegen häufig als störend oder unmusikalisch empfunden.

Nyquist-Frequenz

Die Hälfte der Abtastfrequenz ist die Nyquist-Frequenz. Frequenzen des Audiosignals, welche die Nyquist-frequenz überschreiten, werden an dieser gespiegelt. Wenn die Abtastfrequenz gleich 44100 Hz ist, dann ist die Nyquist-Frequenz gleich 22050. Eine Sinusschwingung, welche z.B. durch eine Verzerrung, bei 30 kHz, entsteht würde, entsteht stattdessen bei 14100 Hz. Die Schwingung wird an der Nyquist-Frequenz gespiegelt. Die gespiegelte Schwingung entsteht bei der Differenz der Nyquist-Frequenz und der Frequenz des Eingangssignals, addiert mit der Nyquist-Frequenz. Dieses Phänomen ist als Alias-Effekt bekannt.

22050 Hz – 30000 Hz + 22050 Hz= 14100 Hz

Durch eine Erhöhung der Abtastfrequenz durch Überabtastung steigt auch die Nyquist-Frequenz. Dadurch kann erreicht werden, das gespiegelte Frequenzen nicht im hörbaren Bereich, zwischen 20 und 20 kHz, entstehen. Das gleiche kann mit einem Antialiasing-Filter erreicht werden. Dies muss ein Tiefpass-Filter sein, welcher so dimensioniert ist, dass alle Frequenzen unter 20 kHz nicht verändert werden. Alle Frequenzen, die über 20 kHz liegen müssen vom Filter stark gedämpft werden.

Plugin

Funktionserweiterung für eine DAW. Ein Plugin ist kein eigenständiges Programm, es funktioniert aber in einer DAW. Ein Plugin kann auf eine Spur geladen werden, um den jeweiligen Klang zu verändern. Es erhält das Signal von der DAW, verändert es und gibt es wieder an die DAW zurück. In dieser Arbeit wird ein solches Audioeffekt-Plugin erschaffen. Plugins können aber auch Analysatoren, Oszillatoren und anderes sein.

Mix

Bereits aufgezeichnete musikalische Komposition in einer DAW. Beim „Mixing“ wird der Pegel der einzelnen Audiospuren gegeneinander gewichtet. Ziel ist es alle Instrumente hörbar zu machen und dadurch die Komposition klanglich aufzuwerten. Neben der Anpassung der Lautstärke kommen auch häufig Equalizer und Verzerrungen zum Einsatz.

Der Zeit- und Frequenz-Bereich, von jedem Element in der musikalischen Komposition wird adäquat verändert.

Waveshaping

Veränderung der Wellenform durch Transferkennlinien. Mit dieser Technik werden viele Verzerrungs-Algorithmen umgesetzt.

Sustain

Das Anhalten eines Tones. Bei einem Klavier beschreibt der Sustain den Verlauf der Amplitude des Klanges, während eine Taste gedrückt gehalten wird.

FFT

Mathematische Operation, welche es ermöglicht ein Frequenzgemisch in einzelne Sinusschwingungen zu zerlegen. Ein FFT-Analysator stellt für die jeweilige Frequenz die Amplitude dar. Dadurch kann der Frequenzbereich eines Signales verstanden werden.

maxAlias

Die Frequenz mit der größten Amplitude, welche durch den Alias-Effekt, in den hörbaren Bereich, zwischen 20 Hz und 20 kHz, gespiegelt wurde. Diese Messgröße wird in der Doktorarbeit von Yeh (2009) verwendet um den Aliasing-Effekt zu messen. Dies entspringt der Annahme das die gespiegelte Frequenz mit der höchsten Amplitude wahrscheinlich auch am besten wahrgenommen werden kann und somit auch am meisten störend sei. Die Messgröße ist einfach zu messen und ist effektiv darin den Einfluss von verschiedenen Überabtastungsfaktoren auf den Aliasing-Effekt einzuschätzen

Formelverzeichnis

Formel 1 Symmetrieeigenschaften einer gerade Funktion 11
Formel 2 Symmetrieeigenschaften einer ungeraden Funktion 11
Formel 3 Hard-Clipping-Algorithmus (Tarr 2018) 22
Formel 4 Hard Clipping der positiven Halbwelle 23
Formel 5 Röhrenverzerrungs-Algorithmus (Araya und Suyama 1996) 26
Formel 6 asymmetrischer Röhrenverzerrungs-Algorithmus (Doidic et al. 1998) 28
Formel 7 Röhrenverzerrungs-Algorithmus (Bendiksen 1997) 29
Formel 8 kubisches Softclipping (Sullivan 1990; Smith III 2008) 32
Formel 9 Modifizierung des kubisches Softclipping 33
Formel 10 hyperbolischer Tangens als Verzerrer 34
Formel 11 Arkustangens als Verzerrer 36

Abbildungsverzeichnis

Abbildung 1Alias-Effekt einer 1kHz Rechteckschwingung mit zweifacher Überabtastung 18
Abbildung 2 Alias-Effekt einer 1kHz Rechteckschwingung ohne Überabtastung 18
Abbildung 3 Alias-Effekt einer 20 kHz Rechteckschwingung mit 32-facher Überabtastung 19
Abbildung 4 Analyse des Hard-Clipping-Algorithmus....................22
Abbildung 5 Analyse des Algorithmus für positive Halbwellen-Limitierung....................23
Abbildung 6 Analyse des Halbwellengleichrichters....................24
Abbildung 7 Analyse des Vollwellengleichrichters 25
Abbildung 8 Analyse des Röhrenverzerrers von Araya und Suyama (1996)...............26
Abbildung 9 Analyse des Röhrenverzerrers von Araya und Suyama (1996) in Reihenschaltung 27
Abbildung 11 Analyse des asymmetrischen Röhrenverzerrers aus Diodic et al. (1998) 28
Abbildung 12 Röhrenverzerrung nach Bendiksen Q=-0.01 Dist= 100....................30
Abbildung 13 Röhrenverzerrung nach Bendiksen Q=-0.75 Dist= 10....................30
Abbildung 14 Dioden-Verzerrung (Shockley 1949; Tarr 2018)....................31
Abbildung 15 kubisches Softclipping (Smith III 2008; Sullivan 1990) 32
Abbildung 16 Modifizierung des kubisches Softclipping für a = 3....................33
Abbildung 18 Modifizierung des kubisches Softclipping für a = 2....................33
Abbildung 17 Modifizierung des kubisches Softclipping für a = 2.5....................34
Abbildung 19 hyperbolischer Tangens als Verzerrer 35
Abbildung 20 Approximation des hyperbolischen Tangens als Verzerrer 35
Abbildung 21 Arkustangens als Verzerrer mit Drive = 100....................36

Tabellenverzeichnis

Tabelle 1 Erklärung der Messung und Beurteilung des Alias-Effekts 21
Tabelle 2 vergleichende Darstellung der Verzerrungsalgorithmen 36

Einführung in das Thema „Verzerrung von Audiosignalen“

1.1 Was ist Verzerrung?

„Verzerrung (engl. Distortion) [ist die] […] Sammelbezeichnung für jede Art der Klangverfälschung oder Klanggestaltung eines zu verstärkenden, zu übertragenden oder aufzuzeichnenden Signals.“ (Enders 1985)

Verzerrung beschreibt eine spezifische Änderung eines Signals, welche die Wellenform des Signals verändert und neue Frequenzkomponenten hinzufügt. Verzerrung kann laut Temme (1992) in zwei Kategorien eingeteilt werden. Lineare Verzerrung und nicht lineare Verzerrung. Bei der linearen Verzerrung entspricht das Ausgangssignal dem Eingangssignal. Es entsteht keine Veränderung der Wellenform und oder der Frequenzkomponenten. Echo ist eine Lineare Verzerrung (Ciciora et al. 2004). Wenn das zeitlich versetzte Signal, mit dem nicht versetzten Eingangssignal gemischt wird, so entstehen starke Änderungen im Zeit- und Frequenzbereich. Diese Technik wird im sehr bekannten Karplus-Strong-Algorithmus zur digitalen Synthese von Saiten- und Trommelgeräuschen verwendet (Karplus und Strong 1983). Der Algorithmus wurde erweitert um mit diesem eine Gitarrenverzerrung zu ermöglichen (Sullivan 1990). Der Fokus dieser Arbeit liegt allerdings auf nicht linearer Verzerrung.

„Distortion occurs whenever the input/ output transfer function alters the waveform of a signal […]“ (Temme 1992)

„AN AUDIO [sic] transmission system that is nonlinear will introduce distortion into a signal passing through it. A characteristic of nonlinear distortion is that new frequency components, that were not present in the input signal, are created in the output signal.“ (Maré 2002)

Nicht lineare Verzerrung erzeugt Obertöne und verändert die Wellenform. Diese Form der Verzerrung kann laut Otala (1977) in statische- und dynamische nicht lineare Verzerrung unterteilt werden. Laut ihm ist das Frequenzverhalten der statischen nicht linearen Verzerrung allein von dem zeitlichen Verlauf der Amplitude des Signals abhängig. Dynamische Verzerrung tritt laut ihm auf, wenn der Frequenzinhalt des Eingangssignals die Transferfunktion des Systems beeinflusst. Statische nicht lineare Verzerrung wird auch als Waveshaping bezeichnet (Roads 1979). Diese wird in der Musiktechnik häufig verwendet. Sie wird verwendet, weil sie einfach anzuwenden ist und verlässlich funktioniert. Einfache Algorithmen benötigen wenig Rechenleistung und erzeugen eine niedrige

Latenz. Je größer die Amplitude des Signals umso mehr Obertöne werden erzeugt. Dieses Verhalten ist besonders aus musikalischer Sicht sinnvoll. Der Gitarrist ist dieses Verhalten von analogen Verstärkern gewöhnt und empfindet es somit als organisch und damit glaubwürdig. Die musikalische Anwendung von nicht linearer Verzerrung ist vielseitig. Sie kann als Gitarrenverzerrung verwendet werden. Sie kann aber auch dafür verwendet werden, um eine Wellenform in eine andere umzuwandeln. Man kann, zum Beispiel, sehr einfach eine Sinusschwingung in eine Rechteckschwingung umwandeln. Nicht lineare Verzerrung können am besten mit einer Transferfunktion beschrieben werden. Dies ist auch die einfachste Möglichkeit, um diese in ein Audio-Plugin zu implementieren. Wenn eine Sinusschwingung so verzerrt wird, entstehen höhere harmonische Obertöne (Guicking 2016). Das bedeutet, dass weitere Sinusschwingungen entstehen, deren jeweilige Frequenz ein ganzzahliges vielfaches der Frequenz des Eingangssignales sind. Eine Verzerrung einer einzelnen Sinusschwingung erzeugt entweder gerade, ungerade oder gerade und ungerade Vielfache der verzerrten Sinusschwingung (Temme 1992). Die Transferfunktion hat einen großen Einfluss darauf welche Vielfache entstehen. Laut Temme (1992) und Roads (1979) erzeugen gerade Transferfunktionen gerade Obertöne und ungerade Transferfunktion ungerade Obertöne. Eine Funktion ist gerade, wenn diese symmetrisch zur y-Achse ist. Eine Funktion ist ungerade, wenn sie zur y-Achse antisymmetrisch ist (Lindner et al. 2008).

$$fg(t) = fg(-t)$$

Formel 1 Symmetrieeigenschaften einer gerade Funktion

$$fu(t) = -fu(-t)$$

Formel 2 Symmetrieeigenschaften einer ungeraden Funktion

Wenn das Eingangssignal aus mehr als einer Sinusschwingung besteht, treten Kombinationsfrequenzen auf. Dies ist als Intermodulationsverzerrung bekannt.

“ Diese Kombinationsfrequenzen ergeben nur dann ein harmonisches Spektrum, wenn die Primärtöne harmonisch zueinander liegen […].“ (Guicking 2016)

Viele Signale in der Musikproduktion sind komplexe Klänge oder Geräusche, welche teilweise Rauschen enthalten wie z.B. ein Schlagzeug. Solche Signale sind von der Intermodulationsverzerrung stark betroffen. Die Intermodulationsverzerrung verstärkt in dem Fall den rauschartigen Charakter des Schlagzeuges. Die Harmonischen Obertöne der Verzerrung gelten als musikalisch und sind erwünscht. Die Intermodulationsprodukte sind eher ein Nebenprodukt und gelten als unmusikalisch (ebd.).

1.2 Bedeutung von Verzerrung in der Musik

Verzerrung tritt bei vielen verschiedenen technischen Prozessen, als unerwünschtes neben Produkt auf. Für gewöhnlich wird großer Entwicklungsaufwand betrieben, um möglichst wenig Verzerrung in einem System zu erzeugen (Temme 1992). In der Musikelektronik wird die Verzerrung allerdings verwendet, um ein Signal nachträglich zu verändern und dieses dadurch künstlerisch aufzuwerten. Das Verwenden von Verzerrung in der Musik erschafft somit neue klangliche Vielfalt. Im Rock and Roll und im Heavy Metall ist die Verzerrung der E-Gitarren ein wichtiges stilistisches Mittel (Fifka 2019; Elflein 2014). Zusätzlich zu der Veränderung des Gitarrenklanges hat die Verzerrung in diesen Genres auch einen starken Einfluss auf die Spielweise des Instruments. Bestimmte Spieltechniken profitieren von der Verzerrung, da die Verzerrung das Signal komprimiert, mehr Sustain hinzufügt und die Hüllkurve verändert (Herbst 2017). Auch in anderen Genres findet Verzerrung breitflächige Anwendung. Vor allem in der elektronischen Musik, in der Synthesizer häufig verwendet werden, ist Verzerrung ein beliebtes Mittel für die klangliche Gestaltung (Dixon 2018). Es können nur wenige Obertöne einem Signal hinzugefügt werden oder sehr viele. Beide Fälle finden in der Musik Anwendung. Was dabei als wohlklingend wahrgenommen wird und was nicht ist sehr subjektiv. Die Anwendung von Verzerrung kann grob in zwei Kategorien eingeteilt werden. Entweder wird sie verwendet, um eine subtile Obertonanreicherung an einem bereits musikalisch hochwertigen Signal vorzunehmen, oder sie soll das Signal so stark verändern, dass dabei ein neuer musikalischer Kontext entsteht. Das bedeutet, dass Verzerrungsalgorithmen, die nur einen harmonischen Oberton erzeugen sinnvoll sind. Das bedeutet aber auch, dass ein Algorithmus, der die Frequenzkomponenten des Eingangssignals fast vollkommen verändert, aus musikalischer Sicht einen nutzen hat. Verzerrung ist des Weiteren ein wichtiger Bestandteil des Mixing und Mastering Prozesses eines Songs und wird dort Genre übergreifend vielseitig verwendet (Savage 2014). In diesem Anwendungsgebiet soll das bereits aufgezeichnete musikalische Werk den letzten feinschliff erhalten. Laut Bonanno (2019) gibt es drei verschiedene Möglichkeiten Verzerrung in einem solchen Scenario anzuwenden. Die erste ist, das Hinzufügen von wenigen Obertönen, um gezielt mehr hohe Frequenzen in das Audiosignal zu bringen. Manche Instrumente können so besser wahrgenommen werden. Der zweite Anwendungsfall macht sich den Einfluss der Verzerrung auf die Transienten des Signales zu nutze. Mit zunehmender Verzerrung sinkt die Amplitude der Transienten. Das Resultat ist ein höherer Effektivwert des Audio-Signals (Volans 2015). Dadurch wird ein Song, welcher sorgfältig mit den Spitzenreduzierenden Eigenschaften der Verzerrung behandelt wurde lauter als der gleiche Song ohne Verzerrung. Verzerrung ist somit fester Bestandteil des sogenannten „Loudness War“.

„The clipping distortion can increase the harmonic content of the signal and the apparent loudness of the signal, and so some musicians often will deliberately introduce some small amount of this distortion, even though it is “digital distortion” which is considered a “harsh” non-linearity“ (Kahrs und Brandenburg 1998)

Um mit der Lautheit der Konkurrenz mithalten zu können muss jeder Künstler für sich selbst entscheiden wie stark er seinen Song verzerren, komprimieren und begrenzen möchte (Toulson et al. 2014). Wenn der Künstler bei dem Mastering übertreibt, wird die Dynamik des Audiosignals so stark vermindert, dass das Hörerlebnis, durch fehlende Amplitudenvarianzen, an Qualität verliert. Zusätzlich kann so viel Verzerrung eintreten, dass das Ursprungsmaterial zu stark verfremdet wird (Ruschkowski 2008). Wenn der Künstler allerdings zu wenig Verzerrung, Kompression und Limitierung der Amplitude des Signals verwendet, so wird der Song weniger laut als die Songs der Konkurrenz. Dadurch geht der Künstler die Gefahr ein, dass sein Werk unter der Vielzahl an anderen lauteren Songs untergeht, da der Hörer nicht unbedingt seine Abhörlautstärke für diesen einen Song ändern wird. Die Dritte Anwendung für Verzerrung die Bonanno (2019) für eine Mixing and Mastering Situation beschreibt ist die, dass Verzerrung auf den Seiten des Stereosignals, aber nicht auf den Mitten, angewandt werden kann. Der Effekt sei eine Vergrößerung des Stereobildes. In der Welt der Technik ist Verzerrung meist ein Makel. In der musikalischen Anwendung ist Verzerrung ein starkes Werkzeug.

1.3 Stand der Verzerrungs-Technik

Technik und Musik stehen in einer Wechselwirkung. Konzepte und Wirkprinzipien aus der traditionellen Technik werden zweckentfremdet und in der Musikindustrie angewendet. Um musikalische Bedürfnisse zu stillen bedarf es allerdings auch neuer Technik. Technischer Fortschritt beeinflusst die Musik und erschafft neue Klänge und damit neue Musikgenres. Verzerrung ist mittlerweile ein fester Bestandteil der Musikkultur. Hochwertige Verzerrungsplugins und Hardware sind nach wie vor gefragt. Allerdings ist die Entwicklung der Verzerrungstechnik in der Musikbranche nicht geradlinig. Während neue Konzepte den Markt erobern, geschieht gleichzeitig auch eine Rückbesinnung auf alte Technik. Das primäre Qualitätsmerkmal ist der Klang. Ein gutes Beispiel für die aktuelle Lage der Verzerrungstechnik ist der *Kemper Amp*. Musiker, vor allem Gitarristen schwören nach wie vor auf Röhrenverstärker, da ihr Obertonverhalten, aber auch ihre Pegeldynamik einzigartig ist. (Bulling 2013) Der *Kemper Amp* allerdings ist ein Digitaler Verstärker, welcher mit einem komplexen patentierten Verfahren auch Röhrenverstärker emulieren kann. Diese Simulation des Röhrenverstärkers ist so gut, dass der *Kemper Amp* von einer Vielzahl an Profis sogar für Live-Auftritte verwendet wird (Güte 2018). Um mit dem *Kemper Amp* einen anderen Verstärker zu simulieren, muss der andere Verstärker mit verschiedenen Testsignalen vermessen werden. Es werden

Impulsantworten, aber auch mit der Gitarre eingespielte Signale verwendet. Durch eine Datenbank an bereits vermessenen Verstärkern, kann der Besitzer eines Kemper Amps auf eine Vielzahl an emulierten Verstärkern zurückgreifen. Viele Musiker schwören auf den analogen Sound. Allerdings kann dieser heutzutage auch mit digitalen Mitteln erreicht werden. Das Ziel von Analog Modeling ist es, eine analoge Schaltung digital nachzubilden. (Eichas und Zölzer 2018) Auf mathematischem Wege können so Schaltungen emuliert werden. Audioplugins mit solchen Algorithmen sind sehr beliebt auf dem Musikmarkt. Mit vergleichsweise wenig Geld kann der Musiker so mit einem Plugin den Sound des teuren analogen Äquivalents erhalten. Wie gut das funktioniert, ist allerdings umstritten. Profis aus dem Bereich des Mixing und Mastering investieren trotzdem in die teure analoge Technik. Allerdings ist auch für diese Profis ein hybrider Workflow von analog und digital Technik alltäglich. Digitale Verzerrung muss allerdings keines Weges immer versuchen analoge Verzerrung zu emulieren. Digitale Verzerrung beruht lediglich auf den Gesetzen der Mathematik und der Informatik. Mit diesen Werkzeugen sind andere Verzerrungen möglich als mit den Gesetzen der Elektrotechnik. Mathematik umschreibt das in der Realität beobachtete auf eine sehr idealistische Art und Weise. Das ermöglicht es Verzerrungen zu erstellen die so in der Elektrotechnik nicht passieren würden, da in dieser sehr viel mehr Faktoren einen Einfluss haben. Hauptsächlich digitale Verzerrungs-Plugins wie z.B. *Fabfilter Saturn* werden für fast alle Genres und Anwendungsbereiche verwendet. Das erwähnte Plugin bietet eine Multibandfunktion, welche es ermöglicht für jedes Band eine andere Verzerrung zu verwenden. Solche komplexen Filtersysteme, mit hoher Flankensteilheit, sind ein großer Vorteil der Digitaltechnik. Mithilfe der Filter und mathematisch idealer Verzerrung ist *Fabfilter Saturn* ein sehr präzises Werkzeug, welches Detailarbeit ermöglicht. Die Firma *Neural DSP* entwickelte eine Reihe von Verstärker Plugins, welche für etwa 100 Euro dem Gitarristen einen Sound geben, welcher den analogen Verstärkern in wenig nachsteht. Die Plugins emulieren dabei den Sound eines Künstlers. Die von ihm verwendete Signal kette wird durch das Plugin emuliert. Ein gelungenes Verkaufskonzept, welches im Preis-Leistungs-Verhältnis überzeugt ist. Die verhältnismäßig niedrigen Preise von Audio-Plugins gegenüber von Audio-Hardware gewährleisten ihre Popularität und Relevanz. In naher Zukunft werden die neusten Erkenntnisse der Informatik wahrscheinlich einen großen Einfluss auf die Audio-Branche haben. Machine learning Algorithmen und Neural Networks können benutzt werden, um analoge Schaltungen zu analysieren (Chowdhury 2020). Fortschritte in der Technik ermöglichen es neuartige Audio-Produkte zu erschaffen. Diese neuen Produkte haben wiederum einen großen Einfluss auf die Musik-kultur. Auch in Zukunft wird Musik und Technik im festen Zusammenhang stehen und sich gegenseitig beeinflussen und weiterentwickeln.

2 Analyse von Verzerrungsalgorithmen

2.1 Einleitung

In diesem Kapitel werden verschiedene Verzerrungsalgorithmen analysiert und anschließend vergleichend dargestellt. Algorithmen, die für brauchbar befunden wurden, werden in das Audio-Plugin eingebaut. Ziel ist es Algorithmen zu finden, die simpel sind und fehlerfrei funktionieren. Für des Audio-Plugin werden Algorithmen benötigt, die unterschiedliches Obertonverhalten aufweisen, um dem Musiker verschiedene Gestaltungsmöglichkeiten zu bieten. Es werden Algorithmen benötigt, die gerade, ungerade oder gerade und ungerade Obertöne erzeugen. Es werden Algorithmen benötigt, die sehr viele Obertöne erzeugen aber auch diejenigen, die nur sehr wenige erzeugen. Ein nicht geeigneter Algorithmus erzeugt Störgeräusche, die nicht mit der Verzerrung im Zusammenhang stehen. Die Ursache des Fehlers ist meistens mathematischer Natur, da nicht alle Befehle und Operationen für Verzerrung geeignet sind. Das Ausgangssignal muss genau wie das Eingangssignal ausschließlich Amplitudenwerte zwischen -1 und 1 beinhalten. Amplitudenwerte von Samples, die diese Grenze überschreiten werden, unerwünschte Störgeräusche erzeugen oder komplizierte Fehler hervorrufen. Es wird eine vergleichende Matrix aufgestellt, die es ermöglichen soll, die Algorithmen in verschiedene Anwendungsfälle einzuteilen und deren Implementierung in ein Audio-Plugin zu erleichtern.

2.2 Überblick des Analyseverfahrens

2.2.1 Analyse des Zeit- und Frequenzbereichs

Es soll dargestellt werden, wie die Transferfunktion des Algorithmus aussieht und welchen Einfluss diese auf die Wellenform und das Spektrum hat. Diese Zusammenhänge sind für jeden Verzerrungsalgorithmus unterschiedlich. Deswegen wird diese Analyse für jeden Algorithmus stadtfinden, beginnend mit Abbildung 4 für die erste Verzerrung. Als Testsignal wird eine Sinusschwingung bei 100Hz verwendet. Sinusschwingungen werden in der Literatur häufig verwendet, um Verzerrungen zu analysieren (Yeh 2009; Temme 1992; Sunnerberg 2019). Der Vorteil ist, dass bei einer verzerrten Sinusschwingung keine Kombinationsfrequenzen auftreten wie, vergleichsweise bei, zum Beispiel, zwei oder mehr Sinusschwingungen. Dadurch kann der Effekt der Verzerrung auf Zeit- und Frequenzverhalten einfacher verstanden werden. Es wurde ein 100 Hz Testsignal gewählt, weil es für die visuelle Darstellung günstig ist, möglichst viele Abtastwerte pro Schwingungsperiode zu haben. Auch eine Sinusschwingung auf einer anderen

Frequenz wäre ein sinnvolles Testsignal, solange genug Bandbreite innerhalb des hörbaren Bereichs für die Beobachtung der entstehenden Obertöne zur Verfügung stehen würde. Allerdings wird mit steigender Frequenz die Darstellung des Zeitbereichs schlechter, da weniger Abtastwerte zur Verfügung stehen. Im ersten Teilbild (siehe Abbildung 4) wird der Zeitbereich des Signals dargestellt. Die Zeit ist auf eine Schwingungsperiode normalisiert. Die Amplitude reicht von -1 bis 1. Grund dafür ist, dass dies die Maximalwerte eines digitalen Audiosignals in einer Digital Audio Workstation sind. Es wird das Eingangssignal, die 100Hz Sinusschwingung in Blau, sowie das Ausgangssignal, der verzerrte Sinus in Rot, dargestellt. Im zweiten Teilbild wird die Transferfunktion, welche im englischen u.a. auch als Input/Output-Curve bezeichnet wird, dargestellt. Wie der englische Name vermuten lässt, wird die Amplitude des Input-Signals gegen die Amplitude des Output-Signal dargestellt. Anhand dieser Darstellung kann man sehen, wie der Algorithmus den Datensatz verändert. Es kann zugeordnet werden welcher Output-Wert durch einen spezifischen Input-Wert erzeugt wird. Diese Darstellung ist sehr Praxis nah. Teilweise sind mathematische Funktionen der Algorithmen in der Transferfunktion direkt erkennbar. Das dritte Teilbild ist eine FFT mit einer Berechnung des Total Harmonic Distortion Wert. Diese von MATLAB eingebaute Funktion, *thd()*, nummeriert direkt die Obertöne. Dadurch kann einfach erkannt werden, ob ein Algorithmus gerade, ungerade, oder gerade und ungerade Obertöne erzeugt. Dabei ist die Anzeige auf maximal 50 Obertöne beschränkt. Es kann die Amplitude der Obertöne im Verhältnis zum Grundton abgelesen werden. Die Betrachtung der Total Harmonic Distortion ist allerdings nicht bedeutsam für den Anwendungsfall, deswegen wird diese auch nicht ausgewertet. Ein größer, oder kleiner, Klirrfaktor hat wenig Aussagekraft über die Qualität der Verzerrung in Bezug auf die musikalische Anwendung. Ein sehr hoher Gain-Wert oder Overdrive-Wert erzeugt mehr Obertöne und damit mehr Aliasing. Es ist also sinnvoll den Algorithmus so zu beschränken das er im hörbaren Bereich Obertöne erzeugt, aber dass die Bandbreitenvergrößerung in Bezug auf das Aliasing gering ist. Dabei soll das Verfahren im nächsten Kapitel behilflich sein.

2.2.2 Analyse des Aliasing-Effekts

„Nonlinear signal processing blocks are known to expand the bandwidth of the incoming signal, which in a DSP system can cause aliasing if the bandwidth of the output exceeds the Nyquist frequency (i.e., half the sampling rate). An amplifier model can distort harmonic signals such as a guitar tone and produce many new harmonics in the output that, through aliasing into the audio range, are no longer harmonically related to the original tone. The resulting noisy, "dissonant" sound owing to aliasing is characteristic of low cost digital implementations of strong distortions and is typically mitigated through running the distortion algorithm at an oversampled rate, which is computationally expensive." (Pakarinen und Yeh 2009, S. 91)

Der Alias-Effekt kann einen Verzerrungsalgorithmus unbrauchbar machen. Oversampling ist der einfachste Weg, um die Auswirkungen des Aliasing-Effekts zu verringern. Allerdings ist die Überabtastung rechenaufwendig. Die Rechenleistung ist begrenzt. Ein Audio-Plugin besteht für gewöhnlich aus mehr als nur einem Verzerrungsalgorithmus. Es könnten zum Beispiel noch Effekte wie Hall o.a., in Reihe, nach der Verzerrung geschalten sein. In dem Fall müsste der Verzerrungsalgorithmus so wenig wie möglich Leistung in Anspruch nehmen. Deswegen sollen unterschiedliche Multiplikatoren der Überabtastung für den jeweiligen Algorithmus gemessen werden. Die Analyse des Aliasing-Effekts basiert auf Yeh (2009). Laut ihm ist die Grundfrequenz einer Gitarre ungefähr zwischen 80 Hz und 1 kHz. Deswegen nutzt er als Testsignal eine 1kHz Sinusschwingung. Je höher die Frequenz des Eingangssignals desto mehr Bandbreite erzeugt die Verzerrung. Deswegen wird der höchste Ton verwendet, den die Gitarre erzeugen kann. Dieses Testsignal soll *Testsignal „Gitarre“* heißen. Das zweite Scenario das Yeh (2009) anspricht ist eine Rechteckschwingung, die durch eine extreme ideale Verzerrung erzeugt wird, bei einer Grundfrequenz von 20 kHz. Dies ist eine theoretische Betrachtung von einer Sinusschwingung bei 20 kHz, welche von einer Verzerrung in eine Rechteckschwingung verwandelt wird. Die Bandbreitenvergrößerung ist bei 20 kHz die größte, im Vergleich mit allen anderen Frequenzen, welche im theoretisch hörbaren Bereich liegen. Die 20 kHz Sinusschwingung ist ein Belastungstest für den Verzerrungsalgorithmus. Wenn man den Verzerrungsalgorithmus für den gesamten hörbaren Bereich benutzen möchte, also von 20 Hz-20 kHz, so muss das Aliasing, welches durch die 20 kHz Sinusschwingung erzeugt wird, annehmbar sein. Dieses Testsignal soll *Testsignal „Breitband“* heißen. Yeh (2009) verwendet eine Einheit, um die Intensität des Aliasing einzuschätzen. Diese ist die maximale Amplitude der durch den Alias-Effekt gespiegelten Frequenzen unter 20 kHz relativ zur Grundfrequenz. Diese Einheit soll *MaxAlias* heißen. Der Alias-Effekt ist für diese Anwendung nur im hörbaren Bereich, von 20 – 20 kHz, störend. Die gespiegelte Frequenz die am lautesten ist bestimmt den Wert. Demonstriert wird dies nun mit einer Rechteckwelle bei 1 kHz. Dies soll eine ideale sehr starke Verzerrung einer Sinusschwingung simulieren. Eine Hard-Clip-Verzerrung kann eine Sinusschwingung in eine Rechteckschwingung verwandeln. In Abbildung 2**Fehler! Verweisquelle konnte nicht gefunden werden.** ist die Spiegelung der Frequenzen der Rechteckschwingung an der Nyquist-Frequenz, der Hälfte der Abtastfrequenz, deutlich zu sehen. Der Maximalwert des Alias-Effekts kann direkt unter 20 kHz abgelesen werden. *MaxAlias = -28 dB.* In Abbildung 1 ist wurde die Abtastfrequenz verdoppelt. Sie wurde von 44,1 kHz auf 88,2 kHz erhöht. Das verschiebt die Nyquist-Frequenz von 22,05 kHz auf 44,1 kHz. Das bedeutet, dass die Obertöne, die durch die 1 kHz Rechteckschwingung entstehen jetzt mehr Bandbreite zur Verfügung haben, bevor sie wieder in den hörbaren Bereich hinein gespiegelt werden. Jeder höhere Oberton nimmt in der Amplitude ab. Dadurch sorgt eine Vergrößerung der Bandbreit dafür, dass die gespiegelten Obertöne eine niedrigere Amplitude aufweisen. Der *MaxAlias* wird dadurch niedriger. Mit zweifacher Überabtastung sinkt der Wert von -28 dB auf -37 dB. Der Wert wird

links neben der grünen Linie, welche 20 kHz markiert, abgelesen. Die Grundfrequenz ist immer bei 0 dB.

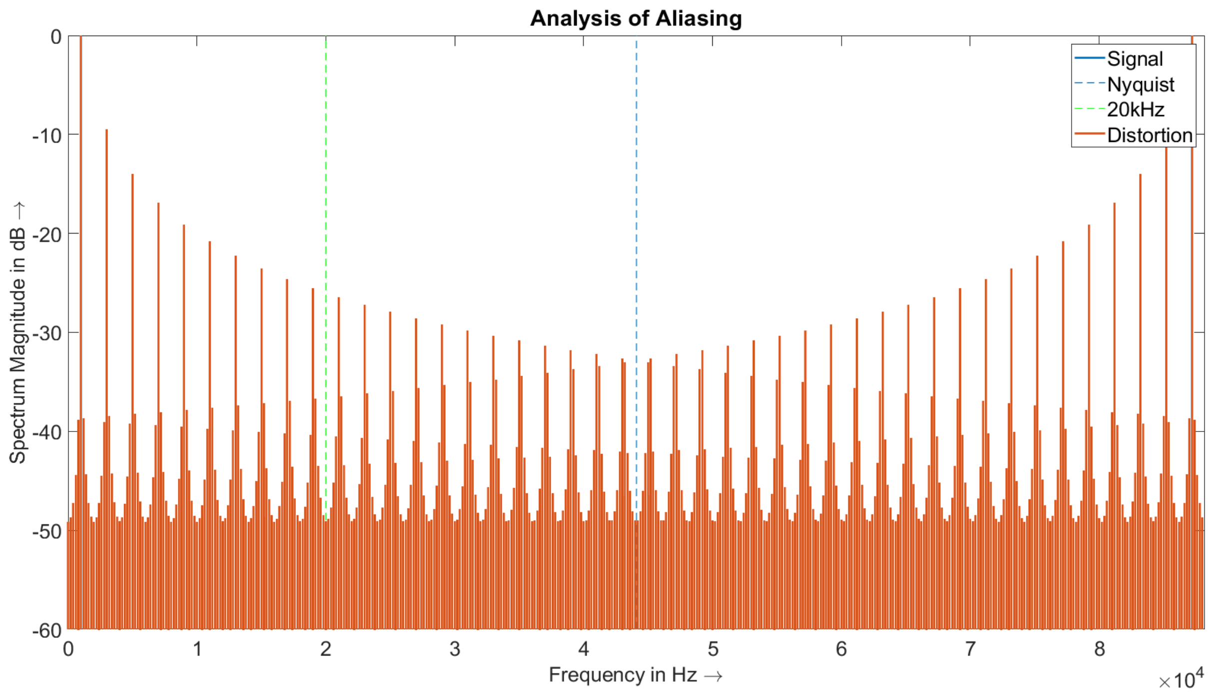


Abbildung 1Alias-Effekt einer 1kHz Rechteckschwingung mit zweifacher Überabtastung

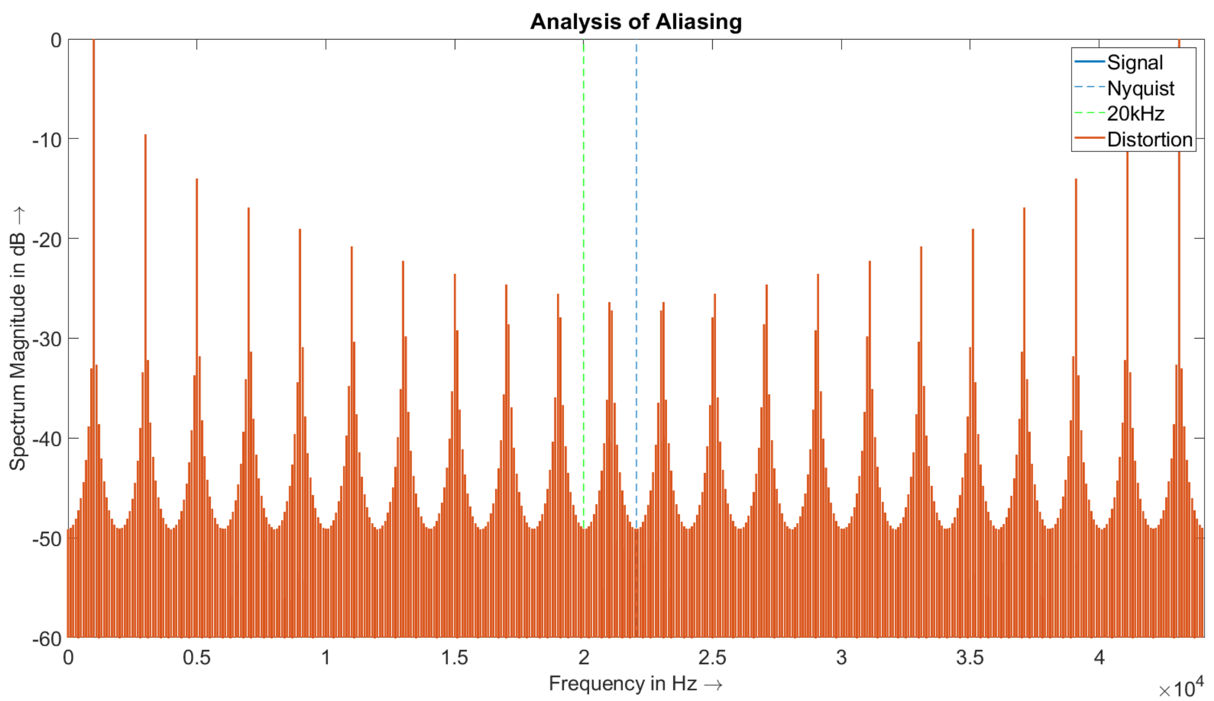


Abbildung 2 Alias-Effekt einer 1kHz Rechteckschwingung ohne Überabtastung

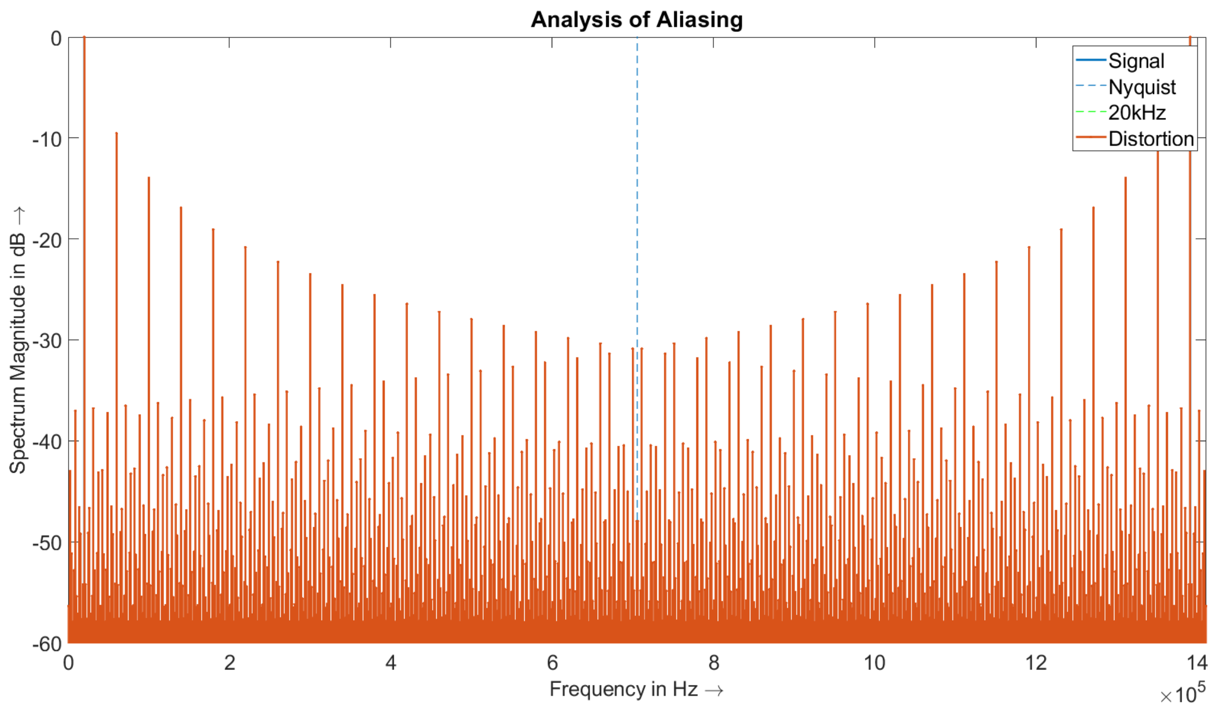


Abbildung 3 Alias-Effekt einer 20 kHz Rechteckschwingung mit 32-facher Überabtastung

Eine 20 kHz Rechteckschwingung erzeugt selbst bei einer 32-fachen Überabtastung einen *MaxAlias* von -37 dB. Die Bandbreitenvergrößerung, die durch diese ideale Verzerrung entstehen würde, ist so groß, dass selbst mit einer Abtastfrequenz von 1,4112 Millionen Hertz keine signifikante Verbesserung des Alias-Effekts erzielt werden kann. Das Obertonverhalten der unterschiedlichen Verzerrungs-Algorithmen und das daraus entstehende Aliasing, ist von Algorithmus zu Algorithmus sehr verschieden. Daraus ergibt sich die Notwendigkeit für jeden Algorithmus eine Betrachtung des Aliasing, in Bezug auf das Oversampling durchzuführen. Es werden folgende Oversampling-Faktoren betrachtet: 1-fach, 2-fach, 4-fach, 8-fach ,16-fach und 32-fach. Der *maxAlias* wird für das Testsignal „*Gitarre*“ und das Testsignal „*Breitband*“ für die genannten Faktoren gemessen. Dies wird für jeden Algorithmus durchgeführt. Das Ergebnis ist eine vergleichende Tabelle. Aus dieser Tabelle erhält man zwei Informationen. Die beiden Testsignale sollen Anwendungsfälle unterscheiden. Grob gesagt könnte man die Eignung der Algorithmen dadurch in eine niederfrequente und eine hochfrequente Anwendung unterscheiden. Für die zwei Fälle stehen dann die gemessenen maxAlias-Werte der jeweiligen Überabtastung zur Verfügung. Anhand dieser Daten kann ein geeigneter Oversampling-Faktor gewählt werden. Es muss abgewogen werden, zwischen niedrigem Aliasing und niedriger Rechenleistung. Erst durch die Messung und die dadurch entstehende Vergleichende Darstellung ist diese Entscheidung möglich. Dieser systematische Ansatz soll es ermöglichen, dass die Entscheidung des Oversampling-Faktors weniger willkürlich ist, sondern das nach objektiven Parametern getroffen werden kann. Aus verschiedenen Quellen ist bekannt, dass eine 8-fache Überabtastung für eine

Gitarren Verzerrung ausreichend ist. Yeh (2009) errechnet mit der 1 kHz Rechteckwelle und einer 8-fachen Überabtastung einen *maxAlias* von -50 dB relativ zu Grundschwingung aus. Im Umkehrschluss bedeutet dies, dass ein *maxAlias* von -50dB ausreichend gut ist. Dieses Ergebnis konnte reproduziert werden. -50 dB soll als Richtwert gelten. Anhand von diesem Richtwert wird entschieden, ob die Verzerrung für den Anwendungsfall geeignet ist.

> „We find that for high quality distortion effects, an oversampling factor of 8× is sufficient. In practice, lower oversampling factors may also be acceptable because the aliasing is masked by complex distorted guitar tones or noise.“ (Yeh 2009, S. 13)

Es ist schwierig einen genauen Wert für *maxAlias* zu finden bei dem man davon ausgehen kann, dass das entstehende Aliasing unhörbar ist oder vom Nutzsignal maskiert wird. Der Wert *maxAlias* ist stark abhängig vom Eingangssignal. Man könnte zum Beispiel argumentieren, dass ein Rauschen ein gutes Testsignal wäre, da die Verzerrung im Anwendungsfall auf komplexe Signale angewandt werden wird, deren Frequenzinhalt stochastische Anteile besitzen kann. Das ist aber kein sinnvoller Test, da es schwierig, bis unmöglich, ist bei einem Rauschen den *maxAlias*, mit gewählter Methode, aus einem FFT-Analysator abzulesen. Der verzerrte Sinus bietet die Möglichkeit, den Alias-Effekt gut darstellen zu können. Für eine Gitarre ist dies ein Sinnvolles Testsignal, da der Klang einer Gitarre aus Harmonischen besteht. Das Aliasing, welches durch andere Klänge entsteht, kann mit der Methode nicht prognostiziert werden. Dies ist allerdings auch nicht das Ziel der Messung. Das Ziel ist es Algorithmen vergleichen zu können und Tendenzen in Bezug auf das Aliasing erkennten zu können. Die Sinnhaftigkeit soll einmal an einem Beispiel erklärt werden. In Tabelle 1 werden die Messwerte der idealen Verzerrung, also des Rechtecksignals und die Werte einer Verzerrung, welche im nächsten Kapitel erklärt wird, dargestellt. Es ist leicht zu erkennen, das große Unterschiede des erzeugten Aliasing vorliegen. Algorithmus 1 erzeugt bei Testsignal *„Gitarre*“ fast kein Aliasing. Auch ohne Oversampling ist der *maxAlias* ca. bei – 200 dB. Dieser Algorithmus benötigt für den Anwendungsfall kein Oversampling. Die Ideale Verzerrung, Verzerrung 2, erzeugt den Richtwert von -50 dB bei 8-facher Überabtastung. Beide Algorithmen sind für die Anwendung bei einem Gitarrensignal geeignet, müssen aber anders implementiert werden. Beim Testsignal *“Breitband“* erreicht die Verzerrung 1 den Richtwert bei 8-fachen Oversampling. Die ideale Verzerrung hingegen erreicht auch bei 32-fachen Oversampling den Richtwert nicht. Während Verzerrung 1 wenig Aliasing beim Testsignal *„Breitband“* erzeugt und somit eine hohe klangliche Qualität bereitstellt, ist Verzerrung 2 nicht geeignet für diesen Anwendungsfall.

Tabelle 1 Erklärung der Messung und Beurteilung des Alias-Effekts

Verzerrungsalgorithmus			1	2
Maximale Amplitude des Alias-Effekts unter 20 kHz in dB	1 kHz Sinus "Gitarre"	x1	-198,5	-28
		x2	-270	-37
		x4	-275	-45
		x8	-285	-51,5
		x16	-285	-58
		x32	-285	-64
	20 kHz Sinus "Breitband"	x1	-11,5	-9,5
		x2	-19,5	-14
		x4	-38,5	-19
		x8	-98,5	-25
		x16	-240	-31
		x32	-240	-37
1. Araya und Suyuma (Reihenschaltung)				
2. Ideale Verzerrung (Rechteckwelle)				

Die Endauswertung folgt nach der Vorstellung aller Algorithmen. Mit Blau markierte Felder unterschreiten den Richtwert von – 50 dB. Diese Überabtastungsfaktoren sind für den jeweiligen Algorithmus geeignet.

2.3 Algorithmen

2.3.1 Hard Clipping

Laut (Tarr 2018) funktioniert die Hard-Clipping-Verzerrung indem die Amplitude des Signales auf ein Maximum limitiert wird. Dieser Effekt soll bei der maximalen Aussteuerung eines Transistors auftreten. Der Parameter *thresh* bestimmt den Wert der Amplitude ab dem das Signal begrenzt wird. In diesem Fall bedeutet begrenzt, dass alle Samples des Audiosignals, die den durch *thresh* festgelegten maximalen Amplitudenwert überschreiten, stattdessen für ihren jeweiligen Amplitudenwert den Wert *thresh* zugewiesen bekommen. Das Verfahren wird für positive Werte, aber auch für negative Werte äquivalent angewandt. Dadurch wird die Symmetrie des Verfahrens erhalten. (ebd.) Im Anhang ist der MATLAB-Code von Tarr aufzufinden [siehe Anhang: 1.1.1 Hard-Clip-Algorithmus (Tarr 2018)].

$$y(n) = \begin{cases} thresh & \text{if x[n] is} > \text{thresh} \\ x[n] & \text{if} - \text{thresh} < x[n] < \text{thresh} \\ -thresh & \text{if x[n] is} < -\text{thresh} \end{cases}$$

Formel 3 Hard-Clipping-Algorithmus (Tarr 2018)

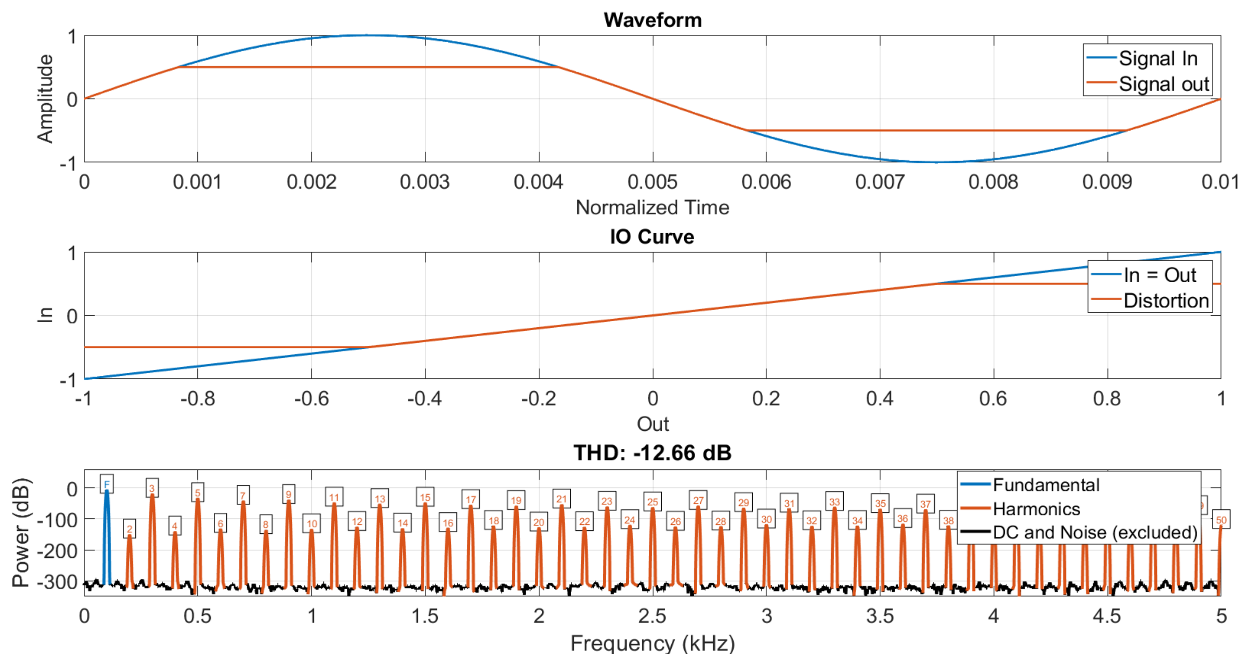


Abbildung 4 Analyse des Hard-Clipping-Algorithmus

Die Verzerrung erzeugt sowohl gerade als auch ungerade Obertöne. Die Ungeraden Obertöne weisen allerdings eine größere Amplitude auf als die geraden. Die geraden Obertöne sind ca. 100 dB leiser als die ungeraden. Der Klang wird also primär von den ungeraden Obertönen bestimmt. Es werden sehr viele Obertöne erzeugt. Es wird ein starkes Aliasing stadtfinden. Im Bild (Abbildung 4) sieht man wie der Parameter *thresh* mit einem Wert von 0,5 die Begrenzung der Amplitude einstellt. Intuitiv würde man denken, dass wenn die Wellenform nur ein wenig begrenzt werden würde, dass dies weniger Obertöne erzeugen würde. Dies ist aber nicht der Fall. Wenn der Sinus mit einer maximalen Amplitude von 1 ab dem Wert 0,9 begrenzt wird, so ergibt sich ein sehr ähnliches Obertonverhalten wie in Abbildung 4. Der Grad von wenig Verzerrung zu voller Verzerrung ist gering. “Ein digitales System hingegen reagiert bei Überschreitung der durch die Wortbreite fest gegebenen Aussteuerungsgrenze mit einem sprunghaften Anstieg des Klirrfaktors (Schultz et al. 2008, S. 2). Die Hard-Clip-Verzerrung ist auch als digitale Verzerrung bekannt und ist in verschiedenen Gebieten der Digitaltechnik ein Problem. Wenn für *thresh* ein Wert von eins gewählt wird, so erhält man genau die Verzerrung, die entstehen würde, wenn man in einer DAW ein Signal, über die Amplitudengrenzen von 1 und -1 hinaus, verstärken würde. Wenn das Signal begrenzt ist, dann erzeugen

Amplitudenschwankungen des Signals kein anderes Obertonverhalten. Das ist aus musikalischer Sicht wenig sinnvoll. Bei einer E-Gitarre würde dadurch eine unterschiedliche Anschlagdynamik wenig Einfluss auf das Oberton verhalten aufweisen. Das Klangerlebnis würde dadurch an Qualität verlieren. Tatsächlich wird die Hardclip-Verzerrung in der Musiktechnik gemieden.

Hard Clipping der positiven Halbwelle

„Signals, like the positive peak limited sine wave, limited only on the upper half-cycle […], contain higher amplitude even order harmonics than odd order harmonics.“ (Temme 1992, S. 2)

Wenn man den Code für das Hard Clipping von Tarr (2018) nur minimal verändert, kann damit das von Temme beschriebene Phänomen erzeugt werden. Der leicht Veränderte Code ist im Anhang zu finden [siehe Anhang: Limitierung von positiven Halbwellen]

$$y(n) = \begin{cases} thresh & \text{if x[n] is} > \text{thresh} \\ x[n] & \text{if x[n] is} \leq \text{thresh} \end{cases}$$

Formel 4 Hard Clipping der positiven Halbwelle

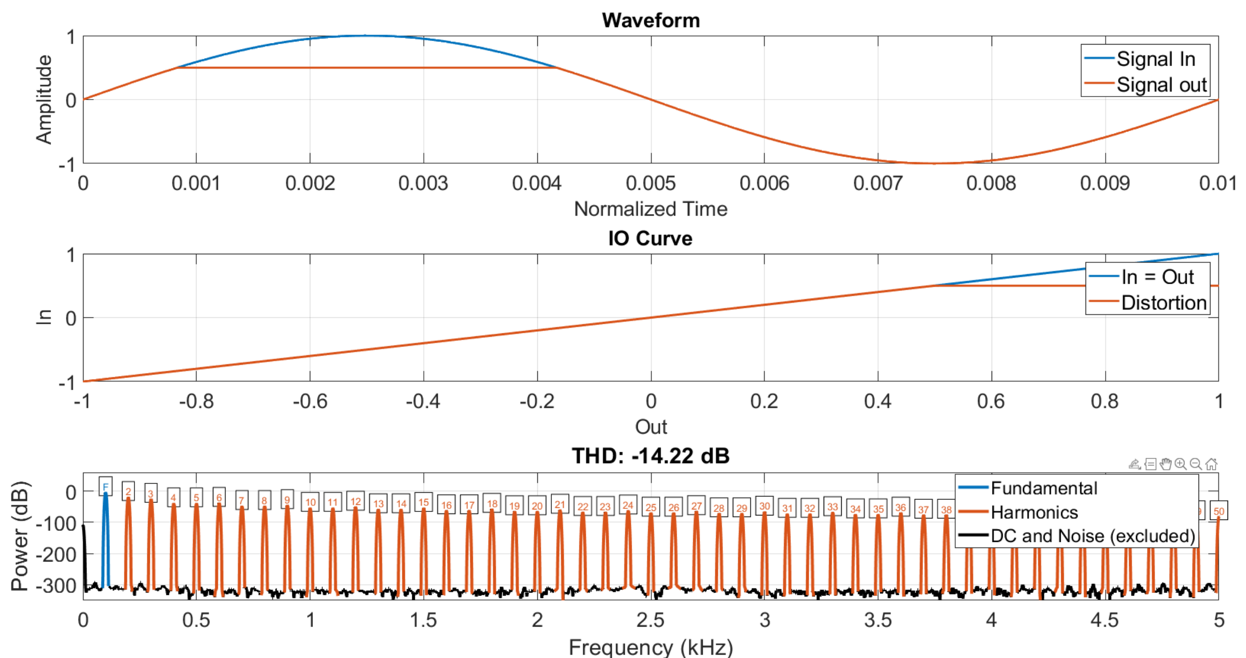


Abbildung 5 Analyse des Algorithmus für positive Halbwellen-Limitierung

Laut Temme (1992) soll dieser Algorithmus gerade Obertöne erzeugen, deren Amplitude größer sei als die Amplitude der ungeraden Obertöne. Der Oberton mit der größten Amplitude ist gerade. Es gibt allerdings andere Algorithmen die einen stärken Fokus auf

die Erschaffung von geraden Obertönen aufweisen. Der Algorithmus hat das gleiche Problem wie der Hard-Clipping-Algorithmus, nämlich das er zu wenig dynamisches Obertonverhalten aufweist.

2.3.2 Gleichrichtung

In Zölzer et al. (2002) wird die Rectification-Verzerrung vorgestellt. Auf Deutsch entspricht dies dem Effekt eines Gleichrichters. Es gibt Halbwellengleichrichtung und Vollwellengleichrichtung. Beides erschafft gerade Obertöne. Tarr (2018) stellt MATLAB Code zur Verfügung, welcher genau diese Verzerrung erschafft [siehe Anhang: Gleichrichter].

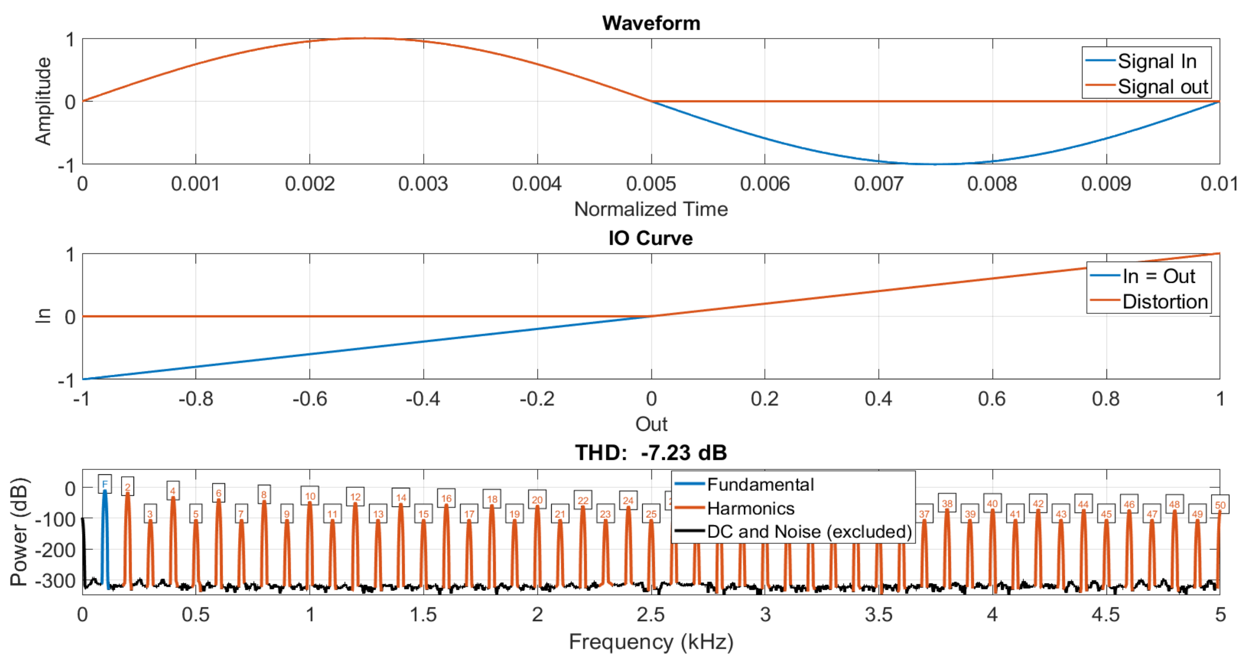


Abbildung 6 Analyse des Halbwellengleichrichters

Alle negativen Amplitudenwerte werden durch Null ersetzt. Dadurch wird die negative Halbwelle der Sinusschwingung entfernt. Es entsteht ein Gleichanteil. Das Obertonverhalten dieser Verzerrung ist bei jeder Eingangsamplitude gleich. Es entstehen gerade und ungerade Obertöne. Allerdings sind die ungeraden etwa 100 dB leiser als die geraden. Die geraden Obertöne werden also deutlich besser wahrgenommen als die ungeraden. Dies ist besonders, für die Gleichrichter-Algorithmen und unterscheidet ihren hörbaren Effekt zu anderen Verzerrungen. In Abbildung 7 wird der Vollwellengleichrichter analysiert. Von allen Eingangswerten wird der Betrag gebildet. Dadurch existieren im Ausgangssignal keine negativen Amplitudenwerte. Die negativen Amplitudenwerte des Eingangssignal werden somit an der x-Achse gespiegelt. Das Resultat ist eine neue Wellenform, welche mit doppelter Frequenz, im Vergleich zur Frequenz des

Eingangssignal schwingt. Die Grundfrequenz des Eingangssignals beträgt 100 Hz. Die Grundfrequenz des Ausgangssignals beträgt 200 Hz. Auch bei dieser Verzerrung entstehen gerade Obertöne deren Amplitude etwa 100dB über der Amplitude der ebenfalls entstehenden ungeraden Obertöne liegt. Wie auch beim Halbwellengleichrichter, hat die Amplitude des Eingangssignals beim Vollwellengleichrichter keinen Einfluss auf das Oberton verhalten. Die FFT markiert die eigentliche 100 Hz Grundschwingung und deren Obertöne mit schwarz, da deren Amplitude niedriger ist als die der neuen Grundschwingung. Sie werden als Gleichanteil und Rauschen interpretiert, aber das ist ein Fehler. Die schwarz markierten Frequenzen sind trotz ihrer niedrigen Amplitude Teil des Ausgangssignals.

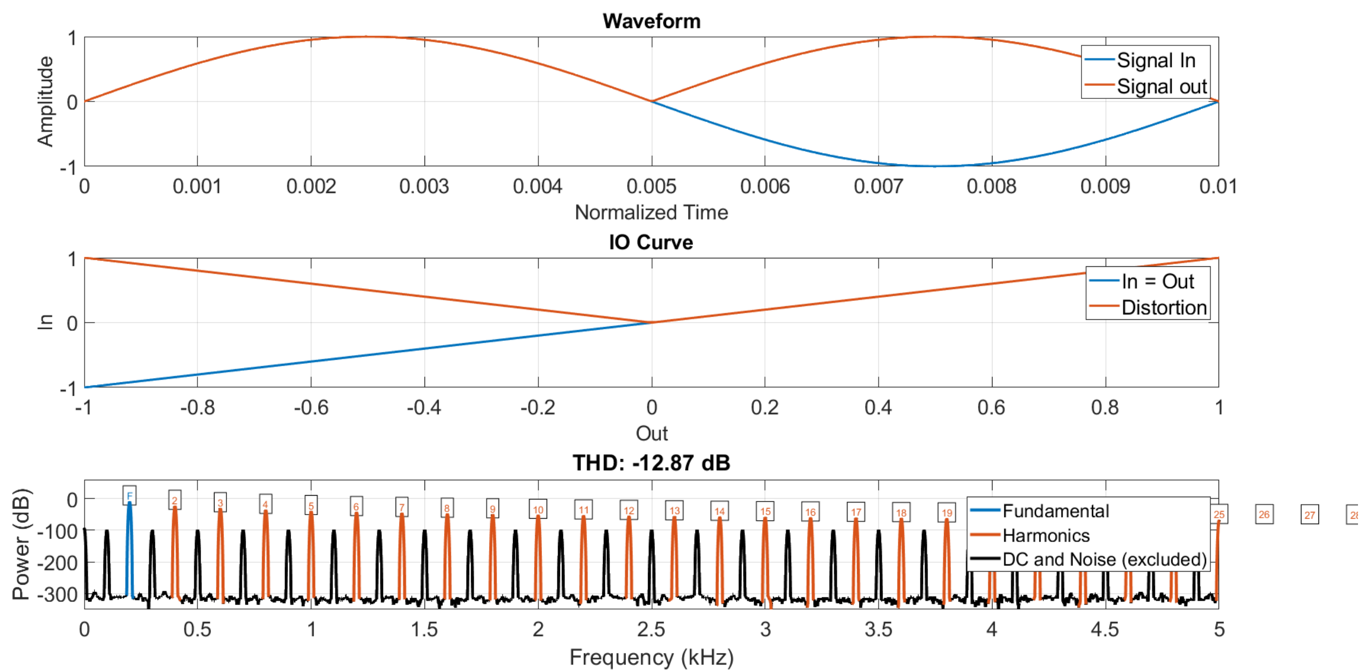


Abbildung 7 Analyse des Vollwellengleichrichters

2.3.3 Araya und Suyama

Die Formeln für die drei Verzerrungen wurden von Pakarinen und Yeh (2009) kompiliert. Es handelt sich um drei verschiedene bereits abgelaufene US-Patente. Diese Algorithmen sind entwickelt wurden, um den Klang von Röhrenverstärkern für Gitarren zu emulieren. Das erste Patent ist von Yamaha. Dies veranschaulicht die Relevanz von Verzerrung, die durch Transferfunktionen erschaffen wird, für die Musikindustrie.

Diese Formel stammt von Araya und Suyama (1996)

$$f(x) = \frac{3x}{2}\left(1 - \frac{x^2}{3}\right)$$

Formel 5 Röhrenverzerrungs-Algorithmus (Araya und Suyama 1996)

Laut Pakarinen und Yeh (2009) wird dieser Algorithmus drei Mal hintereinander in Reihe geschaltet, um mehr Verzerrung zu erhalten. Der Algorithmus abgesehen von den Randbereichen recht linear und erzeugt somit wenig Verzerrung. Jede dieser drei Verzerrungsstufen hat einen Skalierungsfaktor, welcher es ermöglicht die Signalstärke einzustellen. Dadurch können die Verzerrungsstufen individuell angesteuert werden. Beschrieben wird ein Hardware-Effekt. Durch einen Analog-Digital-Wandler wird das Gitarrensignal in diesen dreistufigen Verzerrer geleitet, um anschließend durch einen Chorus und einen Hall geleitet zu werden. Danach wird das Signal durch einen Digital-Analog-Wandler zurück in die analoge Domäne gewandelt.

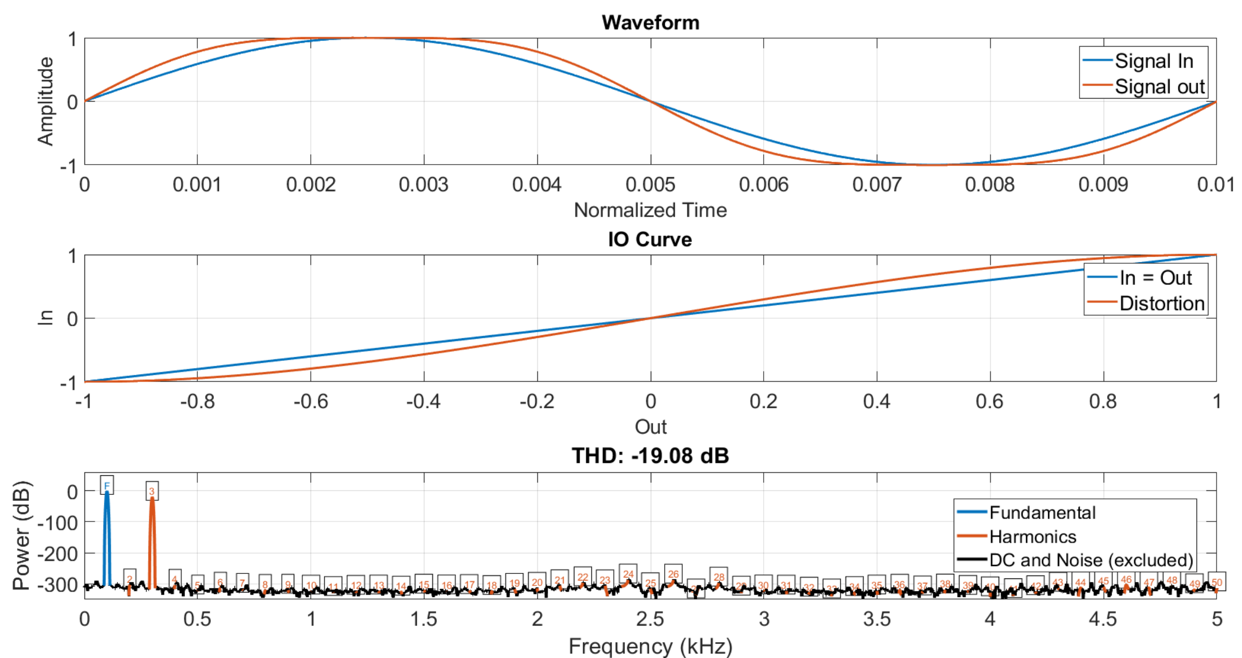


Abbildung 8 Analyse des Röhrenverzerrers von Araya und Suyama (1996)

Der Algorithmus erzeugt nur einen Oberton. Die Frequenz des Obertons ist das dreifache der Grundfrequenz. Es wird wenig bis kein Aliasing erzeugt. Ein niedriger Überabtastungs-Multiplikator wird ausreichen, um eine Breitbandanwendung für diesen Algorithmus zu gewährleisten. Diese Verzerrung könnte dafür benutz werden, um eine subtile Obertonanreicherung an einem Audiosignal vorzunehmen. Dadurch kann z.B. ein Instrument einen „volleren“ Klang erhalten und mehr Dominanz im Mix erhalten. Erst wenn der Algorithmus wie beschrieben dreimal hintereinander in Reihe geschaltet wird, zeichnet sich ein komplexes Obertonverhalten ab. (siehe Abbildung 5) Es wird eine begrenzte Anzahl an ungeraden Obertönen erzeugt. Das ist sinnvoll, um weniger Aliasing zu erzeugen und bestimmt den klanglichen Charakter. Die Amplitude der Obertöne

nimmt mit steigender Frequenz ab. Dieses Verhalten könnte ungefähr mit der Funktion f(x) = log(-x) beschrieben werden. Die ersten Obertöne sind entscheidend für den Klang, da bereits der 7te ungerade Oberton, also der 15te Oberton, eine Amplitude von -100dB im Vergleich zu Grundfrequenz aufweist.

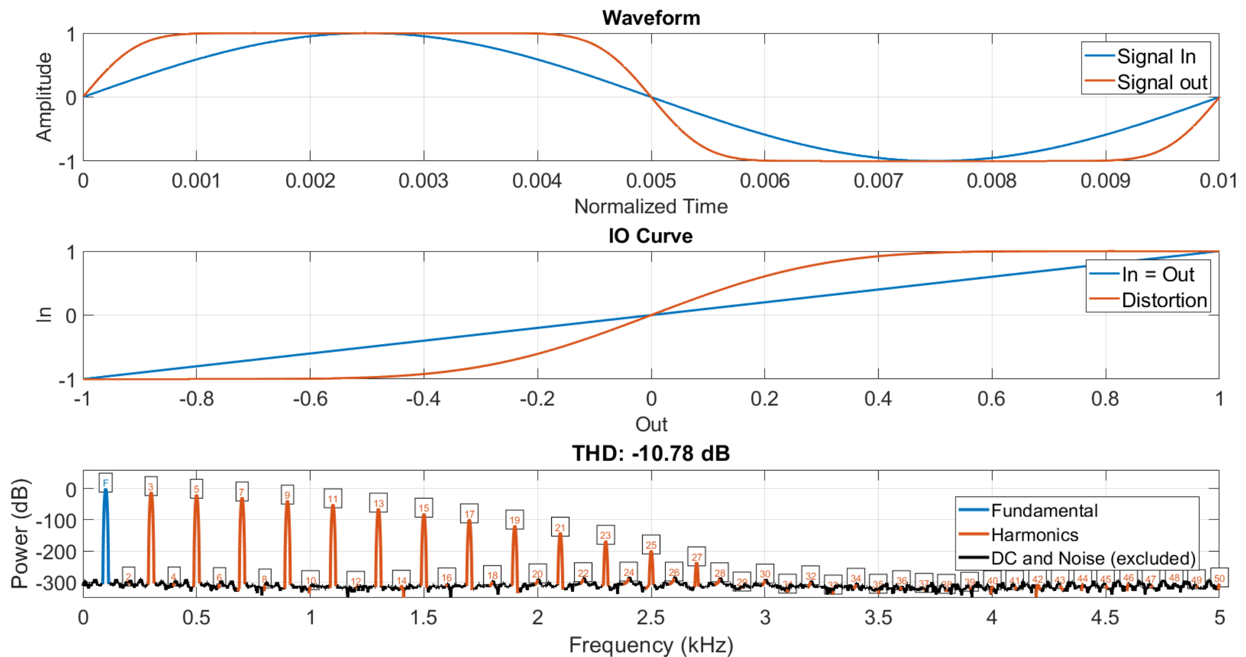


Abbildung 9 Analyse des Röhrenverzerrers von Araya und Suyama (1996) in Reihenschaltung

Nicht nur das Prinzip der Reihenschaltung kann anhand von diesem Algorithmus erklärt werden. Dieser Algorithmus ist viel geeigneter für ein Instrument als das Hard Clipping, weil die Amplitude des Eingangssignal die Amplitude und damit die Anzahl an Obertönen bestimmt. Daraus ergibt sich eine interessante Spielerfahrung für den Musiker. Der Klang verändert sich dynamisch. Eine geringe Amplitude des Eingangssignals erzeugt wenig Obertöne und eine hohe Amplitude erzeugt viele Obertöne.

2.3.4 Diodic et al.

Auch das zweite abgelaufene Patent gehört *Yamaha Guitar Group Inc.* Die folgende Formel ist aus (Doidic et al. 1998). Die Formel wurde in MATLAB Code übersetzt [siehe Anhang 1.1.4.].

$$f(x) = -\frac{3}{4}\left\{1 - [1 - (|x| - 0.032847)]^{12} + \frac{1}{3}(|x| - 0.032847)\right\} + 0.01\,,$$

$$for -1 \leq x < -0.08905$$

$$f(x) = -6.153x^2 + 3.9375x,$$

$$for - 0.08905 \leq x < 0.320018$$

$$f(x) = 0.630035,$$

$$for\ 0.320018 \leq x \leq 1$$

Formel 6 asymmetrischer Röhrenverzerrungs-Algorithmus (Doidic et al. 1998)

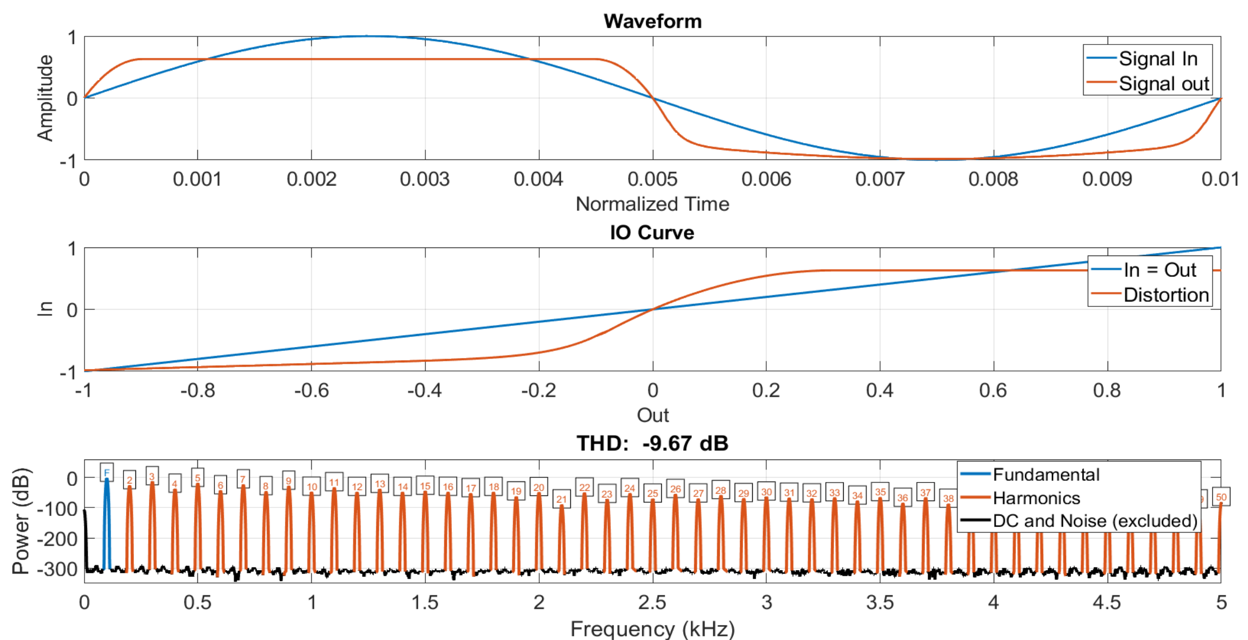


Abbildung 10 Analyse des asymmetrischen Röhrenverzerrers aus Diodic et al. (1998)

Die Formel 6 erschafft eine Funktion, welche sowohl gerade als auch ungerade Obertöne gleichermaßen erzeugt. Die Antisymmetrie zur Y-Achse erzeugt, wie auch in anderen Algorithmen, ungerade Obertöne. Ein Beispiel dafür ist Formel 5. Die geraden Obertöne entstehen durch die Begrenzung der Amplitude der oberen Halbwelle, genau wie bei Formel 4. Die Amplitude der jeweils höheren Obertöne nimmt nicht gleichermaßen ab. Dies ist wahrscheinlich der Approximation der Funktion geschuldet. Das klassische Verfahren, um eine Kennlinie zu erschaffen, ist eine Approximation der gewünschten Funktion mit Taylorreihen. Für vordefinierte Bereiche der x-Achse können unterschiedliche Polynome ausgewählt werden. Dieser Verzerrungsalgorithmus teilt die Amplitudenwerte in drei Bereiche ein und berechnet für die jeweiligen Bereiche das

Ausgangssignal unterschiedlich. Offensichtlich ist dieses Verfahren mit Fehlern verbunden. Fraglich ist, ob dies signifikante Auswirkungen auf den Klang hat. Für eine musikalische Anwendung ist es unwichtig, ob die Mathematik hinter dem Klang perfekt ist. Davon abgesehen ist das Obertonverhalten dieser Verzerrung dynamisch vom Eingangssignal abhängig. Dies ist gut für Instrumentalisten. Scheinbar ist dies eine verwendete Gitarrenverzerrung von Yamaha, da diese Verzerrung, von der Yamaha Guitar Group patentiert war. Wie der Algorithmus angewendet wurde, ist aber nicht öffentlich bekannt.

2.3.5 Bendiksen

Ein weiterer interessanter Röhrenverzerrungsalgorithmus, aus Zölzer et al.(2002), basiert auf der mathematischen Grundlage von Bendiksen (1997). [siehe Anhang: Röhrenemulation (Zölzer et al. 2002)]

$$f(x) = \begin{cases} \dfrac{x-Q}{1-e^{-dist\cdot(x-Q)}} + \dfrac{Q}{1-e^{dist\cdot Q}} & , Q \neq 0, x \neq Q, \\ \dfrac{1}{dist} + \dfrac{Q}{1-e^{dist\cdot Q}} & , \quad x = Q. \end{cases}$$

Formel 7 Röhrenverzerrungs-Algorithmus (Bendiksen 1997)

Laut Zölzer et al. (2002) erzeugt dieser Algorithmus wenig bis keine Verzerrung wenn das Eingangssignal eine niedrige Amplitude aufweist. Die Kennlinie soll große negative Eingangsgrößen verzerren und für positive Größen annähernd linear sein. Besonders ist auch das der Algorithmus viele Parameter zur Ansteuerung bietet. Es gibt den Parameter *Gain* welcher den Grad an Verzerrung einstellt. Mit *Q* kann der Arbeitspunkt eingestellt werden. Mit *dist* kann das Oberton verhalten verändert werden. Zusätzlich ist ein Hochpassfilter implementiert, welcher den Gleichspannungsanteil entfernen soll und ein Tiefpass, welcher die Kapazitäten emulieren soll. Um das ganze abzurunden, wurde noch ermöglicht das unbearbeitete Signal zum Verzerrten Signal dazu zumischen.

Bei den angegebenen Einstellungen für *Q* und *dist* erhält meine eine Verzerrung, welche als Gleichrichtung bekannt ist. Es ist aber keine echte Gleichrichtung, da Q nicht null seien kann. Das Obertonverhalten ist anders als bei einer Gleichrichtung. Werte kleiner Null und größer als minus Eins sind sinnvoll für den Parameter *Q*. Niedrige Werte von *dist* erzeugen Softclipping und hohe Werte erzeugen Hard Clipping.

Während Abbildung 12 einen Extremfall darstellt, und somit die Vielseitigkeit des Algorithmus darstellen soll, so stellt Abbildung 13 den Normalfall dar. Der Algorithmus verändert nur die negative Halbwelle. Besonders ist, dass über den Parameter dist die Anzahl der Obertöne eingestellt werden kann. Dadurch können schwache Verzerrungen, aber auch sehr starke Verzerrungen erzielt werden. Die Parameter Q und dist bieten dem Anwender verschiedene Gestaltungsmöglichkeiten, die andere Algorithmen nicht bieten. Es kann ein Soft- und ein Hard Clipping der negativen Halbwelle entstehen, sowie eine Gleichrichtung. Damit ist der Algorithmus eine Kombination aus anderen Algorithmen, die in dieser Arbeit vorgestellt werden. Eine Messung und Analyse des Obertonverhaltens wird durch die Vielseitigkeit erschwert und erscheint wenig sinnvoll. Das Konzept dieser Verzerrung ist trotzdem in dieser Sammlung an Verzerrungen einzigartig.

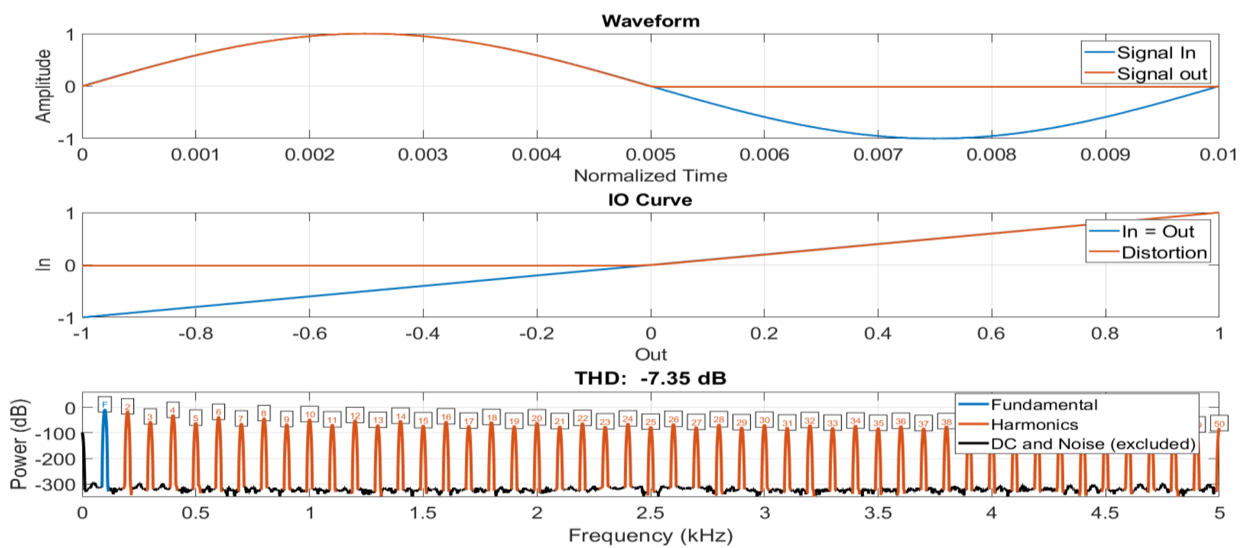


Abbildung 11 Röhrenverzerrung nach Bendiksen Q=-0.01 Dist= 100

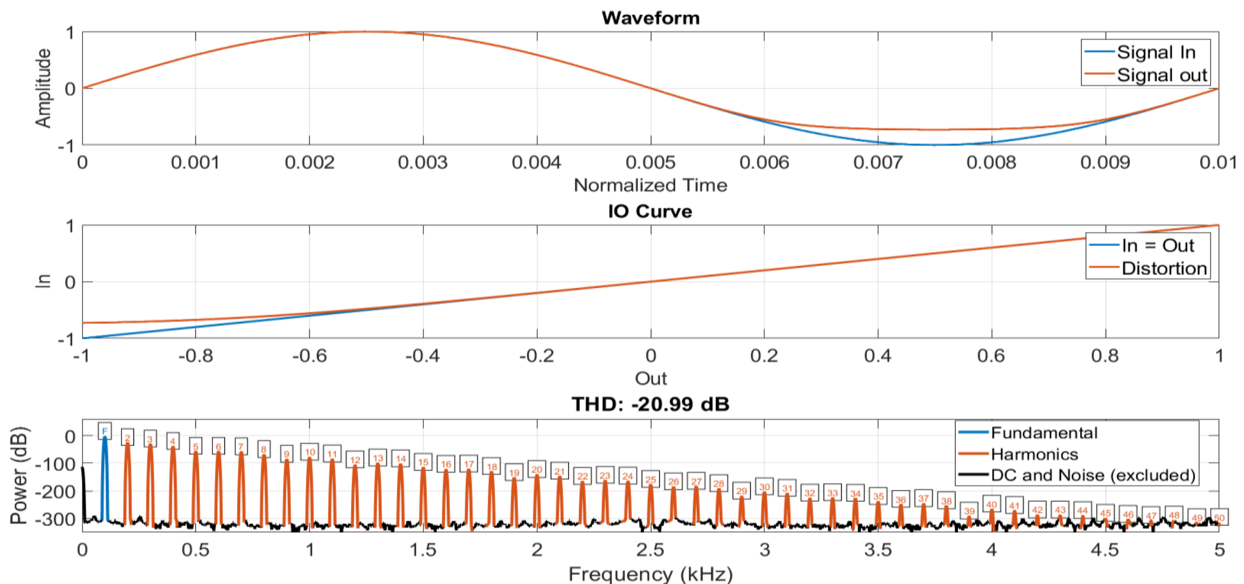


Abbildung 12 Röhrenverzerrung nach Bendiksen Q=-0.75 Dist= 10

2.3.6 Diode

Der Code für die Berechnung der Verzerrung der Diode stammt von Tarr (2018) und ist im Anhang zu finden [siehe Diode (Shockley 1949; Tarr 2018)].

„A circuit component found in audio electronics is the diode. It can be used for nonlinear processing and is found in many distortion guitar pedals and amplifiers. The Shockley diode equation (Shockley, 1949) is a mathematical formula for describing the processing of a diode: $I = I_s \cdot (e^{\frac{V_D}{\eta \cdot VT}} - 1)$.“ (Tarr 2018)

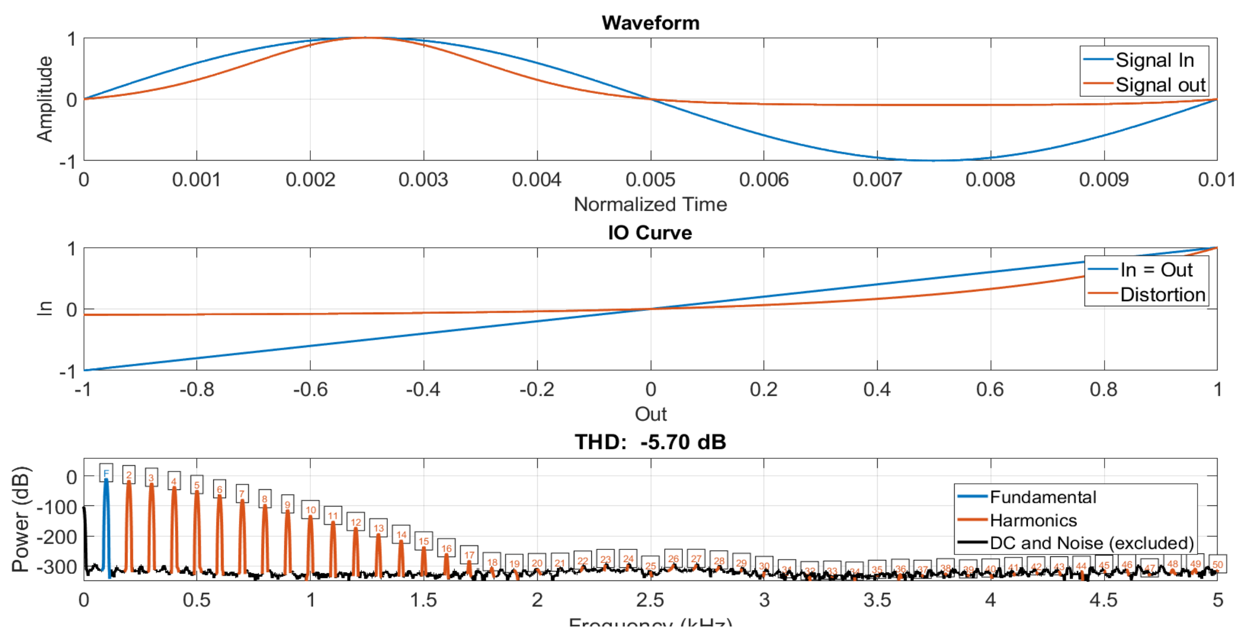


Abbildung 13 Dioden-Verzerrung (Shockley 1949; Tarr 2018)

Tarr (2018) benutzt eine Version dieser Formel, welche für eine Audioanwendung optimiert ist. Hauptsächlich bewirkt diese Optimierung, dass das Ausgangssignal zwischen -1 und 1 liegt, welches zwingend notwendig ist. Diese Verzerrung fügt dem Eingangssignal kontrolliert, gerade und ungerade Obertöne hinzu. Der Pegelabfall der höheren Harmonischen ist ähnlich wie bei Araya und Suyuma (1996). Durch den starken Pegelabfall wird der Alias-Effekt abgeschwächt. Die Diode wird in diesem Fall als Gleichrichter verwendet Die negative Halbwelle verschwindet. Die positive Halbwelle wird verändert.

2.3.7 kubisches Softclipping

Die Transferkennlinie stammt von (Sullivan 1990; Smith III 2008)

$$f(x) = \begin{cases} -\frac{2}{3}, & x \leq -1 \\ x - \frac{x^3}{3}, & -1 \leq x \leq 1 \\ \frac{2}{3}, & x \geq 1 \end{cases}$$

Formel 8 kubisches Softclipping (Sullivan 1990; Smith III 2008)

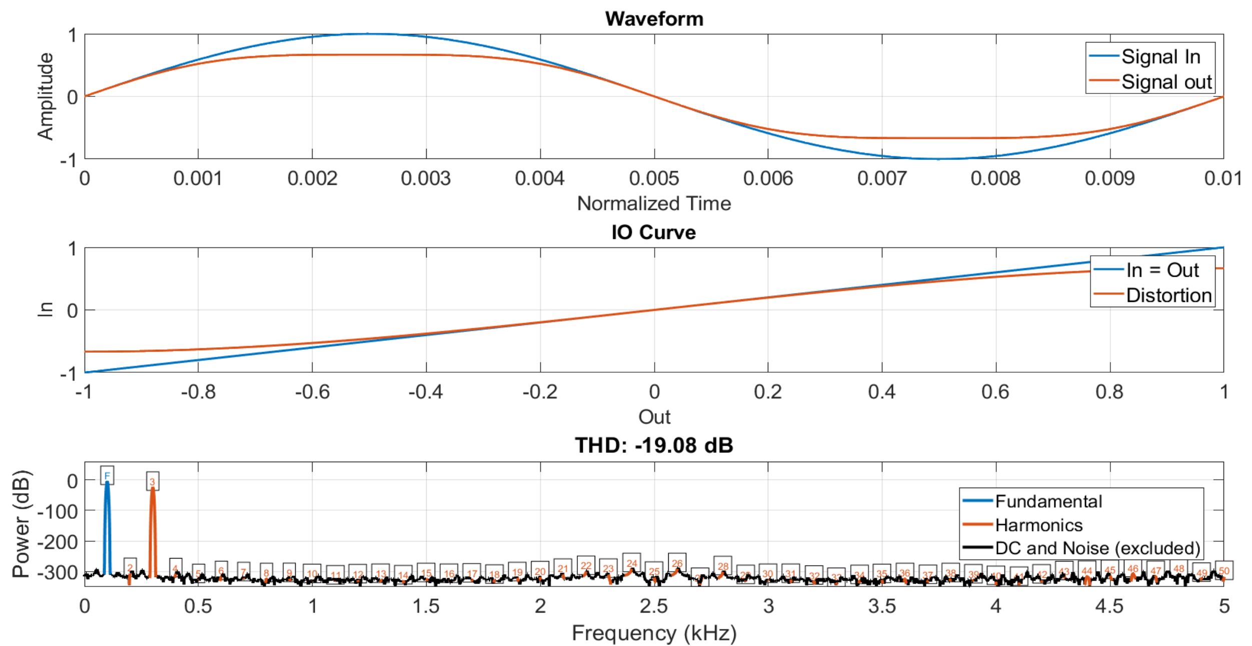


Abbildung 14 kubisches Softclipping (Smith III 2008; Sullivan 1990)

Dieser Algorithmus ist dem von Araya und Suyuma sehr ähnlich. Bei beiden Algorithmen entsteht genau ein ungerader Oberton. Ein offensichtlicher unterschied ist das bei diesem Algorithmus das Signal komprimiert wird. Dies kann aus musikalischer Sicht sinnvoll sein. Viel interessanter ist allerdings das sich die einfache Formel gut modifizieren lässt, um eigene Verzerrungsalgorithmen zu erschaffen. In Zölzer et al. (2002) wird die Tape-Saturation erklärt. Die veraltete Technik der Tonbandaufnahme wird auch heutzutage noch praktiziert, da ein gewünschter Verzerrungs- und Komprimierungseffekt auftritt. Zölzer beschreibt, wie die Kennlinie einer solchen Verzerren aussieht und untermauert dies leider nicht mit einer Formel. Mit einer modifizierten Form der Verzerrung von Sullivan und Smith lassen sich Kennlinien erzeugen, welche den Beschreibungen von Zölzer stark ähneln. Bei niedriger Eingangsamplitude tritt fast keine Verzerrung auf. Bei einer hohen wird die Positive halbwelle beeinflusst. Der Parameter a beeinflusst das Obertonverhalten stark. Für einen Wert von 3 entstehen der zweite, der dritte und der vierte Oberton. Danach überwiegen die geraden Obertöne. Also eine interessante Mischung aus geraden und ungeraden Obertönen.

$$f(x) = \begin{cases} x - \frac{a^3}{3}, & 0 \leq x \leq 1 \\ x, & x < 0 \end{cases} \qquad a = 2; 2.5; 3$$

Formel 9 Modifizierung des kubisches Softclipping

Für ein a von 2 entstehen überwiegend ungerade Obertöne. Und für ein a von 2,5 entstehen sowohl gerade als auch ungerade Obertöne. Der Einfluss des Parameters a auf den Zeitbereich ist gering und optisch schwer zu erkennen. Aber der Einfluss auf den Frequenzbereich ist hingegen stark. Höhere Werte von a erzeugen mehr Obertöne. Das Obertonverhalten ist abhängig von der Amplitude des Eingangssignales. Mit einer simplen Formel entsteht eine hochwertige Verzerrung, welche zusätzlich modifiziert werden kann. Ob diese Verzerrungen dem einer Tonbandmaschine ähneln, ist fragwürdig, da kein zu vergleichender Algorithmus vorliegt.

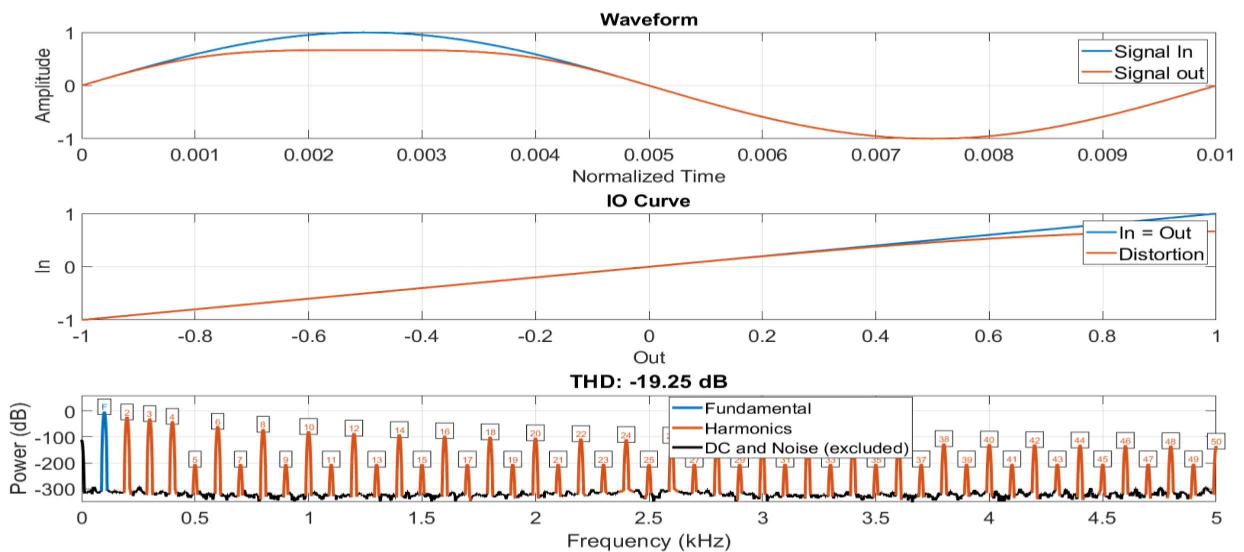


Abbildung 15 Modifizierung des kubisches Softclipping für a = 3

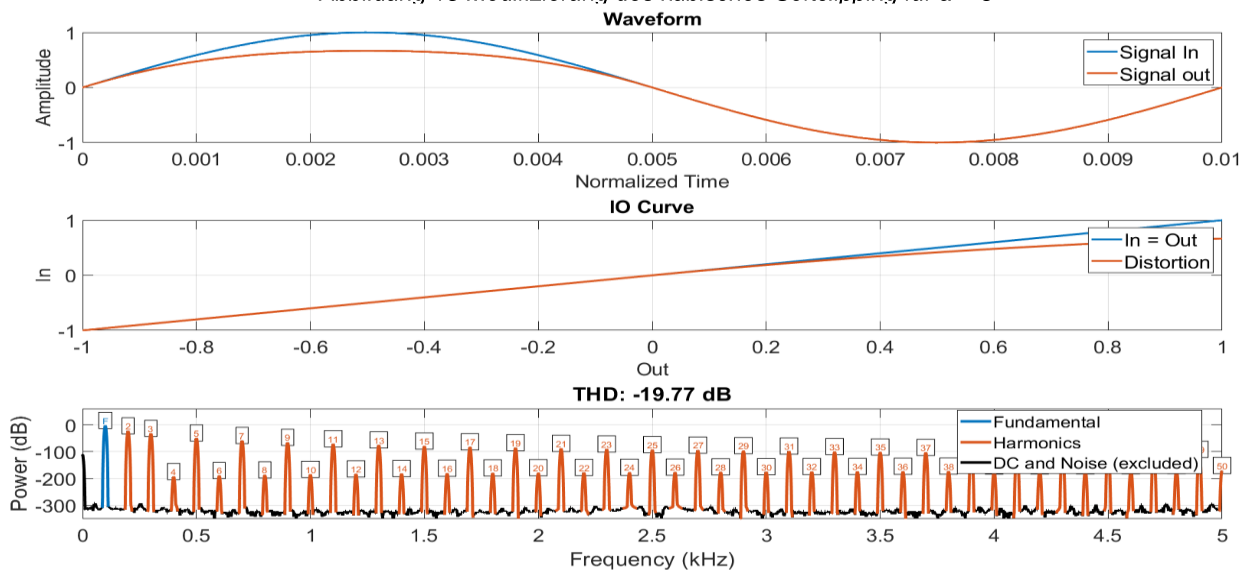


Abbildung 16 Modifizierung des kubisches Softclipping für a = 2

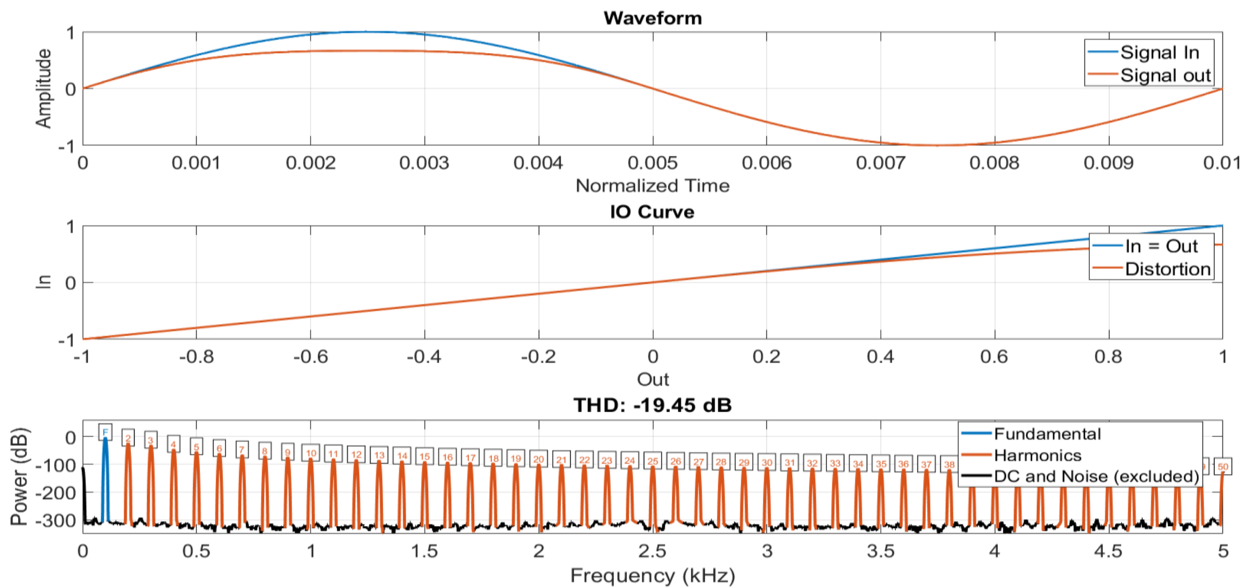


Abbildung 17 Modifizierung des kubisches Softclipping für a = 2.5

2.3.8 tangentiales Softclipping

Viele Verzerrungsalgorithmen basieren auf dem Tangens. Schnellvertretend werden die verzerrenden Eigenschaften des hyperbolischen Tangens, des Arkustangens und einer Approximation des hyperbolischen Tanges vorgestellt. Genannte Funktionen eignen sich perfekt für eine Verzerrung und sind sehr einfach zu implementieren.

Der Code für die Berechnung dieser Verzerrung stammt von Yeh (2009)

$$f(x) = \tanh(x)$$

Formel 10 hyperbolischer Tangens als Verzerrer

Es entstehen ausschließlich ungerade Obertöne in limitierter Anzahl. Das Signal wird komprimiert. Der Pegel des Eingangssignals hat einen Einfluss auf das Obertonverhalten. In Abbildung 20 kann man erkennen, dass das Obertonverhalten der Verzerrung der Approximation des hyperbolischen Tangens sich von der Verzerrung des hyperbolischen Tangens unterscheidet. Es entstehen Obertöne in viel größerer Anzahl und es entstehen zusätzlich gerade Obertöne mit sehr niedriger Amplitude. Die Approximation ersetzt den hyperbolischen Tangens also nicht sondern erzeugt einen etwas anderen Effekt. Der

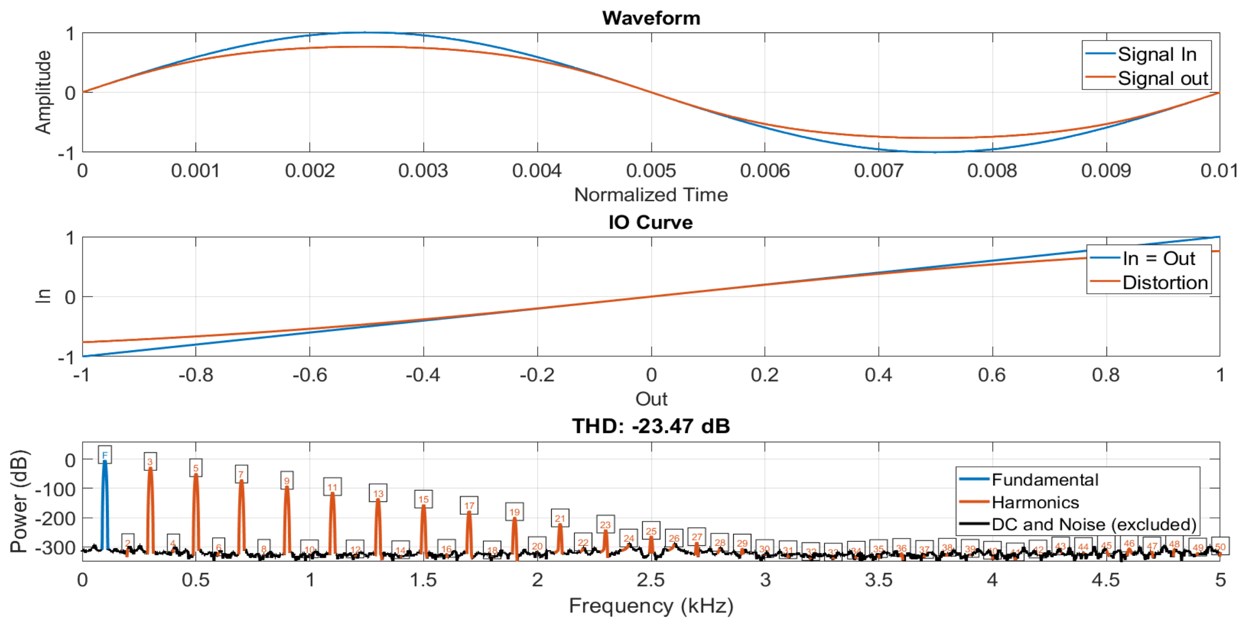


Abbildung 18 hyperbolischer Tangens als Verzerrer

Parameter n kann genutzt werden, um eigene Verzerrungsexperimente zu beginnen. Bei einem Wert von 2,5 ist die Approximation dem Tangens aber am ähnlichsten.

Yeh (2009) nennt Abels (2006) Approximation des hyperbolischen Tangenz.

$$f(x) = \frac{x}{(1 + |x|^n)^{1/n}} \qquad n = 2{,}5$$

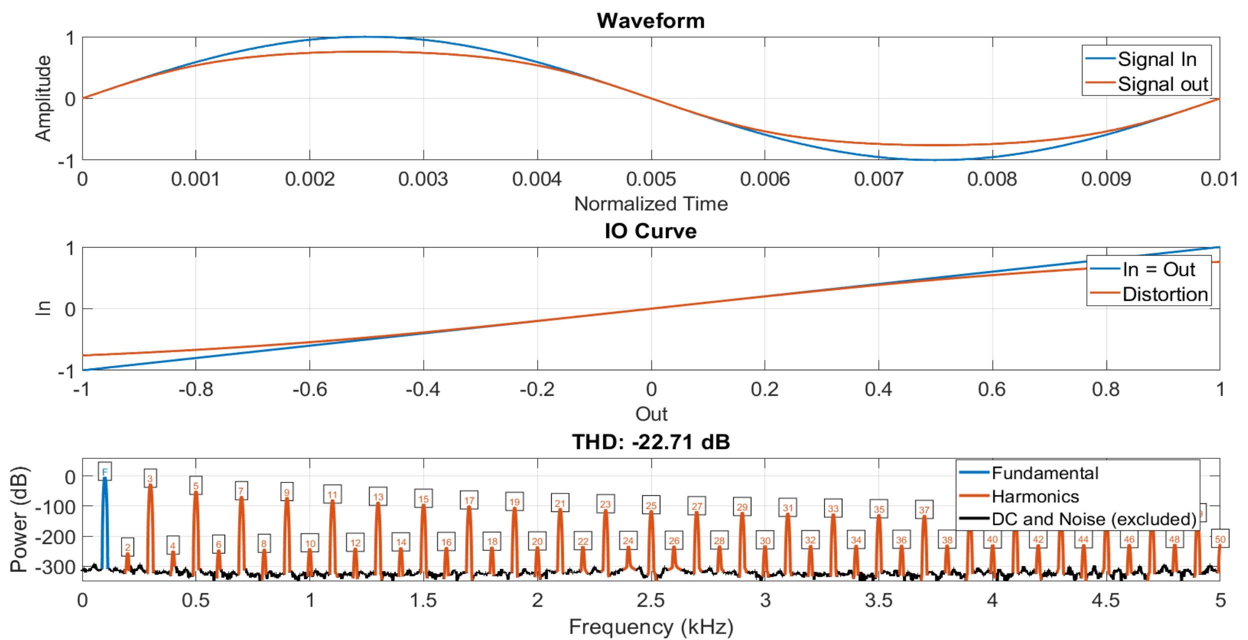


Abbildung 19 Approximation des hyperbolischen Tangens als Verzerrer

Der Arkustangens weist das gleiche Wirkprinzip auf wie der hyperbolische Tangens, erzeugt aber etwas mehr Obertöne. Mit der Formel von Tarr (2018) wird der Arkustangens auf eins normalisiert. Dies ermöglicht die Einführung des Parameters *Drive.* Durch diesen Parameter kann die Sinusschwingung in eine Rechteckschwingung verwandelt werden. Das Resultat ist eine extreme Verzerrung. Je höher *Drive* desto mehr Obertöne entstehen und desto mehr wird das Ausgangssignal einem Rechteck ähnlich.

$$f(x) = \frac{2}{\pi} \cdot tan^{-1}(drive \cdot x)$$

Formel 11 Arkustangens als Verzerrer

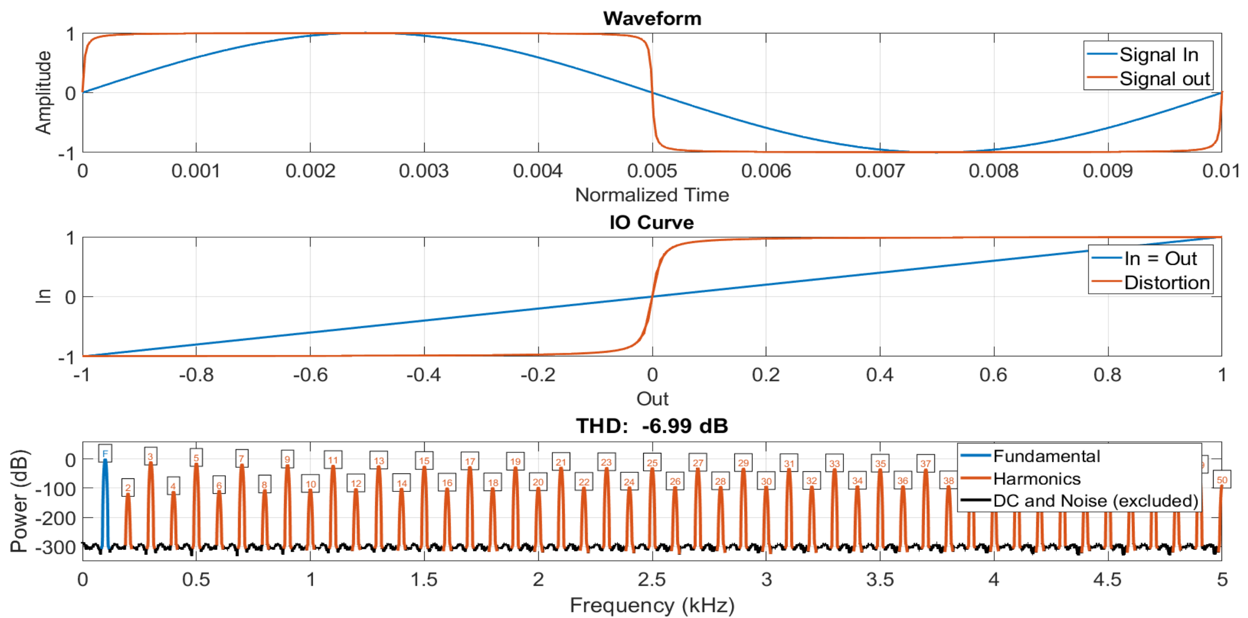


Abbildung 20 Arkustangens als Verzerrer mit Drive = 100

2.4 Vergleich der Verzerrungsalgorithmen

Verzerrungsalgorithmus			**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
Obertonverhalten		gerade	-100	0	-300	-300	0	0	0	0	-300	0	-200	0	-300	-200	-100	-100		
Amplitude in dB stark gerundet		ungerade	0	0	0	0	0	-100	-100	0	0	-200	0	0	0	0	0	0		x
		dynamisch			x	x	x			x	x	x	x	x	x	x	x	x		
		Anzahl	sehr viele	sehr viele	1	wenige	wenige	viele	viele	viele	1	viele	viele	viele	viele	wenige	wenige	sehr viele	0	sehr viele
Maximale Amplitude	1 kHz Sinus	x1	-52	-60,5	-266	-198,5	-266	-56	-56	-54	-266	-110	-90	-102	-111	-260	-212	-30	-266	-28
des Alias-Effekts	"Gitarre"	x2	-68	-77	-270	-270	-270	-74	-73	-80	-270	-144	-116	-133	-151	-270	-270	-43	-270	-37
unter 20 kHz in dB		x4	-83	-92	-274	-275	-270	-87	-87	-82	-275	-172	-138	-158	-183	-274	-274	-57	-270	-45
		x8	-95	-103	-285	-285	-285	-100	-100	-88	-285	-198	-157	-181	-213	-285	-285	-79	-285	-51
		x16	-107	-115	-285	-285	-285	-112	-112	-94	-285	-223	-176	-203	-241	-285	-285	-116	-285	-58
		x32	-119	-128	-285	-285	-285	-124	-124	-100	-285	-248	-195	-224	-268	-285	-285	-183	-285	-64
	20 kHz Sinus	x1	-13	-15	-20	-11	-6	-9	-9	-11	-20	-20	-20	-20	-23	-24	-25	-10	-240	-9
	"Breitband"	x2	-27	-21	-240	-19	-26	-23	-23	-17	-240	-37	-46	-42	-47	-45	-44	-14	-240	-14
		x4	-33	-34	-240	-38	-86,5	-36	-36	-26	-240	-67	-63	-67	-67	-88	-80	-20	-240	-19
		x8	-50	-41	-240	-98	-240	-50	-50	-47	-240	-97	-80	-91	-95	-173	-147	-26	-240	-25
		x16	-62	-65	-240	-240	-240	-62	-62	-64	-240	-121	-99	-113	-125	-240	-240	-34	-240	-31
		x32	-74	-83	-240	-240	-240	-74	-74	-83	-240	-144	-117	-134	-152	-240	-240	-43	-240	-37
Gleichanteil				x			x	x	x	x		x	x	x						
Ansteuerung des Drive-Parameters			0 - 1	0 - 1	0 - 1	0 - 1	0 - 1	-	-	0 - 1	0 - 1	0 - 1	0 - 1	0 - 1	0 - Inf	0 - Inf	0 - Inf	0 - Inf		

1.Hardclipping 2.PositiveHalvewaveClipping 3. Araya und Suyuma 4.Araya und Suyuma (Reihenschaltung) 5.Diode 6.Halbwellengleichrichter 7.Vollwellengleichrichter 8.Diodic et al 9. Sullivan and Smith
10. Sullivan and Smith modified a = 3 11. Sullivan and Smith modified a = 2 12. Sullivan and Smith modified a = 2.5 13. aprox tanh 14. tanh 15. atan drive = 1 16. atan drive 100 17.Sinusschwingung 18. Rechteckwelle

Tabelle 2 vergleichende Darstellung der Verzerrungsalgorithmen

Zuerst wird die Ergebnisdarstellung erklärt, bevor Erkenntnisse geteilt werden. Der Parameter Obertonverhalten, soll es ermöglichen in kurzer Zeit, zu erkennen, ob der Algorithmus gerade oder ungerade Obertöne erzeugt. Dabei wird die Amplitude der Obertöne sehr stark gerundet, dadurch kann man schnell erkennen welche Obertöne für den Klang der Verzerrung wichtig sind und welche eher weniger. Eine Amplitude von 0 dB bedeutet, dass die Obertöne in etwa so laut sind wie die Grundfrequenz. Schon bei einer Amplitude von -100 dB ist fragwürdig wie hörbar dieser Obertonanteil ist. Diese Arbeit beschäftigt sich nicht mit Psychoakustik. Das Betrachten von Maskierungseffekten und der Hörbarkeit verschiedener Obertöne würde in diesen Bereich fallen. Eine gemessene Amplitude von -300 dB entsteht durch das Grundrauschen, dies bedeutet das dieser Obertonanteil praktisch nicht vorhanden ist. Der Parameter *dynamisch* markiert Verzerrungen, bei denen ein Eingangssignal mit einer hohen Amplitude mehr Obertönen erzeugt als ein Eingangssignal mit niedriger Amplitude. Dieses Verhalten lässt sich auch auf den Parameter *Drive* übertragen da dieser lediglich die Eingangsamplitude verstärkt. Verzerrungen, welche dynamisches Verhalten aufweisen, sind geeignet für Instrumentalisten, da diese die Lautstärke ihres Instruments als künstlerisches Mittel nutzen. Dadurch kann die Verzerrung das Instrument erweitern. Bei einer elektrischen Gitarre ist dies essenziell. Der Parameter *Anzahl* soll nur kurz dem Betrachter der Tabelle zeigen wie viele Obertöne von dem Algorithmus grob zu erwarten sind. Die *Maximale Amplitude des Alias-Effekts unter 20 kHz* wurde für verschiede Überabtastungsfaktoren und mit zwei verschiedenen Testsignalen gemessen. Yeh (2009) erklärt anhand eines Rechtecksignales , dass ein Überabtastungsfaktor von 8 ausreichend sei. Viele Verzerrungen verwandeln ein Sinusförmiges Eingangssignal bei starker Verstärkung in ein Rechtecksignal. Der Wert für *maxAlias* den er erhält sind -50 dB. Dadurch entsteht der Richtwert von -50 dB. Überabtastungsfaktoren die niedriger als -50 dB an *maxAlias* erzeugen werden als brauchbar befunden. Diese sind mit Blau markiert. Rechts in der Tabelle wurde eine Sinusschwingung (17) und eine Rechteckschwingung (18) gemessen. Die Sinusschwingung stellt dabei den Fall dar, bei welchen kein Aliasing auftritt. Wenn eine Verzerrung für einen Überabtastungsfaktor den gleichen Wert wie die Sinusschwingung aufweist so entsteht kein Aliasing. Die Rechteckschwingung stellt dabei den Extremfall dar. Auffällig ist das für das von Yeh vorgeschlagene Testsignal „*Gitarre*“, die 1 kHz Sinusschwingung, fast alle Verzerrungen diesen Test problemlos bestehen außer die tangentiale Verzerrung. Da diese eine Sinusschwingung in eine Rechteckschwingung verwandeln kann erzeugt sie sehr viel Aliasing und benötigt eine hohe Überabtastung. Diese Verzerrung wurde mit einem Verstärkungsfaktor, *Drive*, von 1 und 100 gemessen. Für Drive = 1 entsteht fast kein Aliasing. Für Drive = 100 wird eine vierfache Überabtastung benötigt, um den Richtwert zu unterschreiten. Bei einer tangentialen Verzerrung kann das Eingangssignal unendlich stark verstärkt werden und wird vom Tangens trotzdem wieder in den benötigten Amplitudenbereich gewandelt. Yehs Rechenbeispiel zeigt, dass eine Überabtastung von 8 dafür ausreichend sei. Alle anderen Verzerrungen, die gemessen wurden, sind viel subtiler und erzeugen zu wenig Obertöne, um ein starkes Aliasing

hervorzurufen. Also für eine Sinusschwingung bei 1 kHz ist bei den meisten Algorithmen keine Überabtastung nötig. In der Literatur wird eine 8-fache Überabtastung empfohlen. Dies liegt wahrscheinlich daran, dass Gitarren höhere Harmonische erzeugen und damit höhere Frequenzen verzerrt werden. Dies würde in einem stärkeren Aliasing resultieren. Zusätzlich sind manche dieser Verzerrungsalgorithmen darauf optimiert wenig Aliasing zu erzeugen, da ein starker Abfall der Amplitude der Obertöne stattfindet. Für das Test-signal „*Breitband*", also den 20 kHz Sinus entstehen andere Ergebnisse als für das Test-signal „*Gitarre*". Die Bandbreitenvergrößerung der Verzerrung ist bei dieser hohen Frequenz viel größer als bei der 1 kHz Sinusschwingung. Wenn eine Verzerrung gerade Obertöne erzeugt, so würde der erste gerade Oberton bei 40 kHz entstehen. Allerdings ist bei einer Abtastrate von 44100 Hz die Nyquist-Frequenz bei 22050 Hz. An dieser Frequenz wird der Oberton in den hörbaren Bereich gespiegelt. Die gespiegelte Frequenz entsteht bei 4100 Hz. 4100 Hz ist kein harmonisches vielfaches von 20 kHz und wird somit als unmusikalisch empfunden. Der erste Oberton hat für gewöhnlich die größte Amplitude von allen Obertönen. Deswegen erzeugt beim Testsignal „*Breitband*" selbst eine Verzerrung, welche nur einen Oberton erzeugt, ein starkes Aliasing. Das Plugin soll für eine Anwendung von 20 bis 20 kHz entwickelt werden. Die Ergebnisse des hochfrequenten Testsignals liefern Überabtastungsfaktoren bei denen das Aliasing so niedrig ist wie bei den Gitarrenverzerrern die Yeh (2009) beschreibt. Dies soll eine qualitativ hochwertige Breitbandanwendung des Plugins ermöglichen. Das Ergebnis der Messung ist das die meisten Verzerrungen ein 8-faches Oversampling benötigen (1) (4 - 7). Die Verzerrungen, die nur einen Oberton erzeugen benötigen ein zweifaches Oversampling (3) (9). Die Algorithmen, die sehr viele Obertone erzeugen benötigen ein 16-faches Oversampling (2) (8). Einige Algorithmen erzeugen aber auch wenig Obertöne und benötigen nur ein vierfaches Oversampling (10 - 15). Die extreme tangentiale Verzerrung erzeugt zu viel Aliasing, um qualitativ hochwertig implementiert werden zu können (16). Da niedrige Überabtastungsfaktoren weniger Rechenleistung in Anspruch nehmen als hohe, gibt es manche Algorithmen die recheneffizienter implementiert werden können als andere. Der Parameter *Gleichanteil* weist auf das Vorhandenseien eines Gleichanteils hin. Ein Gleichanteil kann durch die Subtraktion seiner Amplitude und eine anschließende Multiplikation um einen konstanten Faktor einfach und effektiv für jedes Sample eines Audiosignals entfernt werden. Die Multiplikation, ist eine Verstärkung oder Abschwächung, welche die ursprüngliche Signalstärke nach der Offset-Kompensation, wieder herstellen soll. Dieses Verfahren ist recheneffizient, einfach und bei guter Umsetzung fehlerfrei. Das fertige Audioplugin bietet einen, wahlweise einschaltbaren, Filter, welcher alle Frequenzen, unter 20 Hz, mit einer hohen Flankensteilheit entfernt. Der Gleichanteil ist nicht hörbar. Aus signaltechnischer Sicht ist er aber ein Makel und deswegen wird die Entfernung des Gleichanteils im Plugin optional angeboten. Der letzte Parameter der Tabelle, *Ansteuerung des Drive-Parameters,* hat eine Relevanz für die Implementierung der Verzerrung. *Drive* stellt den Grad der Verzerrung ein. Dies geschieht über eine Verstärkung des Eingangssignales. Wenn in der Tabelle „0-1"

angegeben ist, so wird das Eingangssignal bei korrekter Implementierung mit einer Zahl zwischen 0 und 1 multipliziert. Hintergrund ist, dass diese Verzerrungen fehlerhaft sind, wenn das Eingangssignal eine größere Amplitude als eins aufweist. Ganz im Gegensatz dazu gibt es Verzerrungen, die ein Eingangssignal mit theoretisch unendlich großer Amplitude fehlerfrei verarbeiten können, diese sind mit „0 - inf “ markiert. Andere Verzerrungen sind nicht mit einem solchen Parameter ansteuerbar, diese werden durch „–“ markiert.

2.5 Fazit der Analyse der Algorithmen

Zusammenfassend kann gesagt werden, dass alle ausgewählten Algorithmen die technischen Mindestanforderungen erfüllen. Das Ausgangssignal liegt bei jedem Algorithmus zwischen den Amplitudenwerten von -1 und 1. Dies gewährleistet, dass das Signal nicht am Ausgang von der DAW verzerrt wird. Da die so erzeugte Verzerrung den Klang des Verzerrungsplugins verfälschen würde. Die Algorithmen haben auch keine Berechnungsfehler, welche unerwünschte Störgeräusche, Knacken, starke Lautstärkeunterschiede oder sonstiges erzeugen würden. Ein klassischer Berechnungsfehler ist z.B. das komplexe Zahlen entstehen, wo keine entstehen sollen. Algorithmen mit so offensichtlichen Fehlern haben es nicht in diese Auswahl an Verzerrungen geschafft. Auffällig ist das es deutlich einfacher ist ungerade Obertöne zu erschaffen als gerade. Gerade Obertöne lassen sich gut durch das entfernen einer Halbwelle, oder durch das Verändern von nur der negativen oder nur der positiven Halbwelle erschaffen. Ungerade Obertöne entstehen immer dann, wenn beide, die positive und die negative, Halbwelle gleich verändert werden. Viele Algorithmen sind eine Kombination aus beiden. Es wurde kein einziger Algorithmus gefunden, der ausschließlich einen geraden Oberton erschafft. Möglichweise existiert so ein Algorithmus. Dieser wurde nur auch nach umfangreichen Recherchen nicht gefunden. Es gibt Algorithmen die ausschließlich gerade oder ungerade Obertöne erzeugen. Die meisten Algorithmen erzeugen sowohl gerade als auch ungerade Obertöne. Meistens ist aber entweder der gerade oder der ungerade Obertonanteil signifikant leiser als der andere. Manche Algorithmen erzeugen gerade und ungerade Obertöne, deren Amplitude in etwa gleich stark ist. Alle Algorithmen erzeugen harmonische Vielfache der Grundfrequenz. Es gibt keinen Algorithmus, welcher Vielfache erzeugt, welche nicht in ganzzahligen Verhältnissen zur Grundfrequenz stehen. Es wurde umfangreiches Wissen über die Vielfalt an Verzerrungsalgorithmen und deren Funktionsweise gewonnen. Es wurden Möglichkeiten vorgestellt, mit denen neue Algorithmen auf einfache Art und Weise kreiert werden können. Eine sinnvolle Implementierung der Algorithmen wurde durch intensive Messungen und Betrachtungen ermöglicht. Das Testsignal „*Gitarre*“ erwies sich als wenig hilfreich bei der Betrachtung des Alias-Effekts. Ein Grund dafür ist, dass die Algorithmen aus professionellen Quellen stammen und damit etwas subtileres Obertonverhalten aufweisen als semiprofessioneller Code

aus dem Internet. Es ist einfacher stark zu verzerren als gezielt wenig Obertöne zu erzeugen. Das Testsignal „*Breitband*“ lieferte dafür genau die Informationen, die für die Implementierung der Algorithmen wichtig sind. Mit den gewonnenen Informationen können diverse Verzerrungs-Plugins erstellt werden, für unterschiedliche Anwendungsgebiete. Die Vergleichende Tabelle ermöglicht es aufgrund von Daten objektive Entscheidungen zu fällen. Benötigt ein Plugin eine subtile Verzerrung mit wenig Aliasing oder eine starke klangliche Veränderung? Anhand der Tabelle kann entschieden werden. In den meisten Plugins ist Verzerrung nur ein Effekt von vielen und die Rechenleistung wird für andere Prozesse mehr benötigt. Eine sachliche Abwägung aller Parameter der Vergleichenden Tabelle ist deswegen notwendig und zielführend.

3 Erstellung von Audio-Plugins in MATLAB

Es ist tatsächlich möglich mit MATLAB Audio-Plugins zu erstellen. Der Fokus liegt auf der Entwicklung von Prototypen. Ein Plugin, welches durch MATLAB erschaffen wurde, kann alle Operationen der digitalen Signalverarbeitungen fehlerfrei durchführen. Der Fokus liegt also darauf die Signalkette zu erschaffen und verschiedene Algorithmen auszuprobieren. Das Userinterface, des Plugins, ist dabei sehr simpel und kann nur wenig verändert werden. Das Plugin wird also nicht großartig aussehen und in der Anwendbarkeit grob sein, aber der Klang kann hochwertig sein. Da die objektorientierte Programmierung in MATLAB einfacher ist als in C++, wird MATLAB häufig dafür verwendet, um Ideen auszuprobieren. Der große Vorteil ist, dass der MATLAB Syntax dem von C++ ähnlich ist. Ideen, die in MATLAB funktionieren, können also so oder so ähnlich auch in C++ funktionieren. Professionelle Audio-Plugins werden in JUCE geschrieben. JUCE ist ein Framework für C++. Die Funktionen und Befehle in JUCE sind speziell für das Erschaffen von Medien-Programmen optimiert. Es können optisch ansprechende Interfaces mit komplexen Interaktionsmöglichkeiten erstellt werden. Zuerst muss aber erstmal ein Plugin mit einer interessanten funktionsweise erschaffen werden, damit es sich lohnt großen Entwicklungsaufwand zu betreiben. Genau dafür ist MATLAB gut geeignet.

3.1 Einführung MALTAB objektorientierte Programmierung

3.1.1 Warum objektorientierte Programmierung?

Die objektorientierte Programmierung in MATLAB hat den großen Vorteil, dass diese das Kompilieren in C++ Code ermöglicht. Der Compiler benötigt diese spezielle Syntax, um die Übersetzung von der einen Sprache in die andere Sprache vornehmen zu können. Dies ermöglicht es simple Programme, Apps und Audio-Plugins mit MALTAB zu schreiben. Audio-Plugins können so direkt in MATLAB geschrieben werden und durch den Compiler direkt im richtigen Format exportiert werden. Auf Windows sind Audio-Plugins durch die Dateiendung *.dll* markiert. Auf einem Mac wird die Dateiendung .vst verwendet. Audioplugins müssen für die jeweilige Plattform optimiert und im richtigen Dateityp exportiert werden. Das Audioplugin wird auf Windows 10 entwickelt und soll auch für Windows 10 primär funktionieren. Eine Version für Mac ist mit zusätzlichem Aufwand aber möglich. MATLAB ist primär für seine prozedurale Programmierung bekannt. Das bedeutet das jeder Befehl hintereinander abgearbeitet wird. Dies ermöglicht es schnell Ideen auszuprobieren. Bei der prozeduralen Programmierung können

Variablen einfach zugewiesen werden und Operationen an jeder Stelle des Skriptes durchgeführt werden. Dies ist alles bei objektorientierter Programmierung nicht der Fall. Der Code wird streng definiert und alles muss an der richtigen Stelle im Skript passieren. Dies wirkt alles zuerst sehr aufwändig. Es bietet allerdings viele Vorteile. Der größte Vorteil ist, dass der Funktionsumfang sich verändert. Viele Funktionen und Befehle aus der prozeduralen Programmierung mit MATLAB funktionieren nicht mehr, dafürkönnen aber andere Funktionen und Befehle genutzt werden. Ein Beispiel dafür ist, dass man Simulink Objekte benutzen kann. Es gibt eine große Anzahl an Objekten und Funktionen, die die meisten Operationen, welche für die Digitale Signal Verarbeitung benötigt werden, durchführen können. Diese Objekte und Funktionen können für Echtzeit-Anwendungen verwendet werden. Ein weiterer Vorteil der objektorientierten Orientierung ist, dass durch die strenge Definierung, manche Flüchtigkeitsfehler vermieden werden können. Es ist nicht ohne weiteres möglich, durch vertippen eine neue Variable zu erschaffen. Es wird der Programmierer eingeschränkt, aber dadurch wird auch der Endnutzer eingeschränkt. Das ist gut da der Endnutzer nur die Parameter kontrollieren können soll, die der Programmierer dafür vorgesehen hat. Der letzte große Vorteil ist, dass C++ Code schneller und stabiler läuft als MATLAB Code.

3.1.2 Verwendete Toolboxen und Versionsnummern

Für das gesamte Projekt wurde eine kostengünstige Studentenlizenz von MATLAB verwendet. Die verwendete Version ist: 9.8.0.1451342 (R2020a) Update 5. Bei Verwendung von anderen MATLAB-Versionen können Modifikationen des Codes nötig sein. Die Programmierung geschah unter Windows 10 auf einem ThinkPad. Verwendete Toolboxen sind: MATLAB Audio Toolbox, MATLAB DSP System Toolbox, MATLAB Coder.

Audio Toolbox

Die Audiotoolbox ist notwendig, um Audio-Plugins zu erschaffen. Die wichtigsten Features sind eine Testumgebung und die Validierung von Code. Mit dem Befehl

audioTestBench myAudioPlugin

wird das Testareal geöffnet. Der code in der Datei *myAudioPlugin.m* kann nun mit verschiedenen Testsignalen geprüft werden. Es können .wav Dateien, Sinus-Sweeps, Rauschen etc. durch das Plugin übertragen werden. Es gibt ein Oszilloskop und einen FFT-Analysator. Verschiedenste Werte können dort berechnet werden unter anderem der Klirrfaktor. Diese Testumgebung beschleunigt den Arbeitsprozess, da man den geschriebenen Code testen kann, ohne ihn zu exportieren und in die DAW zu laden.

DSP System Toolbox

Diese Toolbox ist für digitale Signalverarbeitung optimiert. Viele bekannte Methoden und Wirkprinzipien sind als Objekt oder als Funktion verfügbar. Dies beschleunigt den Arbeitsprozess stark, da es dadurch nicht notwendig ist grundlegende aber mathematisch aufwändige Berechnungen selbst zu schreiben. Viele Befehle aus der prozeduralen MATLAB-Programmierung funktionieren nicht für objektorientierte Programmierung mit variablen Eingangsgrößen. Zum Beispiel funktionieren die „normalen" Filter nicht, aber diese Toolbox beinhaltet andere Filter, die funktionieren.

Coder

Nur in der Kombination mit der Audio Toolbox und dem Coder können Plugins in C++ kompiliert werden und anschließend in einer DAW geladen werden. Damit der Coder funktioniert muss eine Programmierumgebung für C++ installiert sein. Die gewählte Umgebung ist *Microsoft Visual Studio.* Nur mit einer speziellen Version funktioniert dies Kommunikation zwischen *MATLAB* und *Visual Studio.* Dieser Forenbeitrag ist eine notwendige Installationshilfe und fasst die Materie gut in einer Anweisung zusammen: „Install Visual Studio 2017 with Windows 10 SDK and VC++ 2017 toolset" (itectec N/A). Beim Durchführen eines Updates für Visual Studio muss darauf geachtet werden, dass die genannte SDK und das toolset nicht gelöscht werden! Dies wird massive Probleme mit sich bringen und die Kommunikation von MATLAB und Visual Studio stark stören.

3.1.3 Grundgerüst für ein Audio-Plugin

```
classdef myAudioPlugin < audioPlugin
    methods
        function out = process(~,in)
            out = in;
        end
    end
end
```

Skript 1 Minimalanforderungen des Codes zur Plugin-Erstellung

Dies ist das simpelste Plugin, welches in der offiziellen MATLAB-Dokumentation unter dem Suchbegriff „*audioPlugin class*" gefunden werden kann (MATLAB 2021). Exakt dieser Code kann in eine dll. kompiliert werden und von einer DAW ausgeführt werden. Das Resultat ist ein Plugin, welches keine Regler hat und auch keinen Einfluss auf das Signal aufweist. Dieser Code ist nämlich das Grundgerüst für ein Audioplugin. Von diesem Grundgerüst aus kann man weiterführend neue Funktionen und Variablen hinzufügen. Zuerst muss aber das Grundgerüst verstanden werden. Der Code muss als *myAudioPlugin.m* abgespeichert werden. Der Befehl *classdef* weist MATLAB an eine Klasse zu

erschaffen. Diese Klasse heißt *myAudioPlugin.* An dieser Stelle kann der Name des Plugins eingefügt werden. Rechts daneben steht: „< *audioPlugin*“. Diese Anweisung ist essenziel, da sie die Klasse als Audio-Plugin deklariert. Dadurch übernimmt MATLAB sehr viel Arbeit. In anderen Programmiersprachen müsste man bevor man mit der digitalen Signalverarbeitung anfangen kann, erst einmal grundlegende Funktionen einer Anwendung bauen. Darunter zählt: ein Fenster, dass sich öffnet, mit dem interagiert werden kann, welches man, oben rechts schließen, minimieren, verkleinern/ vergrößern, kann. Diesen gesamten Aufwand übernimmt MATLAB. Funktionen in einem Audioplugin sind essenziel, um Aufgaben auszuführen. Diese müssen durch *methods* deklariert werden. Zwischen dem Befehl *methods* und dem dazugehörigen *end* können beliebig viele Funktionen durchgeführt werden. Es darf mehrere *methods* Segmente geben, aber nur einmal ein *classdef* Befehl. Ein Audio-Plugin ist eine Klasse. Dies ist der Syntax einer Funktion aus der Dokumentation:

„function [y1,...,yN] = myfun(x1,...,xM)“ (MATLAB 2021).

Gefolgt nach dem Aufruf *function* erfolgt die Deklarierung des Outputs der Funktion. Der Name *myfun* oder *process* kann geändert werden. Zuletzt werden die Inputs für die Funktion deklariert. Die DAW schickt ein Signal zum Audio-Plugin. Dieses Signal wird in der Variablen *in* abgelegt. Das Ausgangssignal, welches vom Plugin zur DAW geleitet wird, wird in der Variablen *out* abgelegt. Diese beiden Variablen sollten in ihrer Funktionsweise nicht zweckentfremdet werden, da sie für die grundlegende Funktionalität, des Plugins, wichtig sind. Wenn man eine serielle Schaltung bewirken will, so empfiehlt es sich folgenden Syntax in einer Funktion zu verwenden: *x = in ; y=x ; out = y*. Die Inputvariable wird an die Variable *x* weitergegeben. Mit *x* und *y* kann dann eine Bearbeitung des Signals vorgenommen werden. Am Ende wird *y* an *out* durchgegeben.

```
classdef myPlugin < audioPlugin
    properties
        Gain = 1.5;
    end
    properties (Constant)                                %<---
        PluginInterface = audioPluginInterface( ...      %<---
            audioPluginParameter('Gain', ...             %<---
            'DisplayName','Gain', ...                    %<---
            'Mapping',{'lin',0,3}))                      %<---
    end                                                  %<---
    methods
        function out = process(plugin, in)
            out = in*plugin.Gain;
        end
    end
end
```

Skript 2 Einführung in das Ansteuern von Variablen und dem Interface

Auch dieses Beispiel stammt aus der offiziellen MATLAB Dokumentation und kann unter dem gleichen Suchbegriff wie oben gefunden werden (MATLAB 2021). Wenn dieses zweite Grundgerüst verstanden ist, kann man schon simple Plugins schreiben und sich ausprobieren. Dieses Skript erschafft einen Lautstärkeregler. Dieser ist im konkreten Beispiel linear. Es bietet sich an diesen, bei Bedarf, logarithmisch zu gestallten.

Variablen und Zahlentypen

Anhand von diesem Beispiel soll allerdings zuerst einmal das Arbeiten mit Variablen erklärt werden. Zwischen *properties* und dem dazugehörigen *end* können Variablen deklariert werden. Es wird eine Variable mit dem Namen *Gain* angelegt und der numerische Wert von 1.5 wird in dieser abgespeichert. Variablen sind eine Art Container in der man Zahlen abspeichern kann und diese wieder aufrufen kann.

„Alle MATLAB Variablen sind mehrdimensionale *Arrays*, ganz gleich, um welchen Datentyp es sich handelt. Eine *Matrix* ist ein zweidimensionales Array, das häufig für lineare Algebra verwendet wird.“ (MathWorks N/A)

Alle Zahlen werden automatisch als Double-Precision Array angelegt. Dies bedeutet, dass es 64 Bit Gleitkommazahlen sind. Für Audiosignalverarbeitung sind 32 Bit Gleitkommazahlen meistens ausreichend. In MATLAB heißt dies Singel-Precision. Es muss abgewogen werden zwischen der Dynamik, dem Pegel des Grundrauschens und der Rechenleistung. 64-Bit Gleitkommazahlen haben eine höhere Pegeldynamik, ein niedrigeres Grundrauschen und verbrauchen mehr Rechenleistung als 32 Bit Gleitkommazahlen (Analog Devices N/A; Patrick Warrington 2012; StackExchange 2012). Die angegebenen Quellen (ebd.) sind online Artikel, welche die Anwendung verschiedener Zahlenformate diskutieren. Es scheint keine einzig richtige Lösung geben. Man muss für das Programm spezifisch verschiedene Zahlenformate ausprobieren und entscheiden welches Ergebnis ein guter Kompromiss zwischen Leistungsverbrauch und klanglicher Qualität ist. Für die gesamte Entwicklung des Verzerrungsplugins wurde mit Double-Precision gearbeitet, da die Variablen automatisch in diesem Dateityp abgespeichert werden. Um die Variablen als Single-Precision Array oder Double-Precision Array abzuspeichern kann folgende Syntax bei der Deklarierung der Variablen benutzt werden.

y=single(x) ; y=double(x)

Interface und Parameter

Das Userinterface von MATLAN-Plugins ist sehr minimalistisch und kann kaum modifiziert werden. Es gibt verschiedene Regler und Knöpfe, aus denen man wählen kann. Diese können alle einfach und auf gleiche Weise implementiert werden. Es wird nun der

Code in *Skript 2* in Bezug auf das Interface und die Parameter erklärt. Das Interface und dessen Regler werden auch durch ein *properties* deklariert. Es wird eine Variable mit dem Namen *PluginInterface* erschaffen. Danach wird die Funktion, *audioPluginInterface(),* gerufen. In den Klammern können nun eine gewünschte Anzahl von Parametern festgelegt werden. Der Befehl *audioPluginParameter()* erschafft einen Fader im Userinterface. Diesem Fader wird die Variable *Gain* zugewiesen. Die Variable *Gain* muss davor auch durch *properties* deklariert werden und mit einem numerischen Startwert versehen werden. In der nächsten Zeile wird der Name des Faders zugewiesen. Dieser Name wird im Userinterface angezeigt. Der letzte und wichtigste Schritt ist, dass dem Fader Minimal- und Maximal-Werte zugewiesen werden. Im konkreten Beispiel ist *Gain = 0*, wenn der Fader ganz nach links geschoben ist. Wenn dieser ganz nach rechts geschoben wird, so ist *Gain = 3*. Dieser Fader weist ein lineares Verhalten auf, wenn der Fader in die Mitte geschoben wird, ist *Gain = 1,5.* Die Spezifikation *lin* kann durch *log* ausgetauscht werden, um ein logarithmisches Fader-Verhalten zu erhalten. Dies kann für Lautstärkepegel sinnvoll sein. Die Variable *Gain* ist, im Beispiel, ein Multiplikator, welcher auf die Amplitude des Eingangssignales Einfluss nimmt. Wichtig für die Implementierung ist, dass Variablen in *methods* immer im Bezug mit einem Objekt verwendet werden müssen. Dasselbe gilt für die Funktion. `function out = process(plugin, in)`. Die Variable *in* stammt von Objekt *plugin.* Die korrekte Implementierung der Variable ist „`plugin.Gain`“. Am Ende der Zeile muss ein Semikolon gesetzt werden, da dieses die Ausgabe der Variablenwerte in der Befehlszeile unterdrückt. Das nicht vorhanden sein eines Semikolons erzeugt einen Error und wird das kompilieren des Programmes verhindern. Dies unterscheidet sich zur prozeduralen Programmierung.

Erzeugen eines Audio-Plugins

Mit dem Befehl `audioTestBench myEchoPlugin` kann die bereits erwähnte Testumgebung, für genanntes Plugin, gestartet werden. Es kann so überprüft werden, ob das Plugin so funktioniert wie es angedacht war. Mit dem Befehl `validateAudioPlugin myPlugin` wird das Plugin validiert. Das bedeutet, dass es mit verschiedenen Testsignalen getestet wird, um grundliegende Fehler zu finden. Dieser Validierungsprozess stellt sicher, dass das Plugin nicht abstürzt, wenn es in der DAW geladen wird oder andere gravierende Fehler passieren. Mit dem Befehl `generateAudioPlugin myEchoPlugin` wird der MATLAB-Code in C++-Code übersetzt und anschließend im gleichen Verzeichnis wie der Quellcode als .dll Datei exportiert. Diese Datei kann dann in jeder DAW verwendet werden. Bei zufriedenstellender Funktionsweise ist das Plugin jetzt fertig und einsatzbereit. Der Beispiel-Code erschafft ein Plugin, dass einen Regler hat, der die Ausgangsamplitude einstellen kann. Das Resultierende Ausgangssignal, kann dadurch von einer Amplitude von 0 bis zum 3-fachen der Eingangsamplitude skaliert werden.

3.2 Grundkonzepte der Audioprogrammierung

Um komplexe Algorithmen zu erstellen und Fehler im Code zu finden sind grundlegende Kenntnisse über das Audiosignal nötig.

3.2.1 Umgang mit variablen Eingangsgrößen

In MATLAB müssen wir keine Abtastfrequenzen festlegen, da es automatisch die Abtastfrequenz der DAW erkennt. Solange kein Algorithmus geschrieben wird, welcher den numerischen Wert der Abtastfrequenz benötigt, können wir diese vorerst ignorieren. Das Eingangssignal wird in den Code des Plugins eingespeist. Dies erfolgt in Form eines Arrays. Dies ist im konkreten Fall eine Matrix mit zwei Spalten und einer variablen Anzahl an Zeilen. Die Spalten sind der linke und rechte Kanal des Stereo-Audiosignals. Die Zeilen sind die Samples des Audiosignals. Bei z.B. einer Abtastfrequenz von 44100 Hz würde dies in 44100 numerischen Werten in 44100 Zeilen pro Sekunde, für sowohl den linken als auch den rechten Stereo-Kanal, resultieren. Einzelne numerische Abtastwerte werden als Samples bezeichnet. Ein Sample, darf am Eingang und am Ausgang des Plugins, maximal 1 sein und minimal -1 sein. Da Audiosignale unterschiedliche längen haben und bei einer Echtzeitanwendung nicht bekannt ist wann diese endet, muss der Code mit einer variablen Anzahl an Zeilen funktionieren. Eine einfache Lösung für das Problem ist es Funktionen und Operationen zu benutzen, welche die elementweiseweise Verarbeitung benutzen. Anstatt einer Normalen Multiplikation „ ** “* sollte die elementweise Multiplikation verwendet werden „ *.* “*. Dadurch können die meisten einfachen Berechnungen realisiert werden. Viele Funktionen und Operationen wie, zum Beispiel, die Und-Verknüpfung „ *&&* “ benötigen skalare Eingangsgrößen. Wenn Logik benötigt wird, so muss die elementweise Verarbeitung implementiert werden.

3.2.2 Elementweise Signalverarbeitung

Die elementweise Signalverarbeitung wird an einem konkreten Beispiel erklärt. Der Verzerrungsalgorithmus: *„Röhrenemulation (Doidic et al. 1998)“* aus Kapitel 2 wird mit folgendem Code in ein funktionsfähiges Plugin ohne Regler verwandelt. Die elementweise Signalverarbeitung wird benötigt da mehrere Und-Verknüpfungen *„ && “* verwendet werden. Da die Und-Verknüpfung Skalare benötigt ist es nicht möglich gleichzeitig beide Spalten des Audio-Signales zu verarbeiten.

```
classdef Elemtwiseprocessing < audioPlugin

    methods

        function out = process(plugin,in)

           N = length(in)    ;

           out = zeros(N,2) ;

           for n = 1:N

               if  -1 <= in(n,1) && in(n,1) < -0.08905

                   out(n,1) = (-3/4) .* (1-(1-(abs(in(n,1))-0.032847)).^(12)...
                   + (1/3).*(abs(in(n,1))-0.032847)) + 0.01;

               elseif   -0.08905 <= in(n,1) &&  in(n,1)  < 0.320018

                   out(n,1) = -6.153 * (in(n,1)).^2 +  3.9375 * in(n,1);

               elseif   0.320018 <= in(n,1) &&  in(n,1) <= 1

                   out(n,1) = 0.630035;

               end

               if  -1 <= in(n,2) && in(n,2) < -0.08905

                   out(n,2) = (-3/4) .* (1-(1-(abs(in(n,2))-0.032847)).^(12) ...
                   + (1/3).*(abs(in(n,2))-0.032847)) + 0.01;

               elseif   -0.08905 <= in(n,2) &&  in(n,2)  < 0.320018

                   out(n,2) = -6.153 * (in(n,2)).^2 +  3.9375 * in(n,2);

               elseif   0.320018 <= in(n,2) &&  in(n,2) <= 1

                   out(n,2) = 0.630035;
               end
           end
        end
    end
end
```

Skript 3 elementweise Signalverarbeitung

Um eine korrekte Verarbeitung für ein Stereo-Signal durchzuführen, muss zuerst der Verzerrungs-Algorithmus für den linken Kanal und anschließend für den rechten Kanal angewandt werden. Durch eine For-Schleife wird jedes Sample einzeln in den

Algorithmus gespeist. Das Sample besteht aus einem numerischen Wert für linken und rechten Kanal. Zu jedem Zeitpunkt muss das Input-Array, *in*, die gleichen Dimensionen haben wie das Output-Array, *out.* Es darf niemals ein Sample fehlen oder eins zu viel entstehen. Dies würde MATLAB mit einem Error anzeigen und der Code könnte nicht kompiliert werden. Um diese gleichen Dimensionen zu gewährleisten, wird mit der ersten Anweisung, `N = length(in),` die Zeilenanzahl des Eingangs-Arrays bestimmt. Danach wird mit, `out = zeros(N,2),` das Output-Array erschaffen und mit zwei Spalten und *N* Zeilen Nullen beschrieben. Dadurch sind zu jedem Zeitpunkt und für jedes Sample *N* die Dimensionen des Input-Array denen des Output-Arrays gleich. Danach wird durch eine For-Schleife, `for n = 1:N,` eine Laufvariable erschaffen. Die Laufvariable reicht von Sample 1 bis zu Sample N. Dadurch wird gewährleistet das die Laufvariable immer gleich dem aktuellem Sample N ist und nie größer oder kleiner ist. Mit dieser For-Schleife kann man nun Algorithmen für jedes einzelne Sample durchführen. Anschließend werden im Output-Array die bereits eingefügten Nullen ersetzt durch die von dem Algorithmus berechneten Werte. Dies kann mit folgendem Syntax erzielt werden, `out(n,2) = in(n,2)*Algorithmus(vereinfacht)`. Wichtig ist das die Laufvariable *n* die Zeile auswählt aus welcher ausgelesen wird und die Zeile festlegt, in jene geschrieben wird. Jede beliebige Änderung des Input-Arrays, durch einen Algorithmus wird im Output-Array, für das jeweilige Sample, abgespeichert. Am Ende wird ein `end` benötigt, um die For-Schleife abzuschließen. Mit der elementweisen Verarbeitung erhält man viel Kontrolle über jedes einzelne Sample. Dadurch können Fehler vermieden werden und Wirkprinzipien effektiv eingebaut werden, es ist aber nicht immer notwendig. Es ist immer dann notwendig, wenn Skalare benötigt werden. Es ist nicht zielführend in einer elementweisen Verarbeitung die elementweise Multiplikation, *.*,* zu verwenden, da dies unnötig Rechenleistung verschwendet. Für ein Verzerrungs-Plugin wird kein zeitvariantes Verhalten benötigt und auch kein Speicher, aber dies ist mit der elementweisen Verarbeitung gut implementierbar. Es können Samples auf andere Variablen gespeichert werden und es können somit auch vorhergehende Samples abgerufen werden. Mit einer Anweisung wie dieser, `x(n-1,2),` könnte man auf das vorhergehende Sample der Variable *x* zugreifen. So können Filter und Delays erschaffen werden. Allerdings ist es für eine schnelle Entwicklung sinnvoll zuerst MATLABs weitreichende Bibliothek an Funktionen zu durchsuchen.

Wenn *Skript 3* mit *Skript 2* kombiniert wird, erhält man ein Verzerrungs-Plugin mit einem Regler, welcher den Drive einstellt, also eine Verstärkungsstufe, welche den Grad an Verzerrung einstellen kann. Optimale Einstellungen für diesen Parameter können aus der vergleichenden Darstellung aus Kapitel 2.8 entnommen werden.

3.2.3 Implementierung der Überabtastung

Die Überabtastung ist der einfachste Weg um Aliasing zu minimieren und findet in der Verzerrungstechnik breitflächige Anwendung (Araya und Suyama 1996; Doidic et al. 1998; Yeh 2009). Ein Anti-Aliasing Filter währe in Kombination mit Überabtastung das bestmögliche Mittel, um Aliasing zu minimieren. Allerdings ist dies nach einigen Implementierungsversuchen nicht richtig gelungen. Die Ergebnisse der Überabtastung sind auch ohne Anti-Aliasing-Filter gut. Überabtastung beruht auf dem Prinzip die Abtastfrequenz, um einen Faktor, zu erhöhen, die gewünschte Operation, also die Verzerrung durchzuführen und im Anschluss die Abtastfrequenz, um den gleichen Faktor zu verringern. Warum dies Aliasing minimiert, wurde bereits in Kapitel 2.2.2 ausführlich erklärt. Fakt ist, dass es unterschiedliche Möglichkeiten gibt, um ein solches Verfahren zu realisieren. Die Schwierigkeit liegt darin, das MATLAB viel im Hintergrund arbeitet, um dem Programmierer die Arbeit zu erleichtern. Dadurch sind grundlegende Prozesse wie die Abtastung verborgen. Überabtastung zu implementieren Bedarf deshalb viel Wissen über die internen Prozesse von MATALB. Aus diesem Grund wurde die Hilfe des MATLAB Supports herangezogen. Es wurde ein Skript erarbeitet, dass zufriedenstellende Ergebnisse erzielt. Der Überabtastungs-Faktor ist mit einem Drop-Down-Menü wählbar, während das Plugin im Betrieb ist, ohne dass dabei hörbare Fehler entstehen. Dadurch können die in Kapitel 2.4 ermittelten Überabtastungsfaktoren für den jeweiligen Verzerrungs-Algorithmus gewählt werden. In einem finalen kommerziellen Plugin würde man diese Option möglicherweise verbergen, da unerfahrene Benutzer nicht in die technischen Abläufe eines Plugins eingreifen können sollten. Das auf das Wirkprinzip reduzierte Skript für die Überabtastung ist im Anhang unter 1.2.1 zu finden. Die gefundene Lösung ist nicht die einfachste Möglichkeit, um Überabtastung zu implementieren, aber sie funktioniert dafür fehlerfrei und kann einfach angepasst werden. Es werden drei Objekte aus der DSP System Toolbox benötig. Das sind dsp.FIRInterpolator, dsp.FIRDecimator und dsp.AsyncBuffer. Wichtig zu wissen ist, dass diese Objekte nicht einfach so benutzt werden können. Sie sind Baupläne für Werkzeuge. Zuerst muss mit dem Bauplan das Werkzeug erschaffen und definiert werden. Dies nennt man Konstruktor. Mit folgendem Syntax wird ein Objekt *Up2* der Klasse *plugin* erschaffen. Dieses Objekt ist ein FIRInterpolater mit den definierten Eigenschaften. Im konkreten Fall wird zweifach Überabgetastet. Der FIRInterpolater ist ein komplexes Werkzeug, welches es ermöglicht die Abtastfrequenz zu steigern.

```
plugin.Up2 = dsp.FIRInterpolator(2,designMultirateFIR(2,1));
```

Anschließend wird mit `x = plugin.Up2(in);` das Input-Array Überabgetastet und in die Variable *x* abgespeichert. Die meisten MATLAB Objekte müssen erst mit einem Konstruktor erschaffen werden und können dann wie eine Funktion benutzt werden. Das gleiche gilt für den dsp.FIRDecimator und den dsp.AsyncBuffer. Das Skript zur

Überabtastung kann in mehrere Teile gegliedert werden. Zuerst werden alle Variablen definiert. Danach wird ein Interface erschaffen mit welchen man den Überabtastungs-Faktor wählen kann. Danach wird mithilfe der Konstruktoren das jeweilige benötigte Objekt erschaffen. In konkreten Fall bedeutet das, dass für jeden Überabtastungsfaktor jeweilige FIRinterpolater und FIRDecimater Objekte erschaffen und vorberechnet werden. Das Drop-Down-Menü des Interface steuert einen Switch, welcher den Signalweg adäquat verändert. Dieser Switch wählt sowohl den Überabtastungs-Faktor als auch den Unterabtastungs-Faktor. Dadurch wird gewährleistet das die Abtastrate des Ausgangssignal gleich der Abtastrate des Eingangssignals ist. Die überabgetasteten Signale werden unabhängig vom verwendeten Faktor in die gleiche Variable *x* abgespeichert. Dabei beschreibt aber nur der mit dem Switch gewählte Signalweg die Variable. Das ermöglicht es jetzt Operationen mit der Variable *x* durchzuführen, die überabgetastet werden, weil. *X* ist das Input-Array, welches überabgetastet wurde. Eine solche Operation ist im konkreten Fall der Verzerrungsalgorithmus. Man könnte aber somit auch andere Operationen überabtasten. Der Output der Verzerrung wird in eine Variable z abgespeichert. Diese wird im Anschluss unterabgetastet. Dadurch wird die Verzerrung mit einer höheren Abtastrate berechnet aber der Output des Plugins weist dieselbe Abtastrate wie das die DAW auf. Es gibt nur noch eine Krux und die ist, dass das Objekt *dsp.FIRDecimator* keine unendlich großen Eingangsgrößen akzeptiert, sondern mit einer vordefinierten Anzahl an Samples beschrieben werden muss. Das Input-Array des Plugins variiert in seiner Größe, je nachdem wie lange das Plugin in Betrieb ist und mit Daten gespeist wird. Dieses Problem wird mit einem Buffer gelöst. Ein Buffer ist eine Variable, welche feste Dimensionen hat. Wenn ein Buffer eine Größe von 44100 x 1 hat, so wird nacheinander jedes Sample in einem dieser Speicherplätze abgespeichert. Besonders ist, dass das 44101 erste Sample wieder in Position 1 abgespeichert wird und den vorherigen Wert überschreibt. Dieser Buffer wird erst beschrieben und danach direkt ausgelesen. Dadurch hat der *dsp.FIRDecimator* feste Dimensionen des Eingangssignales die er erwarten kann. Der Buffer wird leider zwangsläufig benötigt, um diese Umsetzung der Überabtastung funktionsfähig zu machen. Man könnte diese Lösung als „Workaround“ bezeichnen, da bei einem perfekten Überabtastungsskript kein Buffer benötigt wird. Aber da diese Lösung fehlerfrei funktioniert ist sie gut genug, um des Plugin zu realisieren. Dieses Skript ist ein guter Startpunkt für andere Plugins, welche Überabtastung benötigen. Überabtastung ist trotz des einfach zu verstehenden Wirkprinzips aufwändig in der technischen Umsetzung.

3.2.4 Implementierung von simplen Filtern

Das Verzerrungsplugin beinhaltet eine Filtersektion. Diese ist ein MATLAB Preset. Der MATLAB Support hat dieses Preset so umgeschrieben, dass es als Teil einer Signalkette funktionieren kann. Das Ergebnis ist öffentlich einsehbar (jibrahim MATLAB Staff) und kann einfach implementiert werden. Das benutzen des MATLAB Presets ist notwendig, da so effektiv Zeit gespart werden kann. Das Preset besteht aus vielen hunderten Zeilen an Code mit komplexen Berechnungen. Ein Grund dafür ist, dass dieser Filter in Echtzeit regelbar ist. Das bedeutet, dass wenn mit dem Interface interagiert wird muss der Filter neu berechnet werden. Dies ist viel komplexer als ein nicht regelbarer Filter. Nicht regelbare Filter werden häufig benötigt, um technische Prozesse zu realisieren. In finalen Verzerrungs-Plugin findet eine Offsetkompensation mittels eines nichtregelbaren Hochpassfilters statt. Die Implementierung eines solchen Filters ist einfach und funktioniert fehlerfrei.

```
classdef Offsetkompensation < audioPlugin
    properties
       AFilter
    end
    methods
        %Constructor
        function plugin = Offsetkompensation
            plugin.AFilter = dsp.HighpassFilter('SampleRate',...
            44100,'Filter-
Type','IIR','DesignForMinimumOrder',true,...
            'PassbandFrequency',20,'StopbandFrequency',1, ...
            'StopbandAttenuation',80);
        end
    end
    methods
         function out = process(plugin,in)
            out = plugin.AFilter((in(:,1:2)));
         end
    end
end
```

Dieser Code erzeugt ein Audio-Plugin, welches durch einen Hochpassfilter eine Offsetkompensation realisiert. Dieser Code kann als Baustein für ein größeres Programm fungieren. Zuerst wird die Variable, *AFilter,* deklariert. Danach wird durch das Objekt, *dsp.HighpassFilter,* in die Variable, *AFilter,* ein Filter, mit den definierten Eigenschaften konstruiert und abgespeichert. Der Filter funktioniert nur für eine Abtastfrequenz von 44100 korrekt. Wenn die Abtastfrequenz der DAW geändert wird, so verschiebt sich die Cutoff-Frequenz des Filters. Bei Bedarf könnte man mehrere Filter für unterschiedliche Abtastfrequenzen vorberechnen und dann durch einen Switch für die gerade verwendete Abtastrate auswählen. Der Filter ist ein IIR Filter, welcher von MATLAB automatisch auf Leistung optimiert wird. Alle Frequenzen unter 1 Hz werden um – 80 dB abgesenkt und alle Frequenzen über 20 kHz bleiben unangetastet. Zuletzt wird das neue Filter Objekt,

welches in der Variable *AFilter* abgespeichert wurde, wie eine Funktion verwendet. Ein Input und ein Output, wird zugewiesen. Dies ist die einfachste Art und Weise, um einen Filter im Objekt Orientierten MATLAB zu implementieren. Die sehr simplen Filterbefehle aus der prozeduralen Programmierung funktionieren nicht für Echtzeitanwendungen mit Eingangsgrößen mit variabler Länge.

3.2.5 Anpassung der Signalstärke

Viele Verzerrungsalgorithmen erhöhen die Amplitude des Eingangssignals. Manche senken diese auch ab. Optimalerweise ist die Amplitude des Ausgangssignals gleich der Amplitude des Eingangssignals. Pegelunterschiede können die Empfindung des Benutzers stark beeinflussen. Oft werden Veränderung an einer musikalischen Tonspur als positiv empfunden, wenn diese Veränderung den Pegel anhebt, da dadurch das veränderte Element in der Gesamtheit der Klänge einer Komposition mehr heraussticht. Um diesen Effekt zu vermeiden ist eine Anpassung der Signalstärke des Ausgangssignals an das Eingangssignal sinnvoll. Tatsächlich bieten viele hochwertige Plugins dieses Feature, aber bei weitem nicht alle. Die Lösung, die gefunden wurde, ist nicht perfekt, da eine Funktion in MATLAB fehlt die nützlich gewesen wäre. Das folgende Konzept funktioniert aber meistens und erfüllt damit seine Aufgabe gut.

```
if  plugin.toggleAdjustOutputGain

                    plugin.InMaximum = max(max(abs(in(:,:))));

                    plugin.OutMaximum = max(max(abs(xout(:,:))));

                    if plugin.setMaximum

                    plugin.setInMaximum = plugin.InMaximum;
                    plugin.setOutMaximum = plugin.OutMaximum;

                    end

                    ratioinout = plugin.setInMaximum/plugin.setOutMa-
ximum;

                    xout(:,:) = xout(:,:)*ratioinout;
end
```

Die Funktionsweise ist simpel. Im Endeffekt wird die Amplitude des Eingangssignals und des Ausgangssignals an einem gemeinsamen Zeitpunkt bestimmt. Die Amplitude des Eingangssignals wird durch die Amplitude des Ausgangssignals geteilt und dann mit dem Ausgangssignal multipliziert. Dadurch erhält das Ausgangssignal die gleiche Amplitude wie das Eingangssignal. Das funktioniert nur gut für Signale deren Amplitude relativ gleichbleibend ist. Das funktioniert auch für transiente Signale wie z.B. ein Schlagzeug, in diesem Fall müssen die Amplitudenwerte aber gleichzeitig mit dem Auftreten

eines Transienten bestimmt werden. Das Verfahren funktioniert nicht für Audiosignale mit hoher Pegeldynamik über große Zeiträume. Aus diesem Grund kann die Funktion deaktiviert werden. Die Variable `plugin.toggleAdjustOutputGain` steht im Zusammenhang mit einer Checkbox aus dem User-Interface. Wenn ein Häkchen gesetzt wird, wird in die Variable der Logikwert *true* abgespeichert und wenn kein Häkchen gesetzt wird, wird *flase* abgespeichert. In Kombination mit einer *if-Schaltung*, können so Teile vom Code aktiviert oder ignoriert werden. Die Variable `plugin.setMaximum` funktioniert nach gleichem Prinzip. Sie überträgt die gemessenen Amplitudenwerte an die simple Berechnungsformel der Pegelanpassung. Somit ist das betätigen dieses Knopfes der Zeitpunkt, an dem die Pegelanpassung geschieht. Leider bietet MATLAB keine Knöpfe im Userinterface an und deswegen ist im fertigen Verzerrungs-Plugin eine Checkbox verwendet wurden. Um also die Funktionsweise eines Knopfes nachzuempfinden, muss der Hacken schnell gesetzt und danach wieder entfernt werden. Das ist s unnötig kompliziert. Dadurch wird das Feature leider sehr benutzerunfreundlich, obwohl es dazu gedacht ist die Bedienung des Plugins zu Optimieren. Da aber das Wirkprinzip funktioniert, ist dies vorerst ausreichend. Wenn man das Plugin in C++ realisieren wollen würde, so würde es dort die Möglichkeit geben einen Knopf zu implementieren. Das Verfahren könnte noch mit einer zeitlichen Mittelung optimiert werden. Der RMS kann dafür hilfreich sein. Allerdings ist auch diese simple Pegelanpassung schon funktionsfähig. Es muss nur ein Zeitpunkt für die Pegelanpassung gewählt werden in dem sowohl der Pegel des Eingangs und des Ausgangssignals hoch ist im Vergleich zum Rest des jeweiligen Signals. Also ein Pegel der repräsentativ für da Signal ist. Manchmal werden deswegen mehrere Versuche der Pegelanapassung benötigt. Für den Benutzer dauert das ganze trotzdem nur 10 Sekunden, aber mit einem Knopf anstelle einer Checkbox würde das noch schneller funktionieren.

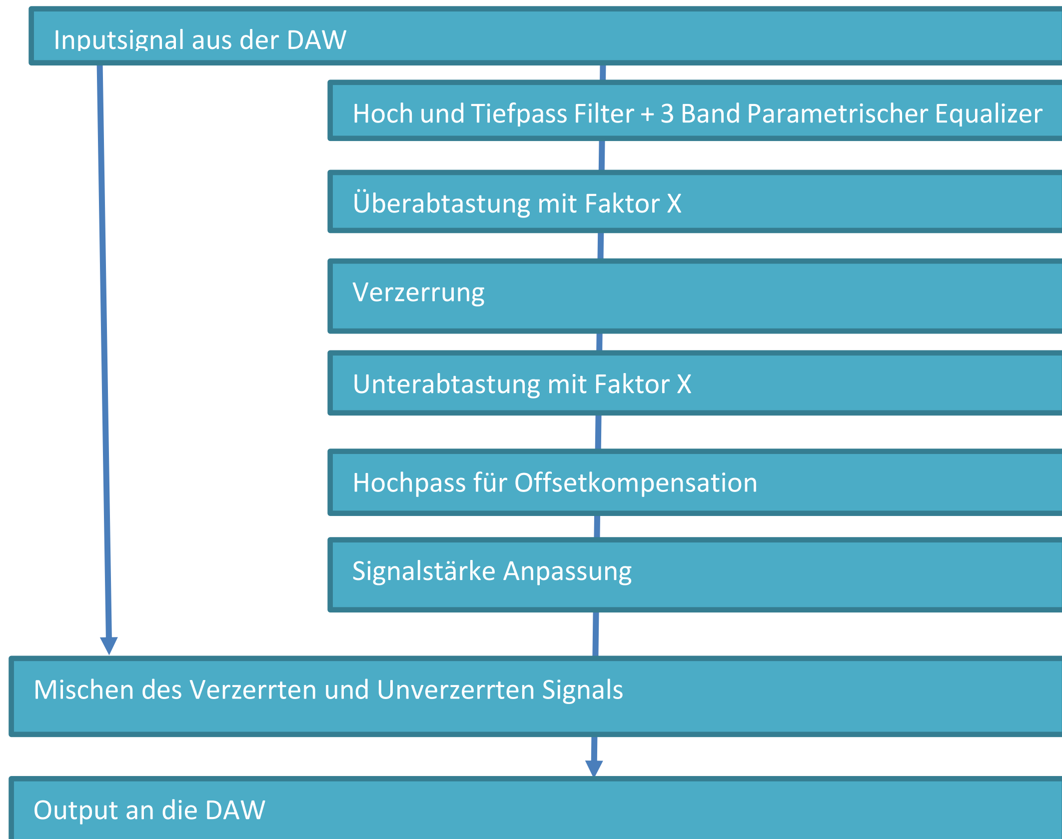
3.2.6 Mischen von zwei Signalen

```
out = ( ( xout .* plugin.DryWet ) + ( in .* ( 1 - plugin.DryWet ) ) )  ;
```

Die Variable `plugin.DryWet` wird aus dem User-Interface von einem Fader mit Werten von 0 bis 1 beschrieben. Die Variable *xout* ist mit dem verzerrten Signal beschrieben. Wenn der Fader nach ganz rechts geschoben wird, resultiert das in einer 1 in der dazugehörigen Variable, dadurch wird nur das verzerrte Signal an die Output-Variable *out* weitergegeben. Bei einem Wert von 0 wird, die Variable *out* nur mit dem unveränderten Eingangssignals beschrieben. In diesem Zustand hat das Plugin keinen hörbaren Effekt. Bei einem Wert von 0.5 wird das Verzerrte und das unverzerrte Signal zu gleichen Teilen gemischt. Diese Methode ermöglicht es zwei Signale miteinander zu mischen. Die Verhältnisse können dabei eingestellt werden. Dadurch kann einem Signal nur wenig Verzerrung hinzugemischt werden. Dies ist in den meisten Fällen sehr nützlich und ein essenzieller Bestandteil des Verzerrungs-Plugins.

3.3 Vorstellung des finalen Verzerrungs-Plugins

3.3.1 Schaltplan



Dieser Schaltplan stellt die Signalwege im Plugin vereinfachend dar. Die Idee des Plugins ist es die klangliche Verfärbung des Eingangssignals durch die Verzerrung kontrollieren zu können. Dafür wurde eine Filtersektion vor der Verzerrung implementiert. Dadurch kann man Frequenzanteile, die nicht oder schwach verzerrt werden sollen, absenken oder komplett entfernen. Über die Mischfunktion kann dann das Ursprungsmaterial wieder hinzugegeben werden. Das Resultat ist, dass nur Frequenzanteile verzerrt werden, bei denen das der Benutzer wünscht. Dadurch wird die Verzerrung zu einem präzisen Werkzeug. Zum Beispiel tiefe Frequenzen unter 300 Hz und hohe Frequenzen über 10 kHz verlieren oft stark an klanglicher Qualität, wenn diese Verzerrt werden. Die sogenannten Mitten, von 700 Hz bis 3 kHz, profitieren dafür oft von einer subtilen Verzerrung. Mit der Filtersektion lassen sich solche Einstellungen realisieren. Die Flankensteilheit der Filter ist von 6 dB pro Oktave bis 48 dB pro Oktave einstellbar. Der 3 Band parametrische Equalizer hat die für einen parametrischen Equalizer typischen Einstellmöglichkeiten. Für jedes Band kann eine Mittenfrequenz eingestellt werden. Es kann +/- 20 dB verstärkt oder abgesenkt werden. Da es sich um eine „Bell“ also um eine Glockenform handelt kann der Q-Faktor, die Güte, eingestellt werden. Der Q-Faktor stellt

die Bandbreite des betroffenen Frequenzbandes ein. Der Equalizer ermöglicht es auch Frequenzanteile zu verstärken. Da Verzerrung im direkten Zusammenhang mit der Eingangsamplitude steht, können so gewählte Frequenzbereiche besonders stark verzerrt werden. Nach der Filtersektion wird das Signal überabgetastet und verzerrt, danach wird durch eine Unterabtastung wieder die ursprüngliche Abtastrate hergestellt. Der Überabtastungsfaktor kann frei gewählt werden. Zur Orientierung dient die Tabelle aus dem Vergleich der Algorithmen in Kapitel 2.4. Der Grad der Verzerrung kann durch einen Multiplikator der Eingangsamplitude eingestellt werden. Danach wird mit einem Hochpass der Gleichanteil entfernt. Diese Offsetkompensation ist ein und ausschaltbar. Sie ist allerdings standartmäßig eingeschaltet, da ein Gleichanteil nicht hörbar ist. Danach erfolgt eine Anpassung der Signalstärke, sodass das Ausgangssignal die gleiche Amplitude aufweist wie das Eingangssignal. Im Anschluss kann das unverzerrte Signal mit dem verzerrten Signal gemischt werden. Das funktioniert besonders gut, nachdem die Pegel angepasst wurden.

3.3.2 Das finale User-Interface

MATLAB bietet vorgefertigte Bauteile für das Erstellen eines User-Interfaces an. Nur diese können benutzt werden. Das spart Zeit, sieht aber nicht gut aus. Zusätzlich fehlen essenzielle Bauteile wie im konkreten Fall der dringen benötigte Knopf. Im Großen und Ganzen konnte aber alles mit dem von MATLAB vorgegebenen Bauteilen realisiert werden. Das Hauptproblem ist aber, dass MATLAB keine Möglichkeit gibt irgendeine Art von Visualisierung einzubauen. Die große Stärke von MATLAB ist seine Darstellung durch die Plot-Funktion. Dies ist in einem Audio-Plugin leider nicht möglich. Das allermindeste für ein Verzerrungs-Plugin wäre eine Lampe die Rot aufleuchtet, wenn das Signal übersteuert. Die meisten simplen Verzerrungs-Plugins bieten ein *VU-Meter.* Mit dessen Hilfe kann dann die Amplitude des Eingangs- und des Ausgangssignals abgelesen werden. Manche gute Verzerrungs-Plugins stellen das Signal über Zeit dar und den Einfluss der Verzerrung auf das Signal, sowie die Kennlinie der Verzerrung. Diese Darstellungen sind dafür da, um dem Benutzer verstehen zu lassen, welchen Einfluss sein Handeln hat und wie er die Pegel korrekt einzustellen hat. Beim MATLAB Plugin muss auf die gesamte Kommunikation zwischen Programm und Benutzer verzichtet werden. Dadurch fühlt sich das Plugin nicht intuitiv and und weniger wertig. Ein Plugin muss intuitiv verstanden werden können. Es muss bei der ersten Benutzung ein gutes Resultat erzielt werden, damit ein Benutzer das Plugin nochmal freiwillig öffnet. Das User-Interface ist mindestens genauso wichtig wie die digitale Signalverarbeitung. Allerdings kann mit dem simplen User-Interface die gesamte Funktionalität des Plugins erkundet werden. Für ein *Proof of Concept* ist das ausreichend. Während die digitale Signalverarbeitung gut in MATLAB funktioniert, so hört die Plugin-Entwicklung spätestens beim User-Interface auf.

3.3.3 Der Klang

Diese Einschätzung ist subjektiv, aber auch objektiv kann gesagt werden, dass das Plugin keine Störgeräusche erzeugt, ein Grundrauschen von -300 dB aufweist und das durch die Überabtastung der Aliasing-Effekt stark vermindert wird. Das Plugin wurde also technisch gesehen gut umgesetzt, sodass der Klang nur vom Verzerrungs-Algorithmus erzeugt wird. Ob man den Klang des Verzerrungs-Algorithmus als wohlklingend empfindet ist subjektiv. Meine Erwartungen wurden übertroffen. Das Plugin klingt gut und ist benutzbar. Nicht jeder Algorithmus gefällt mir persönlich. Einige Verzerrungen aus kommerziellen Plugins sind den Verzerrungen aus dem selbstgeschriebenen Plugin sehr ähnlich oder möglicherweise sogar identisch. Es ist sehr wahrscheinlich das andere Entwickler während ihren Recherchen auf die gleichen Quellen gestoßen sind, wie jene die in dieser Arbeit verwendet wurden. Zusätzlich sind manche Wirkprinzipien wie z.B. die Gleichrichtung klar definiert und bieten nicht viel Varianz in der technischen Umsetzung an. Die verwendeten Verzerrungs-Algorithmen sind klanglich hochwertig und stehen professionellen Plugins in wenig nach. Der gute Klang steht auch in direktem Zusammenhang mit der Überabtastung. In einer experimentellen Vorläuferversion

existierte noch keine Überabtastung. Das Resultat war, das das Aliasing deutlich hörbar war und dadurch verschiedene Algorithmen sehr ähnlich klangen. Mit Überabtastung existiert dieses Problem nicht und die meisten Algorithmen sind voneinander differenzierbar, indem sie klare klangliche Unterschiede erzeugen. Schlussendlich muss gesagt werden, dass ein besseres Interface, das mehr mit dem Benutzer interagiert, auch für einen besseren Klang sorgen kann. Erst wenn der Algorithmus präzise angesteuert werden kann, kann mit Erfahrung die optimale Klangerfahrung erzeugt werden. Verzerrung ist stark abhängig von der Eingangsamplitude. Schon ein dB-Meter, welches die Eingangsamplitude in dBFS anzeigt, würde die Ansteuerung der Verzerrung stark verbessern und damit auch die Ergebnisse, die damit erzielt werden können

3.3.4 Leistungsoptimierung

Ziel war es ein Plugin zu erschaffen, dass funktioniert. Also eine Signalverarbeitung die fehlerfrei ist. Das ist gelungen. Die besten Mittel zu wählen, um einen Signalverarbeitungsprozess schnell und effizient zu gestalten bedarf viel Erfahrung, die während der Durchführung dieses Projektes noch nicht vorhanden war. Es gibt aber einige Methoden wie die Leistungseffizienz von diesem Plugin stak gesteigert werden kann. Der einfachste weg ist das den verwendeten Zahlentyp zu ändern. Das Plugin benutzt 64 Bit Gleitkommazahlen. Man könnte aber auch 32 Bit Gleitkommazahlen benutzen. Dadurch würde man in etwa die Hälfte an Rechenleistung einsparen. Fraglich ist, ob man die beiden Zahlenformate klanglich unterscheiden könnte. Eine weitere Möglichkeit, wäre das Verwenden von Lookup-Tabellen. Mit dieser Technik können aufwändige Berechnungen umgangen werden und sie bietet sich für die verwendeten Verzerrungs-Algorithmen an. Man könnte eine Tabelle erstellen, die für die Kennlinien der Verzerrung, alle Eingangswerte und die dazugehörigen Ausgangswerte, in einer bestimmten Auflösung, abspeichert. Diese Tabelle kann dann anstatt des Algorithmus verwendet werden. Für die Eingangswerte werden dann die dazugehörigen Ausgangswerte einfach aus der Tabelle ausgelesen. Diese Technik ist in der Informatik weit verbreitet und würde für die Verzerrungsalgorithmen auch gut funktionieren. Lookup-Tabellen sind auch in MATLAB möglich. Die nächste Idee für eine mögliche Leistungsoptimierung ist in MATLAB nicht möglich. Im JUCE- Framework für C++ gibt es eine fast-Math Funktionssammlung. Es handelt sich um klassische mathematische Funktionen, wie z.B. den Tangens oder dem Sinus, welche in Lookup-Tabellen abgespeichert wurden. Dabei ist die Auflösung so gewählt wurden das die Quantisierung für das menschliche Ohr nicht hörbar ist, aber so stark ist das viel Rechenleistung gespart werden kann. Die hohe mathematische Präzision von MATLAB-Funktionen ist für ein Plugin nicht zwangsläufig notwendig und nicht in jedem Fall sinnvoll. Mit viel Erfahrung lassen sich das simple Verzerrungs-Plugin noch weiter Vereinfachen. Je simpler der Code ist umso besser ist dieser. Schlussendlich bedarf das Plugin mehr Rechenleistung als seine professionellen Gegenstücke. Dabei ist

der Funktionsumfang der professionellen Plugins deutlich höher als beim selbstgeschriebenen Plugin. An dieser Stelle ist also noch Raum nach oben. Allerdings ist die benötigte Rechenleistung des Plugins nicht so hoch, dass es dadurch unbenutzbar wird. Man kann auf einem guten Computer weit über 10 Instanzen von diesem Plugin benutzen, ohne die Computer voll auszulasten. Die benötigte Rechenleistung ist stark abhängig vom verwendeten Überabtastungsfaktor. Es muss zwischen klanglicher Qualität und Rechenleistung abgewogen werden.

3.3.5 Verbesserungsansätze

3.3.6 Wie könnte es weiter gehen?

Um das Plugin auf ein professionelles Niveau anzuheben, ist definitiv eine Übertragung des Programms in ein C++ Framework notwendig. Für Audioplugins wird das JUCE-Framework verwendet, da es sowohl für Signalverarbeitung als auch für die Erstellung von Apps und Interfaces optimiert ist. Die einfachste Möglichkeit, um das MATLAB-Plugin in JUCE zu erhalten, ist in MATLAB das Plugin zu kompilieren. Es gibt einen Befehl bei dem das Projekt nicht zu einer dll. Oder vst. kompiliert wird, sondern zu einer JUCE-Datei. Das ermöglicht es ein qualitativ hochwertiges User-Interface für das Plugin zu entwickeln.

4 Fazit

Es wurden verschiedene Verzerrungs-Algorithmen kompiliert und analysiert. Durch die Analyse des Aliasing-Effekts der jeweiligen Algorithmen, konnten diese qualitativ hochwertig in das Plugin implementiert werden. Dem Plugin wurden verschiedene Features hinzugefügt, um seine Funktionalität zu erhöhen. Diese sind die Überabtastung, ein Equalizer, eine Pegelanpassung und eine Entfernung des Gleichanteils. Das Plugin besitzt das rustikale MATLAB-Interface und kann dadurch nicht mit professionellen Plugins konkurrieren. Das Plugin ist nicht intuitiv in seiner Benutzung. Das Plugin klingt gut und hat keine offensichtlichen Störgeräusche. Die technische Umsetzung ist also gut gelungen. Der Code könnte effizienter seien. Es gibt möglicherweise bessere Wirkprinzipien und technische Lösungen als die die Verwendet wurden. Die Bestandteile des Plugins wurden anhand von Code erklärt. Der Code wurde bereitgestellt und kann dem Leser nützlich sein. Der komplette Code für das Plugin ist im Anhang zu finden und kann zum finalen Plugin kompiliert werden. Das kompilierte Plugin ist voll funktionsfähig und kann benutzt werden. Zusammenfassend kann gesagt werden das ein guter Prototyp erstellt wurde und das die Ziele der Arbeit erfüllt wurden.

Literaturverzeichnis

Literaturverzeichnis

Analog Devices (N/A): Fixed-Point vs. Floating-Point Digital Signal Processing. Technology Overview and Application Considerations. Online verfügbar unter https://www.analog.com/en/technical-articles/fixed-point-vs-floating-point-dsp.html, zuletzt geprüft am 04.05.2021.

Araya, Toshinori; Suyama, Akio (1996): Sound effector capable of imparting plural sound effects like distortion and other effects: Google Patents.

Bendiksen, Ragnar (1997): Digitale lydeffekter. In: *Institutt for teleteknikkAkustikk, Norges Teknisk-Naturvitenskapelige Universitet, Diplomoppgave*.

Bonanno, Josh (2019): 3 Subtle Ways to Use Distortion in Your Mixes. Online verfügbar unter https://www.waves.com/subtle-ways-to-use-distortion-in-your-mixes, zuletzt geprüft am 03.03.2021.

Bulling, Philipp (2013): Echtzeit-Implementierung einer Röhrenvorstufe zur Verzerrung diskreter Audiosignale. Hochschule Ulm, Bibliothek.

Chowdhury, Jatin (2020): A Comparison of Virtual Analog Modelling Techniques for Desktop and Embedded Implementations. In: *arXiv preprint arXiv:2009.02833 [Titel anhand dieser ArXiv-ID in Citavi-Projekt übernehmen]*.

Ciciora, Walter S.; Ciciora, Walter; Large, David; Adams, Michael; Farmer, James (2004): Modern cable television technology: Morgan Kaufmann.

Dixon, Daniel (2018): What Is Distortion in Music? When and How to Use It. iZotope. Online verfügbar unter https://www.izotope.com/en/learn/what-is-distortion-in-music-when-and-how-to-use-it.html, zuletzt geprüft am 03.03.2021.

Doidic, Michel; Mecca, Michael; Ryle, Marcus; Senffner, Curtis; others (1998): Tube modeling programmable digital guitar amplification system: Google Patents.

Eichas, Felix; Zölzer, Udo (2018): Virtual Analog Modeling of Guitar Amplifiers with Wiener-Hammerstein Models. In:

Elflein, Dietmar (2014): Schwermetallanalysen: die musikalische sprache des heavy metal: transcript Verlag (6).

Enders, Bernd (1985): Lexikon Musikelektronik. 3.Auflage 1997: Schott Music International.

Fifka, Matthias S. (2019): Rockmusik in den 50er und 60er Jahren: von der jugendlichen Rebellion zum Protest einer Generation: Nomos Verlag.

Guicking, Dieter (2016): Schwingungen: Springer.

Güte, Stephan (2018): Die Kemper AMP Story, Gitarrenrevolution aus Deutschland. Online verfügbar unter https://www.amazona.de/die-kemper-amp-story-gitarrenrevolution-aus-deutschland/, zuletzt geprüft am 07.03.2021.

Herbst, Jan-Peter (2017): Virtuoses Gitarrenspiel im Rock und Metal: Zum Einfluss von Verzerrung auf das ‚Shredding'[Virtuoso solo guitar in rock and metal music: About the influence of distortion on'shredding']. In: Transcript.

itectec (N/A): MATLAB: How to set up Microsoft Visual Studio 2017 for SLRT. Online verfügbar unter https://itectec.com/matlab/matlab-how-to-set-up-microsoft-visual-studio-2017-for-slrt/, zuletzt geprüft am 03.05.2021.

jibrahim MATLAB STAFF: Distortion Plugin with variable Oversampling. Online verfügbar unter https://de.mathworks.com/matlabcentral/answers/734210-distortion-plugin-with-variable-oversampling?s_tid=prof_contriblnk, zuletzt geprüft am 08.06.2021.

jibrahim MATLAB Staff: How to use Audio Plugin Example System Objects in a custom Plugin. Online verfügbar unter https://de.mathworks.com/matlabcentral/answers/733873-how-to-use-audio-plugin-example-system-objects-in-a-custom-plugin?s_tid=srchtitle, zuletzt geprüft am 09.06.2021.

Kahrs, Mark; Brandenburg, Karlheinz (1998): Applications of digital signal processing to audio and acoustics: Springer Science & Business Media.

Karplus, Kevin; Strong, Alex (1983): Digital synthesis of plucked-string and drum timbres. In: *Computer Music Journal* 7 (2), S. 43–55.

Lindner, Helmut; Brauer, Harry; Lehmann, Constans (2008): Taschenbuch der Elektrotechnik und Elektronik: Carl Hanser Verlag GmbH Co KG.

Maré, Stefan (2002): Detection of nonlinear distortion in audio signals. In: *IEEE transactions on broadcasting* 48 (2), S. 76–80.

MathWorks (N/A): Matrizen und Arrays. Online verfügbar unter https://de.mathworks.com/help/matlab/learn_matlab/matrices-and-arrays.html, zuletzt geprüft am 04.05.2021.

MATLAB (2021): 9.8.0.1451342 (R2020a) Update 5. Natick, Massachusetts: The MathWorks Inc.

Otala, M. (1977): Non-linear distortion in audio amplifiers. In: *Wireless World*.

Pakarinen, Jyri; Yeh, David T. (2009): A review of digital techniques for modeling vacuum-tube guitar amplifiers. In: *Computer Music Journal* 33 (2), S. 85–100.

Patrick Warrington (2012): Fixed-point vs. floating-point numbers in audio processing. Online verfügbar unter https://www.tvtechnology.com/opinions/fixedpoint-vs-floating-point-numbers-in-audio-processing, zuletzt geprüft am 04.05.2021.

Roads, Curtis (1979): A tutorial on non-linear distortion or waveshaping synthesis. In: *Computer Music Journal*, S. 29–34.

Ruschkowski, Arne von (2008): Loudness war: na.

Savage, Steve (2014): Mixing and mastering in the box: The guide to making great mixes and final masters on your computer: Oxford University Press.

Schultz, Frank; Cholakov, Vladimir; Maempel, Hans-Joachim (Hg.) (2008): Zur Hörbarkeit von digitalen Clipping-Verzerrungen. TONMEISTERTAGUNG – VDT INTERNATIONAL CONVENTION, 2008. Technische Universität Berlin, Fachgebiet Audiokommunikation. Online verfügbar unter https://www2.ak.tu-berlin.de/~akgroup/ak_pub/2008/Schultz%202008%20Zur%20Hoerbarkeit%20von%20digitalen%20Clippingverzerrungen%20TMT.pdf, zuletzt geprüft am 18.03.2021.

Shockley, William (1949): The Theory of p-n Junctions in Semiconductors and p-n Junction Transistors. In: *Bell System Technical Journal* 28 (3), S. 435–489.

Smith III, Julius (2008): Physical Audio Signal Processing: Digital Waveguide Modeling of Musical Instruments and Audio Effects. Online verfügbar unter https://ccrma.stanford.edu/~jos/pasp/, zuletzt geprüft am 14.03.2021.

StackExchange (2012): When to consider double (64 bit) floating point for Audio. Online verfügbar unter https://dsp.stackexchange.com/questions/6079/when-to-consider-double-64-bit-floating-point-for-audio, zuletzt geprüft am 04.05.2021.

Sullivan, Charles R. (1990): Extending the Karplus-Strong algorithm to synthesize electric guitar timbres with distortion and feedback. In: *Computer Music Journal* 14 (3), S. 26–37.

Sunnerberg, Timothy Douglas (2019): Analog musical distortion circuits for electric guitars.

Tarr, Eric (2018): Hack Audio: An Introduction to Computer Programming and Digital Signal Processing in MATLAB: Routledge.

Temme, Steve (1992): Audio distortion measurements. In: *Application Note, Bruel & Kjar*.

Toulson, Rob; Paterson, Justin; Campbell, William (2014): Evaluating harmonic and intermodulation distortion of mixed signals processed with dynamic range compression. In: Future Technology Press.

Volans, Mo (2015): Music Production Techniques Part 3: Distortion. Online verfügbar unter https://ask.audio/articles/music-production-techniques-part-3-distortion, zuletzt geprüft am 07.03.2021.

Yeh, David Te-Mao (2009): Digital Implementation of Musical Distortion Circuits by Analysis and Simulation.

Zölzer, Udo; Amatriain, Xavier; Arfib, Daniel; Bonada, Jordi; Poli, Giovanni de; Dutilleux, Pierre et al. (2002): DAFX-Digital audio effects: John Wiley & Sons.

Anlagen

1. verwendeter Code aus Quellen

1.1. Verzerrungs-Algorithmen

1.1.1. Hard-Clip-Algorithmus (Tarr 2018)

```
% HARDCLIP
% This function implements hard-clipping
% distortion. Amplitude values of the input signal
% that are greater than a threshold are clipped.
%
% Input variables
% in : signal to be processed
% thresh : maximum amplitude where clipping occurs
%
% See also INFINITECLIP, PIECEWISE, DISTORTIONEXAMPLE
function [out] = hardClip(in,thresh)
N = length(in);
out = zeros(N,1);
for n = 1:N
if in(n,1) >= thresh
% If true, assign output = thresh
out(n,1) = thresh;
elseif in(n,1) <= -thresh
% If true, set output = -thresh
out(n,1) = -thresh;
else
out(n,1) = in(n,1);
end
end
```

Der Code für diese Verzerrung ist von Tarr (2018, S. 158).

1.1.2. Limitierung von positiven Halbwellen

Dieser Code ist eine Abänderung vom Hard-Clip-Algorithmus (Tarr 2018).

```
thresh=0.5

N = length(x);

y=zeros(N,1)

for n=1:N

      if x(n,1) >= thresh

          y(n,1) = thresh;

     else
         y(n,1) = x(n,1);
     end

end
```

1.1.3. Röhrenemulation (Araya und Suyama 1996)

```
N = length(x);

y=zeros(N,1)

for n=1:N

a = ( ( 3 .* x(n,1) ./ 2 ) .* (1- ( x(n,1) .^2 ) ./ 3 )) ;


b = ( ( 3 .* a ./ 2 ) .* (1- ( a .^2 ) ./ 3 )) ;

y(n,1) = ( ( 3 .* b ./ 2 ) .* (1- ( b .^2 ) ./ 3 )) ;

end
```

Die Formel dieser Verzerrung stammt von Araya und Suyuma (1996) und wurde in einem Paper gefunden (Pakarinen und Yeh 2009).

1.1.4. Röhrenemulation (Doidic et al. 1998)

```
N = length(x);

y=zeros(N,1)


for n=1:N

  if  -1 <= x(n,1) && x(n,1) < -0.08905

  y(n,1)  =  (-3/4)  .*  (1-(1-(abs(x(n,1))-0.032847)).^(12)  +
(1/3).*(abs(x(n,1))-0.032847)) + 0.01;

  elseif  -0.08905 <= x(n,1) &&  x(n,1)  < 0.320018

  y(n,1) = -6.153 * (x(n,1)).^2 +  3.9375 * x(n,1);

  elseif  0.320018 <= x(n,1) &&  x(n,1) <= 1

  y(n,1) = 0.630035;

  end

end
```

Die Formel 6 von Doidic et al. (1998) wurde in MATLAB-Code übersetzt.

1.1.5. Röhrenemulation (Zölzer et al. 2002)

```
function y=tube(x, gain, Q, dist, rh, rl, mix)
% function y=tube(x, gain, Q, dist, rh, rl, mix)
% Author: Bendiksen, Dutilleux, Zölzer

% y=tube(x, gain, Q, dist, rh, rl, mix)
% "Tube distortion" simulation, asymmetrical function
% x - input
% gain - the amount of distortion, >0->
% Q - work point. Controls the linearity of the transfer
% function for low input levels, more negative=more linear
% dist - controls the distortion's character, a higher number gives
% a harder distortion, >0
% rh - abs(rh)<1, but close to 1. Placement of poles in the HP
% filter which removes the DC component
% rl - 0<rl<1. The pole placement in the LP filter used to
% simulate capacitances in a tube amplifier
% mix - mix of original and distorted sound, 1=only distorted

q=x*gain/max(abs(x));                                   %Normalization
if                                                                Q==0
z=q./(1-exp(-dist*q));
for i=1:length(q)                                   %Test because of the
if q(i)==Q                                          %transfer function's
z(i)=1/dist;                                             %0/0 value in Q
end;
end;
else
z=(q-Q)./(1-exp(-dist*(q-Q)))+Q/(1-exp(dist*Q));
for i=1:length(q)                                   %Test because of the
if q(i)==Q                                          %transfer function's
z(i)=1/dist+Q/(1-exp(dist*Q));                           %0/0 value in Q
end;
end;
end;
y=mix*z*max(abs(x))/max(abs(z))+(1-mix)*x;
y=y*max(abs(x))/max(abs(y));
y=filter([1 -2 1],[1 -2*rh rh^2],y);                                 %HP
filtery=filter([1-rl],[1 -rl],y); %LP filter
```

Die Formel für diese Verzerrung stammt von Bendiksen (1997). Aus Zölzer et al. (2002) Stammt der Matlab Code für die Anwendung der Verzerrung.

„

1.1.6. Diode (Shockley 1949; Tarr 2018)

```
% DIODE
% This function implements the Shockley
% ideal diode equation for audio signals
% with an amplitude between -1 to 1 FS
%
% See also ASYMMETRICAL, DISTORTIONEXAMPLE
function [out] = diode(in)
% Diode characteristics
Vt = 0.0253; % thermal voltage
eta = 1.68; % emission coefficient
Is = .105; % saturation current
N = length(in);
out = zeros(N,1);
for n = 1:N
out(n,1) = Is * (exp(0.1*in(n,1)/(eta*Vt)) - 1);
end
```

Die ursprüngliche Formel stammt von (Shockley 1949) . Tarr (2018, S. 170–171)zeigt eine Version der Formel, die für eine Audioanwendung sinnvoll ist.

1.1.7. Gleichrichter

Der Code für die beiden Algorithmen ist von (Tarr 2018, S. 154–157).

```
% HALFWAVERECTIFICATION
% This function implements full-wave rectification
% distortion. Amplitude values of the input signal
% that are negative are changed to zero in the
% output signal.
% See also FULLWAVERECTIFICATION, DISTORTIONEXAMPLE
function [out] = halfwaveRectification(in)
N = length(in);
out = zeros(N,1);
for n = 1:N
if in(n,1) >= 0
% If positive, assign input to output
out(n,1) = in(n,1);
else
% If negative, set output to zero
out(n,1) = 0;
end
end
```

```
% FULLWAVERECTIFICATION
% This function implements full-wave rectification
% distortion. Amplitude values of the input signal
% that are negative are changed to positive in the
% output signal.
%
% See also HALFWAVERECTIFICATION, DISTORTIONEXAMPLE
function [ out ] = fullwaveRectification(in)
N = length(in);
out = zeros(N,1);
for n = 1:N
if in(n,1) >= 0
% If positive, assign input to output
out(n,1) = in(n,1);
else
% If negative, flip
out(n,1) = -1*in(n,1);
end
end
```

1.1.8. Kubisches Softclipping (Smith III 2008; Sullivan 1990)

Die Formel stammt von (Smith III 2008; Sullivan 1990) und wurde in der Doktorarbeit von Yeh (2009) erwähnt. Eine Implementierung in MATLAB kann mit folgendem Code erzielt werden.

```
N = length(x) ;

y=zeros(N,1) ;

 for n=1:N

if x(n,1) <= -1

    y(n,1) = -2/3;

elseif -1 <= x(n,1) && x(n,1) <= 1

    y(n,1) = x(n,1) - (((x(n,1))^3)/3 );

elseif x(n,1) >= 1

    y(n,1) = 2/3;
end
end
```

Die Modifizierung dieses Algorithmus funktioniert mit folgendem Code.

```
N = length(x) ;

y=zeros(N,1) ;

 for n=1:N

if 0 <= x(n,1) && x(n,1) <= 1

    y(n,1) = x(n,1) - (((x(n,1))^a)/3 );

elseif x(n,1) < 0

    y(n,1) = x(n,1);
end
end
```

1.1.9. Tangentiales Softclipping

```
N = length(x) ;

Out = zeros(N,1) ;

tanh

for n = 1:N

out(n,1) = tanh(x(n,1)* Drive);

end

tanh  aproxximation Abel (2006)

for n = 1:N

out(n,1) = x(n,1)/((1+abs((x(n,1)))^2.5)^0.4);

end

tan (Tarr 2018)

for n = 1:N

out(n,1) = 2/pi * atan (x(n,1)*Drive);

end
```

1.2. Verzerrungs-Plugin

1.2.1. Oversampling

Das Skript für das Oversampling wurde mit viel Hilfe vom MATLAB Support (jibrahim MATLAB STAFF) realisiert. Die Konversationen sind öffentlich und können eingesehen werden. Danke für die Hilfe.

```
classdef (StrictDefaults)Copy_of_PickOversamplingSimplified < matlab.System & audioPlugin
    properties
        OversampleFactor {mustBeMember(OversampleFactor,{'1','2','16'})} = '1'
    end
    properties (Access = protected)
        poF = 1;
    end
    properties (Constant, Hidden)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('OversampleFactor',...
            'DisplayName','Oversample Fsctor','Mapping',{'enum','1','2','16'}))
    end
    properties (Access = private)
        Up2;
        Down2;
        Up16;
        Down16;
        pBuff
    end
    methods
        % Constructor
        function plugin = Copy_of_PickOversamplingSimplified
            plugin.Up2 = dsp.FIRInterpolator(2,designMultirateFIR(2,1));
            plugin.Up16 = dsp.FIRInterpolator(16,designMultirateFIR(16,1));
            plugin.Down2 = dsp.FIRDecimator(2,designMultirateFIR(1,2));
            plugin.Down16 = dsp.FIRDecimator(16,designMultirateFIR(1,16));
            plugin.pBuff = dsp.AsyncBuffer('Capacity', 4 * 192000-1);
        end
        function set.OversampleFactor(plugin,val)
            plugin.OversampleFactor = val;
            switch val
                case {'2'}
                    plugin.poF = 2; %#ok
                case{'16'}
                    plugin.poF = 16; %#ok
                otherwise
                    plugin.poF = 1; %#ok
            end
        end
    end
    methods(Access = protected)
        function out = stepImpl(plugin, in)
            o = plugin.poF;
            switch o
                case {2}
                    x = plugin.Up2(in);
                case{16}
                    x = plugin.Up16(in);
                otherwise
                    x = in;
            end

            %Distortion
            % TO BE ADDED HERE
            % workaround the limitation of FIRDecimator (no unbounded
            % inputs allowed)
            write(plugin.pBuff, x(:,1:size(in,2)));
            FrameLength = size(in,1);
            z = read(plugin.pBuff, o*FrameLength);
            z = z(1:min(4*196000, o*FrameLength),:);
               switch o
                case {2}
                    out = plugin.Down2(z);
                case{16}
                    out = plugin.Down16(z);
                otherwise
                    out = x;
            end
        end
        function resetImpl(plugin)
            reset(plugin.Up2);
            reset(plugin.Down2);
            reset(plugin.Up16);
            reset(plugin.Down16);
            reset(plugin.pBuff);
        end
    end
end
```

1.2.2. Finales Plugin

```
classdef (StrictDefaults)FinalDistortionPlugin < matlab.System & audioPlugin
    properties
        OversampleFactor {mustBeMember(OversampleFac-
tor,{'1','2','4','8','16','32','64'})} = '8'
         Drive=30;
        DryWet=1;
         Mode = OperatingModeDistortion.Distortion1;
         toggleAdjustOutputGain = false;
         setMaximum       = false;
         removeDcOffset = true;
    end
    properties (Access = protected)
        poF = 1;
    end


    properties (Access = private)

        distx;
        InMaximum=0;
        OutMaximum=0;
        setInMaximum=1;
        setOutMaximum=1;
    end

    properties
        LowCutoff = 20
        HighCutoff = 18e3;
        LowSlope = '12'
        HighSlope = '30'

        CenterFrequency1 = 100
        CenterFrequency2 = 1000
        CenterFrequency3 = 10000

        QualityFactor1 = 2
        QualityFactor2 = 2
        QualityFactor3 = 2

        PeakGain1 = 0
        PeakGain2 = 0
        PeakGain3 = 0

    end

    properties (Access = private)
        pVarSlopeBP
        pParamEQ
    end

    properties (Constant, Hidden)
         PluginInterface = audioPluginInterface( ...
            'InputChannels',2,...
            'OutputChannels',2,...
            'PluginName','Example EQ',...
            audioPluginParameter('LowCutoff', ...
            'DisplayName',  'Low Cutoff', ...
            'Label',  'Hz', ...
            'Mapping', { 'log', 20, 20000},...
            'Style', 'rotaryknob', 'Layout', [1 1]),...
            audioPluginParameter('HighCutoff', ...
            'DisplayName',  'High Cutoff', ...
            'Label',  'Hz', ...
            'Mapping', { 'log', 20, 20000},...
            'Style', 'rotaryknob', 'Layout', [1 2]),...
            audioPluginParameter('LowSlope', ...
            'DisplayName',  'Low Slope', ...
            'Label',  'dB/octave', ...
            'Mapping', { 'enum', '0','6','12','18','24','30','36','42','48'},...
            'Layout', [3 1]),...
```

```
            audioPluginParameter('HighSlope', ...
            'DisplayName',  'High Slope', ...
            'Label',  'dB/octave', ...
            'Mapping', { 'enum', '0','6','12','18','24','30','36','42','48'},...
            'Layout', [3 2]), ...
            audioPluginParameter('PeakGain1','DisplayName','Low Gain','Label','dB',...
            'Mapping',{'lin',-20,20},'Style','vslider','Layout',[5 1]),...
            audioPluginParameter('CenterFrequency1','DisplayName','Low Frequency','La-
bel','Hz',...
            'Mapping',{'log',20,20e3},'Style','rotaryknob','Layout',[7 1]),...
            audioPluginParameter('QualityFactor1','DisplayName','Low Q',...
            'Mapping',{'log',0.2,700},'Style','rotaryknob','Layout',[9 1]),...
            audioPluginParameter('PeakGain2','DisplayName','Mid Gain','Label','dB',...
            'Mapping',{'lin',-20,20},'Style','vslider','Layout',[5 2]),...
            audioPluginParameter('CenterFrequency2','DisplayName','Mid Frequency','La-
bel','Hz',...
            'Mapping',{'log',20,20e3},'Style','rotaryknob','Layout',[7 2]),...
            audioPluginParameter('QualityFactor2','DisplayName','Mid Q',...
            'Mapping',{'log',0.2,700},'Style','rotaryknob','Layout',[9 2]),...
            audioPluginParameter('PeakGain3','DisplayName','High Gain','Label','dB',...
            'Mapping',{'lin',-20,20},'Style','vslider','Layout',[5 3]),...
            audioPluginParameter('CenterFrequency3','DisplayName','High Frequency','La-
bel','Hz',...
            'Mapping',{'log',20,20e3},'Style','rotaryknob','Layout',[7 3]),...
            audioPluginParameter('QualityFactor3','DisplayName','High Q',...
            'Mapping',{'log',0.2,700},'Style','rotaryknob','Layout',[9 3]),...
            audioPluginParameter('OversampleFactor',...
            'DisplayName','Oversampling Factor','Map-
ping',{'enum','1','2','4','8','16','32','64'}),...
            audioPluginParameter('Mode',...
                'Mapping',{'enum','HardClip','Clip Positive Halvewave','Roehrenemula-
tion1','Roehrenemulation2','Diode','Halbwellengleichrichter','Vollwellengleichrich-
ter','Kubisches Softclipping','Tape 1','Tape 2','Tape 3','tanh','tanh
approx','arctan','Sin-Fold','Bitcrusher'}),...
            audioPluginParameter('Drive',...
             'DisplayName','Drive',...
             'Label','dB',...
             'Mapping',{'lin',0,60},...
             'Style', 'rotaryknob','Layout', [1 2]),...
             audioPluginParameter('DryWet',...
             'DisplayName','DryWet',...
             'Label','dB',...
            'Mapping',{'lin',0,1},...
             'Style', 'rotaryknob','Layout', [1 3]),...
             audioPluginParameter('toggleAdjustOutputGain', ...
            'Mapping', {'enum','Bypass','auto adjust output gain'}, ...
            'Layout',[1,1], ...
            'Style','vtoggle', ...
            'DisplayNameLocation','none'),...
             audioPluginParameter('removeDcOffset', ...
            'Mapping', {'enum','Bypass','removeDcOffset'}, ...
            'Layout',[1,1], ...
            'Style','vtoggle', ...
            'DisplayNameLocation','none'),...
        audioPluginParameter('setMaximum', ...
            'Mapping', {'enum','Bypass','setMaximum'}, ...
            'Layout',[1,1], ...
            'Style','vtoggle', ...
            'DisplayNameLocation','none'))



    end
    properties (Access = private)
        Up2;
        Down2;
        Up4;
        Down4;
        Up8;
        Down8;
        Up16;
        Down16;
        Up32;
        Down32;
        Up64;
        Down64;
```

```
        pBuff;
        AFilter;
        fs;

    end
    methods
        % Constructor
        function plugin = FinalDistortionPlugin
            plugin.Up2 = dsp.FIRInterpolator(2,designMultirateFIR(2,1));
            plugin.Up4 = dsp.FIRInterpolator(4,designMultirateFIR(4,1));
            plugin.Up8 = dsp.FIRInterpolator(8,designMultirateFIR(8,1));
            plugin.Up16 = dsp.FIRInterpolator(16,designMultirateFIR(16,1));
            plugin.Up32 = dsp.FIRInterpolator(32,designMultirateFIR(32,1));
            plugin.Up64 = dsp.FIRInterpolator(64,designMultirateFIR(64,1));
            plugin.Down2 = dsp.FIRDecimator(2,designMultirateFIR(1,2));
            plugin.Down4 = dsp.FIRDecimator(4,designMultirateFIR(1,4));
            plugin.Down8 = dsp.FIRDecimator(8,designMultirateFIR(1,8));
            plugin.Down16 = dsp.FIRDecimator(16,designMultirateFIR(1,16));
            plugin.Down32 = dsp.FIRDecimator(32,designMultirateFIR(1,32));
            plugin.Down64 = dsp.FIRDecimator(64,designMultirateFIR(1,64));
            plugin.pBuff = dsp.AsyncBuffer('Capacity', 32 * 192000-1);
            plugin.pVarSlopeBP = audiopluginexample.VarSlopeBandpassFilter;
            plugin.pParamEQ = audiopluginexample.ParametricEqualizerWithUDP;

            plugin.fs = plugin.getSampleRate;
            setSampleRate(plugin.pVarSlopeBP, plugin.fs);
            setSampleRate(plugin.pParamEQ, plugin.fs);
            plugin.AFilter = dsp.HighpassFilter('SampleRate',44100,'Filter-
Type','IIR','DesignForMinimumOrder',true,'PassbandFrequency',20,'StopbandFrequency',1,'S
topbandAttenuation',80);

        end
        function set.LowCutoff(plugin,val)
            plugin.pVarSlopeBP.LowCutoff = val;
            plugin.LowCutoff = val;
        end

        function set.HighCutoff(plugin,val)
            plugin.pVarSlopeBP.HighCutoff = val;
            plugin.HighCutoff = val;
        end

        function set.LowSlope(plugin,val)
            plugin.pVarSlopeBP.LowSlope = val;
            plugin.LowSlope = val;
        end

        function set.HighSlope(plugin,val)
            plugin.pVarSlopeBP.HighSlope = val;
            plugin.HighSlope = val;
        end

        function set.CenterFrequency1(plugin,val)
            plugin.pParamEQ.CenterFrequency1 = val;
            plugin.CenterFrequency1 = val;
        end

        function set.CenterFrequency2(plugin,val)
            plugin.pParamEQ.CenterFrequency2 = val;
            plugin.CenterFrequency2 = val;
        end

        function set.CenterFrequency3(plugin,val)
            plugin.pParamEQ.CenterFrequency3 = val;
            plugin.CenterFrequency3 = val;
        end

        function set.QualityFactor1(plugin,val)
            plugin.pParamEQ.QualityFactor1 = val;
            plugin.QualityFactor1 = val;
        end

        function set.QualityFactor2(plugin,val)
            plugin.pParamEQ.QualityFactor2 = val;
```

```
            plugin.QualityFactor2 = val;
        end

        function set.QualityFactor3(plugin,val)
            plugin.pParamEQ.QualityFactor3 = val;
            plugin.QualityFactor3 = val;
        end

        function set.PeakGain1(plugin,val)
            plugin.pParamEQ.PeakGain1 = val;
            plugin.PeakGain1 = val;
        end

        function set.PeakGain2(plugin,val)
            plugin.pParamEQ.PeakGain2 = val;
            plugin.PeakGain2 = val;
        end

        function set.PeakGain3(plugin,val)
            plugin.pParamEQ.PeakGain3 = val;
            plugin.PeakGain3 = val;
        end
        function set.OversampleFactor(plugin,val)
            plugin.OversampleFactor = val;
            switch val
                case {'2'}
                    plugin.poF = 2;
                case {'4'}
                    plugin.poF = 4;
                case {'8'}
                    plugin.poF = 8;
                case{'16'}
                    plugin.poF = 16;
                case{'32'}
                    plugin.poF = 32;
                case{'64'}
                    plugin.poF = 64;
                otherwise
                    plugin.poF = 1;
            end
        end
    end

      methods(Access = protected)
        function out = stepImpl(plugin, in)

            xin = in ;
             y = plugin.pVarSlopeBP(xin);
             xin = plugin.pParamEQ(y);

            o = plugin.poF;
            switch o
                case {2}
                    x = plugin.Up2(xin);
                case {4}
                    x = plugin.Up4(xin);
                case {8}
                    x = plugin.Up8(xin);
                case{16}
                    x = plugin.Up16(xin);
                case{32}
                    x = plugin.Up32(xin);
                case{64}
                    x = plugin.Up64(xin);
                otherwise
                    x = xin;
            end

                    N = length(x)    ;

                    xxout = zeros(N,2) ;

            switch (plugin.Mode)
                 case OperatingModeDistortion.Distortion1
```

```
%--------------------  Hard-Clip-Algorithmus (Tarr 2018)

                     thresh= plugin.Drive/60 ;


                     for n = 1:N

                     if x(n,1) >= thresh

                     xxout(n,1) = thresh;
                     elseif x(n,1) <= -thresh
                     xxout(n,1) = -thresh;
                     else
                     xxout(n,1) = x(n,1);
                     end

                     if x(n,2) >= thresh

                     xxout(n,2) = thresh;
                     elseif x(n,2) <= -thresh
                     xxout(n,2) = -thresh;
                     else
                     xxout(n,2) = x(n,2);
                     end
                     end

                   plugin.distx=xxout ;

                 case OperatingModeDistortion.Distortion2

   %-------1.1.2. Limitierung von positiven Halbwellen

                     thresh=  plugin.Drive/60 ;

                    for n=1:N

                        if x(n,1) >= thresh

                           xxout(n,1) = thresh;

                        else
                           xxout(n,1) = x(n,1);
                        end

                        if x(n,2) >= thresh

                           xxout(n,2) = thresh;

                        else
                           xxout(n,2) = x(n,2);
                        end

                    end

                     plugin.distx=xxout ;


                 case OperatingModeDistortion.Distortion3

 %-----------------     Roehrenemulation  (Araya und Suyama 1996)

                        x= x .* plugin.Drive/60;

                      for n=1:N

                          la = ( ( 3 * x(n,1) / 2 ) * (1- ( x(n,1) ^2 ) / 3 )) ;


                          lb = ( ( 3 * la / 2 ) * (1- ( la ^2 ) / 3 )) ;
```

```
                    xxout(n,1) = ( ( 3 * lb / 2 ) * (1- ( lb ^2 ) / 3 )) ;

                    ra = ( ( 3 * x(n,2) / 2 ) * (1- ( x(n,2) ^2 ) / 3 )) ;


                    rb = ( ( 3 * ra / 2 ) * (1- ( ra ^2 ) / 3 )) ;

                    xxout(n,2) = ( ( 3 * rb / 2 ) * (1- ( rb ^2 ) / 3 )) ;

                end

          plugin.distx=xxout ;

         case OperatingModeDistortion.Distortion4

%--------------Röhrenemulation (Doidic et al. 1998)
         x= x * plugin.Drive/60;


         for n = 1:N

         if  -1 <= x(n,1) && x(n,1) < -0.08905

             xxout(n,1) = (-3/4) .* (1-(1-(abs(x(n,1))-0.032847)).^(12)...
             + (1/3).*(abs(x(n,1))-0.032847)) + 0.01;

         elseif   -0.08905 <= x(n,1) &&  x(n,1)  < 0.320018

             xxout(n,1) = -6.153 * (x(n,1)).^2 +  3.9375 * x(n,1);

         elseif   0.320018 <= x(n,1) &&  x(n,1) <= 1

             xxout(n,1) = 0.630035;

         end

         if  -1 <= x(n,2) && x(n,2) < -0.08905

             xxout(n,2) = (-3/4) .* (1-(1-(abs(x(n,2))-0.032847)).^(12) ...
             + (1/3).*(abs(x(n,2))-0.032847)) + 0.01;

         elseif   -0.08905 <= x(n,2) &&  x(n,2)  < 0.320018

             xxout(n,2) = -6.153 * (x(n,2)).^2 +  3.9375 * x(n,2);

         elseif   0.320018 <= x(n,2) &&  x(n,2) <= 1

             xxout(n,2) = 0.630035;
         end
         end

             plugin.distx=xxout ;

           case OperatingModeDistortion.Distortion5

%------------1.1.6. Diode (Shockley 1949; Tarr 2018)

           x= x * plugin.Drive/60;

           for n = 1:N
               xxout(n,1) = .105 * (exp(0.1*x(n,1)/(1.68*0.0253)) - 1);
               xxout(n,2) = .105 * (exp(0.1*x(n,2)/(1.68*0.0253)) - 1);
           end

           plugin.distx=xxout ;


          case OperatingModeDistortion.Distortion6
```

```
%---------------HALFWAVERECTIFICATION          (Zölzer et al. 2002)

                    x= x * plugin.Drive/60;

                    for n = 1:N
                    if  x(n,1) >= 0

                        xxout(n,1) = x(n,1);
                    else

                        xxout(n,1) = 0;
                    end

                    if  x(n,2) >= 0

                        xxout(n,2) = x(n,2);
                    else

                        xxout(n,2) = 0;
                    end

                    end

                    plugin.distx=xxout ;

                case OperatingModeDistortion.Distortion7

%-------------------FULLWAVERECTIFICATION    (Zölzer et al. 2002)

                    x= x * plugin.Drive/60;

                    for n = 1:N

                    if x(n,1) >= 0

                       xxout(n,1) = x(n,1);
                    else

                       xxout(n,1) = -1*x(n,1);
                    end

                     if x(n,2) >= 0

                       xxout(n,2) = x(n,2);
                    else

                       xxout(n,2) = -1*x(n,2);
                    end
                    end

                    plugin.distx=xxout ;


                    case OperatingModeDistortion.Distortion8

     %-------------------1.1.8.  Kubisches Softclipping (Smith III 2008; Sullivan 1990)

                    x= x * plugin.Drive/60;

                    for n=1:N

                      if x(n,1) <= -1

                        xxout(n,1) = -2/3;

                      elseif -1 <= x(n,1) && x(n,1) <= 1

                        xxout(n,1) = x(n,1) - (((x(n,1))^3)/3 );
```

```
    elseif x(n,1) >= 1

     xxout(n,1) = 2/3;
    end

    if x(n,2) <= -1

      xxout(n,2) = -2/3;

    elseif -1 <= x(n,2) && x(n,2) <= 1

      xxout(n,2) = x(n,2) - (((x(n,2))^3)/3 );

    elseif x(n,2) >= 1

     xxout(n,2) = 2/3;
    end

  end

  plugin.distx=xxout ;

  case OperatingModeDistortion.Distortion9
%-------------------Kubisches Softclipping Modified    2

  x= x * plugin.Drive/60;

  for n = 1:N

  if 0 <= x(n,1) && x(n,1) <= 1

     xxout(n,1) = x(n,1) - (((x(n,1))^2)/3 );

  elseif x(n,1) < 0

     xxout(n,1) = x(n,1);
  end

  if 0 <= x(n,2) && x(n,2) <= 1

     xxout(n,2) = x(n,2) - (((x(n,2))^2)/3 );

  elseif x(n,2) < 0

     xxout(n,2) = x(n,2);
  end


  end

  plugin.distx=xxout ;

  case OperatingModeDistortion.Distortion10
%-------------------Kubisches Softclipping Modified    2.5

  x= x * plugin.Drive/60;

  for n = 1:N

  if 0 <= x(n,1) && x(n,1) <= 1

     xxout(n,1) = x(n,1) - (((x(n,1))^2.5)/3 );

  elseif x(n,1) < 0

     xxout(n,1) = x(n,1);
```

```
                end

                if 0 <= x(n,2) && x(n,2) <= 1

                   xxout(n,2) = x(n,2) - (((x(n,2))^2.5)/3 );

                elseif x(n,2) < 0

                   xxout(n,2) = x(n,2);
                end


                end

                plugin.distx=xxout ;

                case OperatingModeDistortion.Distortion11

 %-------------------Kubisches Softclipping Modified    3

                x= x * plugin.Drive/60;

                for n = 1:N

                if 0 <= x(n,1) && x(n,1) <= 1

                   xxout(n,1) = x(n,1) - (((x(n,1))^3)/3 );

                elseif x(n,1) < 0

                   xxout(n,1) = x(n,1);
                end

                if 0 <= x(n,2) && x(n,2) <= 1

                   xxout(n,2) = x(n,2) - (((x(n,2))^3)/3 );

                elseif x(n,2) < 0

                   xxout(n,2) = x(n,2);
                end

                end

                plugin.distx=xxout ;

                case OperatingModeDistortion.Distortion12

%-------------------tanh


                for n = 1:N

                xxout(n,1) = tanh(x(n,1)*plugin.Drive);
                xxout(n,2) = tanh(x(n,2)*plugin.Drive);

                end

                plugin.distx=xxout ;

                case OperatingModeDistortion.Distortion13

  %-------------------tanh   aprox

                x= x * plugin.Drive;

                for n = 1:N

                xxout(n,1) = x(n,1)/((1+abs((x(n,1)))^2.5)^0.4);
```

```
                    xxout(n,2) = x(n,2)/((1+abs((x(n,2)))^2.5)^0.4);

                    end

                    plugin.distx=xxout ;


                    case OperatingModeDistortion.Distortion14

 %--------------------tan      (Tarr 2018)

                    for n = 1:N

                    xxout(n,1) = 2/pi * atan (x(n,1)*plugin.Drive);
                    xxout(n,2) = 2/pi * atan (x(n,2)*plugin.Drive);

                    end


                    plugin.distx=xxout ;

                    case OperatingModeDistortion.Distortion15

%--------------------sin fold

                    for n = 1:N

                        xxout(n,1) = sin(x(n,1)*plugin.Drive);
                        xxout(n,2) = sin(x(n,2)*plugin.Drive);

                    end

                     plugin.distx=xxout ;

                    case OperatingModeDistortion.Distortion16

%-------------------%HackAudio BitCrusher (Tarr 2018)

                    for n = 1:N

                    nBits        = (( plugin.Drive/4 ) +1 )   ;

                    ampValues    = 2 ^ (nBits - 1) ;

                    xxout(n,1) = 0.9 * ceil(x(n,1) * ampValues)*(1/ampValues)   ;
                    xxout(n,2) = 0.9 * ceil(x(n,2) * ampValues)*(1/ampValues)   ;

                    end

                    plugin.distx=xxout ;

                 otherwise

                     plugin.distx = x ;
            end

            write(plugin.pBuff, plugin.distx(:,1:size(in,2)));
            FrameLength = size(in,1);
            z = read(plugin.pBuff, o*FrameLength);
            z = z(1:min(32*196000, o*FrameLength),:);
            switch o
                case {2}
                    xout = plugin.Down2(z);
                case {4}
                    xout = plugin.Down4(z);
                case {8}
                    xout = plugin.Down8(z);
                case{16}
                    xout = plugin.Down16(z);
                case{32}
```

```
                xout = plugin.Down32(z);
            case{64}
                xout = plugin.Down64(z);
            otherwise
                xout = plugin.distx;
        end

     if plugin.removeDcOffset

     xout = plugin.AFilter((xout(:,1:2)));

     end

         if  plugin.toggleAdjustOutputGain

               plugin.InMaximum = max(max(abs(in(:,:))));

               plugin.OutMaximum = max(max(abs(xout(:,:))));

               if plugin.setMaximum

               plugin.setInMaximum = plugin.InMaximum;
               plugin.setOutMaximum = plugin.OutMaximum;

               end

               ratioinout = plugin.setInMaximum/plugin.setOutMaximum;

               xout(:,:) = xout(:,:)*ratioinout;
        end

        out = ( ( xout .* plugin.DryWet ) + ( in .* ( 1 - plugin.DryWet ) ) )   ;

    end
    function resetImpl(plugin)
        reset(plugin.Up2);
        reset(plugin.Down2);
        reset(plugin.Up4);
        reset(plugin.Down4);
        reset(plugin.Up8);
        reset(plugin.Down8);
        reset(plugin.Up16);
        reset(plugin.Down16);
        reset(plugin.Up32);
        reset(plugin.Down32);
        reset(plugin.Up64);
        reset(plugin.Down64);
        reset(plugin.pBuff);
        plugin.fs = plugin.getSampleRate;
        setSampleRate(plugin.pVarSlopeBP, plugin.fs);
        reset(plugin.pVarSlopeBP);
        setSampleRate(plugin.pParamEQ, plugin.fs);
        reset(plugin.pParamEQ);
%
    end
  end
end
```

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 Vorname Nachname