Gesperrt bis 20.11.2016

Entwicklung eines VST-Plug-ins für die Matrizierung eines Doppel-MS Mikrofonarrays

Bachelorarbeit

im Studiengang
Audiovisuelle Medien

vorgelegt von

Tobias Gutenkunst
Matr.-Nr.: 25716

am 26. November 2015
an der Hochschule der Medien Stuttgart

zur Erlangung des akademischen Grades eines
Bachelor of Engineering

Erstprüfer:	Prof. Oliver Curdt
Zweitprüfer:	Bernfried Runow

Eidesstattliche Versicherung

Name:	Gutenkunst	Vorname:	Tobias
Matrikel-Nr.:	25716	Studiengang:	Audiovisuelle Medien (AM7)

Hiermit versichere ich, Tobias Gutenkunst, an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel „Entwicklung eines VST-Plug-ins für die Matrizierung eines Doppel-MS Mikrofonarrays“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der eidesstattlichen Versicherung und prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO der Hochschule der Medien Stuttgart) sowie die strafrechtlichen Folgen (siehe unten) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

Auszug aus dem Strafgesetzbuch (StGB)

§ 156 StGB Falsche Versicherung an Eides Statt

Wer von einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Ort, Datum

Unterschrift

Kurzfassung

Gegenstand der vorliegenden Arbeit ist die Entwicklung eines Audio-Plug-ins, welches aus einem angelegten Doppel-MS Signal in Echtzeit ein Signal eines virtuellen Mikrofons erzeugt. Diese Mikrofonaufstellung erlaubt, dass der Winkel in der horizontalen Ebene und die Richtcharakteristik des virtuellen Mikrofons in erster Ordnung beliebig wählbar sind. Zusätzlich erfolgt eine Geräuschreduzierung im spektralen Bereich, um die Richtwirkung zu verbessern. Über die Oberfläche des Plug-ins sind Winkel und Richtcharakteristik des virtuellen Mikrofons sowie Intensität der Geräuschreduzierung und Richtcharakteristik des Störgeräuschsignals steuerbar. Somit kann man diese Parameter im Gegensatz zur herkömmlichen Mikrofonaufstellung noch während oder nach der Aufnahme verändern.

Schlagwörter: Beamforming, VST-Plug-in, Doppel-MS, Geräuschreduzierung, Mikrofonarray

Abstract

The aim of this thesis is to implement an audio plug-in, which processes a double M/S input into an output of a virtual microphone in real time. This setup allows any angle in the horizontal plane and any first order polar pattern of the virtual microphone. Spectral noise reduction is used for better beamforming directivity. The user interface allows control over four parameters: virtual microphone angle, virtual microphone polar pattern, noise reduction intensity and noise polar pattern. These parameters can be still altered while or after recording.

Keywords: beamforming, VST-plug-in, double M/S, noise reduction, microphone array

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

MS	Mid/Side, Mikrofonaufstellung für Zweikanal Stereo
Doppel-MS	Doppel Mid/Side Mikrofonarray
DAW	Musikproduktionssoftware (engl.: Digital Audio Workstation)
FFT	Fast Fourier Transform
VST	Virtual Studio Technology, Schnittstelle für Audio-Plug-ins
AAX	Avid Audio Extension, Schnittstelle für Audio-Plug-ins
AU	Audio Unit, Schnittstelle für Audio-Plug-ins
WDL-OL	Bibliothek zur Erstellung von Audio-Plug-ins

1 Einleitung

Bei der herkömmlichen Mikrofonierung wählt man vor der Aufnahme die Position und die Richtcharakteristik des Mikrofons aus. Diese Eigenschaften können während oder nach der Aufnahme nicht mehr verändert werden. Falls sich die Schallquelle bewegt und somit aus der Keule des Mikrofons heraustritt, kann das zu unerwünschten Ergebnissen in der Aufnahme führen. Das Signal wird leiser und durch das frequenzabhängige Abstrahlverhalten der Schallquelle verändert sich das Frequenzspektrum der Aufnahme. Um dies zu verhindern, kann man Druckempfängermikrofone einsetzen, welche den Schall aus allen Richtungen gleichmäßig aufnehmen. Daraus ergibt sich aber bei im Raum vorkommenden Störgeräuschen ein ungünstiges Verhältnis von Nutzschall zu Störgeräusch, da der aus allen Richtungen kommende Störschall lauter aufgenommen wird, als bei Richtcharakteristiken mit hohem Bündelungsmaß.

Es ist also wünschenswert eine Nachbearbeitung der Richtwirkung und des Aufnahmewinkels des Mikrofons zu ermöglichen.

Die Eigenschaften der Mikrofonaufnahme können durch Zusammenschalten mehrerer Mikrofone verbessert werden. Diese Mikrofonaufstellungen nennt man Mikrofonarrays. So ist es mit Mikrofonarrays möglich, Aufnahmerichtung und Richtcharakteristik des Mikrofons auch während oder nach der Aufnahme zu verändern. Es gibt viele unterschiedliche Mikrofonarrays, welche verschiedene Vor- und Nachteile bezüglich Frequenzlinearität, Richtwirkung, Baugröße, Ortungsfähigkeit und Latenz mit sich bringen.

Koinzidente Arrays sind kompakt, bieten eine variable Ausrichtung und variable Richtcharakteristik. Außerdem ist der Frequenzgang weitestgehend linear. Hier sind aber nur Richtcharakteristiken erster Ordnung umsetzbar, welche ein relativ geringes Bündelungsmaß haben. Dieses kann zusätzlich durch adaptive Geräuschreduzierung verstärkt werden, indem der Störgeräuschanteil aus dem Nutzsignal entfernt wird.

Runow (2015b) hat dazu zwei Verfahren Entwickelt, welche eine Matrizierung eines Doppel-MS-Signals mit Geräuschreduzierung vornehmen.

Ziel dieser Arbeit ist es, ein Audio Plug-in zu entwickeln, welches den Beamformingalgorithmus für Doppel-MS Mikrofonarrays mit Geräuschreduzierung in der Frequenzdomäne von Runow (2015b) in Echtzeit durchführen kann, wobei die Parameter auch in Echtzeit ansteuerbar sind. Außerdem werden Funktionalität und Probleme evaluiert.

2 Grundlagen

2.1 Richtcharakteristik bei Mikrofonen

Die Richtcharakteristik bestimmt wie ein Mikrofon Schall aus unterschiedlichen Richtungen abnimmt.

Eine stärkere Richtwirkung kann durch unterschiedliche Methoden erzielt werden. Die geringste Richtwirkung haben Mikrofone mit Kugelcharakteristik. Sie werden auch Druckempfänger genannt, da sie den Schalldruck nur an einem Punkt messen. Im Gegensatz dazu stehen die Druckgradientenempfänger, welche den Schalldruckunterschied zwischen zwei Punkten messen. Das einfachste Beispiel dafür ist die Achtercharakteristik, bei der der Schall von allen Seiten auf die Membran treffen kann. Bei seitlichem Schalleinfall wird die Membran also nicht ausgelenkt. Bei Schalleinfall von vorne ist der Weg der Schallwelle von Vorder- zu Rückseite maximal, wodurch ein maximaler Druckunterschied zwischen Membranvorder- und Rückseite entsteht. Es ergibt sich eine stärkere Richtwirkung als bei einer Kugelcharakteristik (Görne 2007, S. 35).

Für andere Richtcharakteristiken werden Laufzeitglieder dazu verwendet, den Schall an der Rückseite der Membran zeitversetzt ankommen zu lassen. Bei 180 Grad Phasendrehung ergibt sich dabei die Richtcharakteristik einer Niere. Um den Phasenversatz frequenzunabhängig zu halten, wird eine frequenzabhängige Dämpfung im Laufzeitglied eingesetzt (Dickreiter 2014, S. 154).

Eine variable Richtcharakteristik kann mechanisch erzielt werden, indem umschaltbare Elemente im Laufzeitglied den Phasenversatz verändern. Eine andere Methode besteht darin zwei Membranen in einem Mikrofon zu verschalten. Durch Addition und Subtraktion der beiden Signale erhält man die gewünschte Richtwirkung.

2.2 Mikrofonarrays

Als Mikrofonarray bezeichnet man eine Aufstellung von mehreren Mikrofonen aus denen man ein Signal durch Addition, Subtraktion oder Verzögerung berechnet. Das Ziel dieser Aufstellungen ist es einerseits die Richtung des Aufgenommenen Schalls zu bestimmen. Andererseits ist es dadurch auch möglich die Richtwirkung zu verbessern. Diesen Vorgang nennt man Beamforming. Man unterscheidet zwischen räumlichen und koinzidenten Arrays, also solchen, bei denen die Mikrofonkapseln in einem bestimmten Abstand zueinander aufgestellt sind und solchen, bei denen sich die Mik-

rofonkapseln auf einem Punkt befinden. Dabei haben räumliche und koinzidente Arrays unterschiedliche Eigenschaften (Runow und Curdt, 2014).

2.2.1 Räumliche Arrays

Räumliche Arrays haben einen definierten Abstand zwischen den einzelnen Mikrofonkapseln. Dies sorgt für eine Zeitdifferenz zwischen den abgenommenen Signalen der Mikrofone. Aus diesem Phasenversatz lässt sich die Richtung des eintreffenden Schalls bestimmen.

Für räumliche Arrays sollte der Abstand der Mikrofone kleiner sein als die Hälfte der kleinsten zu übertragenden Wellenlänge. Für additive räumliche Arrays kommt hinzu, dass das Array größer als die größte zu übertragende Wellenlänge sein muss. Räumliche Mikrofonarrays mit additiver Signalverarbeitung brauchen also große Maße und viele Mikrofonkapseln um das gesamte Spektrum des Hörbereichs abzudecken (Runow und Curdt, 2014).

2.2.2 Koinzidente Arrays

Bei koinzidenten Mikrofonarrays befinden sich alle Mikrofonkapseln auf einem Punkt. Koinzidente Arrays haben also theoretisch keinen Phasenversatz zueinander. Praktisch ist dies nicht umzusetzen, da die einzelnen Kapseln nicht auf dem selben Punkt platziert werden können. Für hohe Frequenzen treten hier also Kammfilter auf.

2.2.3 Gradientensynthese

Ein koinzidentes Mikrofonarray mit zwei Membranen wird bei der Gradientensynthese eingesetzt. Die beiden Signale werden addiert. Das Verhältnis der beiden Signale zueinander bestimmt dabei die Richtcharakteristik am Ausgangssignal. Man erzielt dadurch eine variable Richtcharakteristik erster Ordnung.

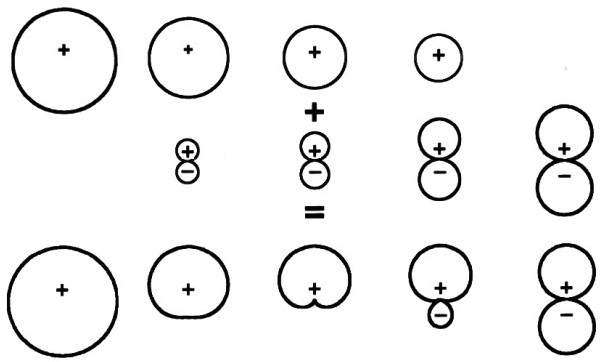

Abbildung 1: Gradientensynthese mit Kugel und Acht (Runow und Curdt, 2014)

Dabei können entweder zwei Nieren oder eine Kugel und eine Acht verschaltet werden. Beide Varianten sind gleichwertig. Für das Array mit Kugel w(t) und Acht x(t) gilt für das Ausgangssignal u(t):

$$u(t) = A \cdot w(t) + (1 - A) \cdot x(t)$$

A ist dabei ein Koeffizient mit $0 \leq A \leq 1$, der beide Anteile gewichtet (Runow und Curdt, 2014).

2.2.4 Mid/Side Array

MS (Mid/Side) ist ein Stereoverfahren bei dem ein Mittensignal M und ein Seitensignal S aufgenommen werden und dann die erwünschten Kanäle per Addition und Subtraktion errechnet werden. Für das Mittensignal wird normalerweise eine nach vorne gerichtete Niere verwendet, es können aber auch andere Richtcharakteristiken verwendet werden. Das Seitensignal wird mit einer dazu um 90 Grad gedrehten Acht aufgenommen. Die phasenrichtige Seite des Mikrofons zeigt nach links.

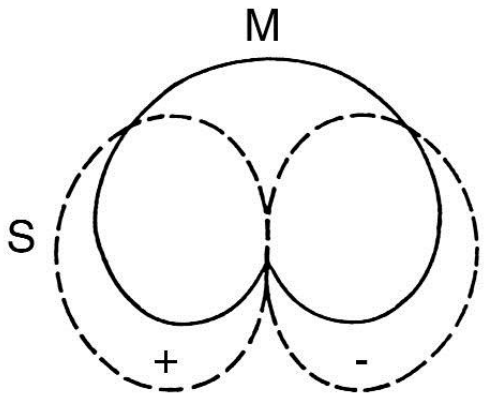

Abbildung 2: MS Richtcharakteristik mit Niere als Mittensignal (Wuttke, www.ingwu.de)

Um die Zweikanal Stereo Signale L und R zu bekommen, berechnet man (Dickreiter 2014, S. 254):

$$L = (M + S) \cdot 1/\sqrt{2}$$

$$R = (M - S) \cdot 1/\sqrt{2}$$

Die matrizierten Signale L und R entsprechen einer bestimmten äquivalenten XY Aufstellung. Der Öffnungswinkel zwischen L und R und deren Richtcharakteristik ist abhängig von der Richtcharakteristik von M und dem Mischungsverhältnis von M und S. Man kann also durch verstärken des Seitensignals einen größeren Öffnungswinkel zwischen L und R erzeugen. Allerdings verändert man so automatisch die Richtcharakteristik der beiden Signale (Görne 2007, S. 102f). Dies wird in Abbildung 3 veranschaulicht:

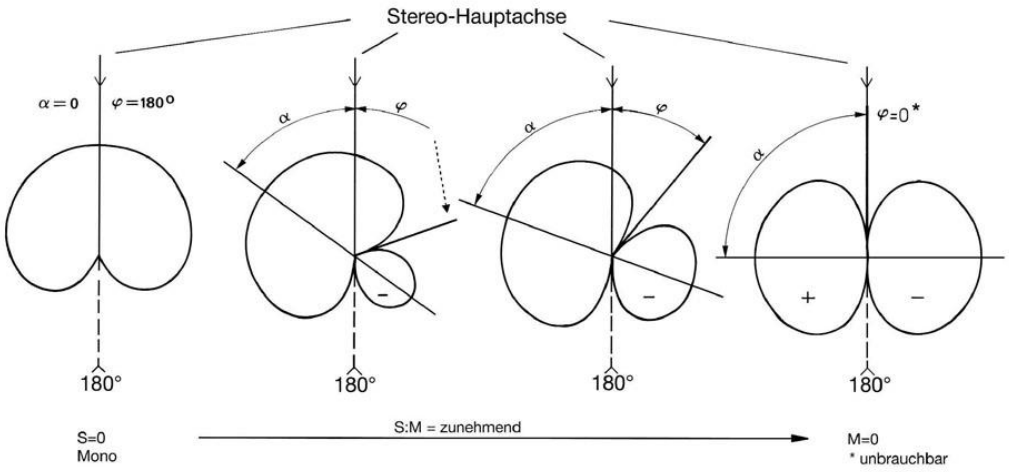

Abbildung 3: linker Kanal nach Matrizierung (M+S) (Wuttke, www.ingwu.de)

2.2.5 Blumlein

Blumlein ist ein koinzidentes Stereoverfahren, bei dem zwei senkrecht zueinander stehende Mikrofone mit Achtercharakteristik verwendet werden (Abbildung 4). Aufgrund der guten Trennung beider Kanäle und der Richtwirkung nach hinten kann so eine gute Räumlichkeit dargestellt werden.

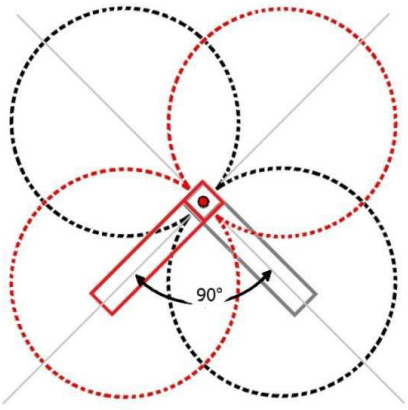

Abbildung 4: Anordnung nach Blumlein (Runow 2015a)

Eine weitere Eigenschaft dieser Aufstellung ist, dass man durch Matrizieren der beiden Kanäle mit geeigneter Gewichtung ein virtuelles Mikrofon mit Achtercharakteristik erzeugen kann. Für die vom Einfallswinkel θ abhängige Empfindlichkeit Γ_V des virtuellen Mikrofons gilt:

$$\Gamma_V(\theta) = k_x \cdot cos(\theta) + k_y \cdot sin(\theta)$$

Wobei k_x und k_y die Gewichtungskoeffizienten der Mikrofone in x und y Richtung sind. Für sie gilt mit dem Ausrichtungswinkel α des virtuellen Mikrofons (Runow 2015a):

$$k_x = \cos(\alpha)$$

$$k_y = \sin(\alpha)$$

Wobei beide Mikrofone nicht wie beim Stereoverfahren um 45 Grad zur Mitte gedreht sind, sondern Mikrofon x nach vorne zeigt und y dementsprechend orthogonal dazu. Der Winkel des virtuellen Mikrofons kann in der Ebene also beliebig gewählt werden.

2.2.6 Ambisonics

Ambisonics ist ein Mikrofonierungsverfahren, welches das gesamte Schallfeld an einem Punkt aufzeichnet. Nötig ist dafür ein koinzidentes Mikrofonarray mit vier Kapseln. Dabei gibt es zwei verschiedene Formate. Das A-Format (auch Soundfield genannt) besteht aus vier breite Niere Kapseln in tedraedischer Aufstellung. Das B-Format wird mit einer Kugel und drei orthogonal zueinander stehenden Achten realisiert. Beide Systeme sind ineinander überführbar und somit gleichwertig. Für die A-Format-Signale LF, RF, LB und RB und B-Format-Signale W, X, Y und Z gilt (Görne 2007, S. 108):

$$W = \frac{LF + LB + RF + RB}{2}$$

$$X = \frac{LF - LB + RF - RB}{2}$$

$$Y = \frac{LF + LB - RF - RB}{2}$$

$$Z = \frac{LF - LB + RB}{2}$$

Durch Matrizieren der vier Kanäle kann man eine beliebige Richtcharakteristik erster Ordnung mit beliebigem Raumwinkel erzeugen. Dabei kann man Format-B als dreidimensionales Blumleinarray betrachten, bei dem die resultierende Acht mit der Kugel per Gradientensynthese matriziert wird.

2.2.7 Doppel-MS Array

Doppel-MS ist ein koinzidentes Mikrofonarray mit drei Mikrofonkapseln. Dabei wird das MS-Array durch eine weitere Niere erweitert, welche in die Gegenrichtung der ersten Niere zeigt. Dieses Format wurde als Surround-Mikrofonarray entwickelt. Dabei wird ursprünglich die vordere Niere mit dem Seitensignal der Acht für die Matri-

zierung der Kanäle L und R verwendet; die hintere Niere mit dem Seitensignal für Kanäle Ls und Rs. Die vordere Niere liefert das Center Signal.

Doch die Doppel-MS Aufstellung bietet den besonderen Vorteil, dass im Gegensatz zum normalen MS-Verfahren keine Abhängigkeit zwischen Winkel und Richtcharakteristik besteht, wenn man alle drei Signale in die Berechnung mit einbezieht. Hierbei kann man die Aufstellung in ein zweidimensionales Ambisonics Format B überführen, indem man aus den beiden Nieren durch Subtrahieren eine Acht und durch Addieren eine Kugel erzeugt. Durch Addieren und Subtrahieren der drei Signale kann so ein virtuelles Mikrofon mit beliebiger Charakteristik erster Ordnung mit beliebigem Winkel in der horizontalen Ebene matriziert werden (Wittek, Haut und Keinath, 2006, S. 5). Es gilt für das Ausgangssignal v(t), mit dem Winkel α und dem Druckanteil A (Runow 2015a, S. 5):

$$v(t) = niere_{vorne}(t) \cdot (A + (1 - A)cos\alpha)$$
$$+ \ niere_{hinten}(t) \cdot (A - (1 - A)cos\alpha)$$
$$+ acht(t) \cdot ((1 - A)sin\alpha)$$

Abbildung 5: Schoeps Doppel-MS Anordnung (Wuttke, www.ingwu.de)

2.3 Fourier Transformation

Ein wichtiger Bestandteil der Signalverarbeitung ist die Fourier Transformation. Sie besagt, dass ein komplexes zeitliches Signal auch als Summe von Sinuswellen verschiedener Frequenzen, Amplituden und Phasen dargestellt werden kann. Ein Signal

s(t) kann also vom Zeitbereich in den Spektralbereich transformiert werden (Görne 2014, S. 143):

$$S(f) = \int_{-\infty}^{\infty} s(t) e^{-j2\pi f t} dt$$

Diese Funktion ist komplex. Als Ergebnis erhält man also für jede Frequenz einen Realteil und einen Imaginärteil. Eine anschaulichere, aber gleichwertige Betrachtung erhält man, wenn man dies in Amplitude $|S(f)|$ und Phasenwinkel $\varphi(f)$ umwandelt:

$$\varphi(f) = \arctan\left(\frac{Im\{S(f)\}}{Re\{S(f)\}}\right)$$

$$|S(f)| = \sqrt{Re^2\{S(f)\} + Im^2\{S(f)\}}$$

Um das Spektrum *S(f)* wieder in ein Zeitsignal *s(t)* zu wandeln verendet man die inverse Transformation (Görne 2014, S. 143):

$$s(t) = \int_{-\infty}^{\infty} S(f) e^{j2\pi f t} df$$

Diese Formel gilt für kontinuierliche Signale. Signale werden in der digitalen Signalverarbeitung aber abgetastet. Sie liegen also nicht zeitkontinuierlich, sondern nur diskret vor. Für diskrete Signale gibt es die diskrete Fourier Transformation DFT (Görne 2014, S. 146):

$$X(k) = \sum_{n=0}^{M-1} x(n) e^{-j2\pi nk/M} \qquad k = 0, \dots, M-1$$

Für die inverse Transformation gilt (Görne 2014, S. 146):

$$x(n) = \frac{1}{M} \sum_{k=0}^{M-1} X(k) e^{j2\pi nk/M} \qquad n = 0, \dots, M-1$$

Die diskrete Fourier Transformation gibt ein diskretes Spektrum aus. Je nach Länge M des zeitlichen Signals x(n), verändert sich die Auflösung des Frequenzspektrums X(k). Für M Zeitwerte gibt die DFT M Frequenzwerte zurück. Dabei ist zu beachten, dass die spektrale Auflösung der Frequenzen linear und nicht wie das menschliche Gehör logarithmisch ist. Für tiefe Oktaven erhält man also eine schlechtere Auflösung als für hohe Oktaven.

In der Praxis werden Signale nicht am Stück transformiert, sondern in Abschnitte bestimmter Länge unterteilt. Da die Fourier Transformation Signale als periodisch betrachtet, entstehen bei der Transformation an den Rändern der Abschnitte Fehler. Um diese Fehler zu minimieren, gewichtet man die Ränder der Abschnitte weniger stark

als die Mitte. Diesen Vorgang nennt man Fensterung. Dabei werden die Fenster so überlappt, dass die Gesamtenergie des Signals erhalten bleibt. Eine häufig verwendete Fensterfunktion ist das Hanning-Fenster. Es blendet das Signal cosinusförmig ein und aus (Görne 2014, S. 146):

$$w(n) = 0{,}5 - 0{,}5\cos(\frac{2\pi n}{M})$$

Eine Implementierung der diskreten Fourier Transformation ist die schnelle Fourier Transformation (engl.: FFT, fast fourier transform). Diese ist wesentlich schneller als andere Implementationen, hat aber den Nachteil, dass nur Fenster der Länge einer Zweierpotenz erlaubt sind ($M = 2^n$). Ein typischer Wert ist $M = 1024$. Bei einer Abtastrate von 44100 Hz ergibt sich daraus eine Fensterlänge von $1024/44100Hz \approx 23\ ms$ und eine spektrale Auflösung von $44100Hz/512 \approx 86Hz$.

Um eine höhere spektrale Auflösung zu erreichen, kann man das Eingangssignal mit Nullen auffüllen, zum Beispiel auf die doppelte Länge. Dies nennt man Zero-Padding. Dadurch erreicht man eine spektrale Auflösung von $44100Hz/1024 \approx 43Hz$.

Die inverse Transformation wird auch Fourier Synthese genannt. In der Verarbeitung von Audiosignalen kann die Fourier Transformation dazu verwendet werden ein Signal im Spektralbereich zu bearbeiten und wieder zu resynthetisieren. Man kann also unerwünschte Frequenzbänder entfernen oder andere Frequenzen verstärken. Zu beachten gilt dabei, dass eine starke spektrale Veränderung des Signals nach dem Synthetisieren als Fehler hörbar werden kann. Diese Fehler treten zum Beispiel bei niedrigen Codierungsraten verlustbehafteter Datenreduktionsverfahren wie MP3 auf.

2.4 Audio-Plug-ins

Audio Plug-ins sind Programme, die sich in DAWs (Musikproduktionsumgebungen) einbinden lassen. Man unterscheidet zwischen Effekt-Plug-ins und Instrumenten-Plug-ins. Instrumenten-Plug-ins sind Klangerzeuger, welche meistens kein Inputsignal erhalten (abgesehen von MIDI-Steuerdaten). Effekt-Plug-ins verarbeiten am Input angeschlossene Klänge und geben sie als Output aus. Dabei wird dem Plug-in in der Regel das Signal pro Kanal Paketweise übergeben und das Plug-in gibt das verarbeitete Signal wieder zurück. Einen Spezialfall bilden hier die Analyse-Plug-ins, welche das Signal visualisieren, aber keine Klangänderung vornehmen. Im Folgenden wird ein kleiner Überblick auf unterschiedliche Plug-in-Schnittstellen gegeben. Diese unterscheiden sich weniger in ihren Funktionen, als in der Unterstützung durch unterschiedliche Hostprogramme. Die meisten Plug-ins werden für alle gängigen Plug-in-Schnittstellen angeboten.

2.4.1 VST

Virtual Studio Technology wurde 1996 von Steinberg entwickelt und ist wohl das am weitesten verbreitete Format für Audio Plug-ins. Es wird von vielen DAWs unterstützt und liegt aktuell in der dritten Version vor.

2.4.2 AAX

AAX (Avid Audio Extension) ist die Plug-in Schnittstelle von Avid, dem Hersteller von Pro Tools. Aufgrund der weiten Verbreitung von Pro Tools in professionellen Tonstudios ist diese Schnittstelle auch von Bedeutung. AAX hat die ältere Avid Schnittstelle RTAS (Real Time Audio Suite) abgelöst.

2.4.3 AU

AU (Audio Unit) ist die Plug-in Schnittstelle von Apples core audio System. Verwendet werden sie unter anderem in Apples DAW Logic.

2.4.4 JUCE

JUCE ist eine Bibliothek zur Schnittstellenübergreifenden Erstellung von Audio Plug-ins. Sie bietet eine gute Dokumentation und unterstützt viele Plug-in Formate. Zusätzlich werden viele Optionen für die Erstellung der Bedienoberfläche gegeben. JUCE ist aber nur für Open Source Plug-ins kostenlos, für andere Zwecke recht teuer.

2.4.5 WDL-OL/IPlug

WDL (engl.: whittle) kann man als kostenlose open-source Alternative zu JUCE ansehen. Dabei bietet die von Oli Larkin weiterentwickelte Version WDL-OL/IPlug auch die Möglichkeit, den Code in verschiedene Schnittstellen zu exportieren (VST, VST3, AAX, AU). Die Dokumentation ist dabei aber nicht so ausführlich wie bei JUCE.

3 Zwei Verfahren zur Störgeräuschreduktion mit koinzidenten Mikrofonarrays nach Runow

Runow (2015b) hat zwei Verfahren entwickelt eine Störgeräuschreduktion für ein Doppel-MS Minkrofonarray durchzuführen. Bei beiden Verfahren wird zuerst ein Nutzsignal $x_{util}(t)$ in Richtung der Schallquelle matriziert und ein Störsignal $x_{ambi}(t)$, welches von der Nutzschallquelle weg zeigt. Wobei $x_{ambi}(t)$ möglichst genau den Störschall einfangen soll, den $x_{util}(t)$ zusätzlich zum Nutzschall abnimmt. Dabei kann der Nutzschall als Direktschallfeld und der Störgeräuschschall als Diffusschallfeld angesehen werden. $x_{ambi}(t)$ tritt also ungefähr von allen Seiten gleich auf. Um das Signal der Nutzschallquelle zu erhalten, muss $x_{ambi}(t)$ aus $x_{util}(t)$ entfernt werden. Das direkte subtrahieren der beiden Zeitsignale führt zum falschen Ergebnis, da $x_{ambi}(t)$ nicht genau dem Störschallanteil in $x_{util}(t)$ entspricht.

3.1 Subtraktion im Spektralbereich

Das erste der beiden Verfahren verwendet die Subtraktion im Spektralbereich. Dabei werden die Signale per DFT in den Frequenzraum transformiert, um dort die Absolutwerte voneinander abzuziehen:

$$|Y(k)| = |X_{util}(k)| - |X_{ambi}(k)| \cdot w(k)$$

Dabei ist w(k) ein reeller Faktor mit dem einerseits die Intensität der Geräuschreduzierung eingestellt werden kann. Andererseits wird es dazu verwendet den Gesamtwert der Berechnung nicht negativ werden zu lassen.

Durch dieses Verfahren lässt sich eine deutlich wahrnehmbare Erhöhung der Richtwirkung erziehlen. Allerdings entstehen dabei Artefakte in Form von „musical tones“ (Runow 2015b).

3.2 Bearbeitung mit einer Mel-Filterbank

Ein Problem der Subtraktion im Spektralbereich besteht darin, dass die DFT das Frequenzspektrum linear aufteilt. Das menschliche Gehör verarbeitet es im Gegensatz dazu logarithmisch. Die DFT ist also für hohe Oktaven gut und für niedrige Oktaven schlecht aufgelöst.

Um eine bessere Anpassung der Frequenzbänder an die menschliche Wahrnehmung zu erreichen, verwendet das zweite Verfahren eine Mel-Filterbank. Diese ist genau der

Tonheit nachempfunden, also der Wahrnehmung von Tonhöhe. Jedes Frequenzband hat hier also im Gegensatz zur DFT einen gleich großen Tonheitsbereich.

Die Filterbänder von Signalen Ambi und Util werden dann in ihrer Schalleistung verglichen und die entsprechend in der Leistung angepassten Util-Filterbänder werden wieder zusammengefügt.

Dieses Verfahren schafft es auch eine deutliche Erhöhung der Richtwirkung zu erziehlen, allerdings sind auch hier die Artefakte vorhanden (Runow 2015b).

4 Implementierung

Ziel ist es ein Plug-in zu entwickeln welches die Doppel-MS Anordnung hinsichtlich ihrer Vielseitigkeit in Winkel und Richtcharakteristik des dekodierten Signals nach Runow (2015b) nutzt. Dabei soll das Hauptaugenmerk auf dem Beamforming liegen. Eine Stereomatrizierung steht nicht im Mittelpunkt. Als Output wird ein Monosignal ausgegeben, da davon ausgegangen wird, dass eine Abnahme der Schallquelle ohne Raum- und Störgeräusche gewünscht ist. Trotzdem ist es natürlich möglich mehrere Instanzen für ein Stereobild zu verwenden, bei dem viele Optionen bezüglich Winkel und Richtcharakteristik vorhanden sind.

Die Störgeräuschreduzierung per Mel-Filterbank wird nicht implementiert, da diese aufgrund der vielen Faltungen viel rechenintensiver ist, als die Störgeräuschreduzierung im spektralen Bereich. Damit ist die Störgeräuschreduzierung im Frequenzbereich für Echtzeitanwendungen vorzuziehen. Dies ist kein schwerwiegender Nachteil, da beide Verfahren ähnliche Ergebnisse erzeugen (Runow 2015b).

Das Plug-in wird als VST realisiert, da diese Schnittstelle von den meisten DAWs unterstützt wird. Um das Plug-in auch für andere Schnittstellen exportieren zu können, wird es im WDL-OL/IPlug Framework erstellt. Wie die meisten Audioanwendungen werden VSTs in der Programmiersprache C++ programmiert.

Für die Fourier Transformation wird FFTW verwendet. FFTW ist eine in C/C++ geschriebene FFT-Library, die für nichtgewerbliche Anwendungen kostenlos ist. Sie wurde von Matteo Frigo und Steven G. Johnson am MIT entwickelt. Diese Library wird statisch in das Programm eingebunden. Eine externe dynamische Bibliothek ist also nich notwendig. Für die Fensterfunktion wird Mathworks Implementierung des Hanning Fensters verwendet.

In diesem Kapitel wird zuerst die Funktionsweise des Codes der wichtigsten zwei Funktionen erklärt. Danach wird auf Besonderheiten des Plug-ins hinsichtlich steuerbarer Parameter, Fensterlänge der FFT und Einbindung in das Hostprogramm eingegangen.

4.1 C++ Konstruktor DoubleMSBeamformer

Im Konstruktor des Plug-ins werden alle benötigten Daten initiiert. Zuerst wird die Größe *winsize* der Fensterung festgelegt und dann ein Hanning Fenster dieser Größe erzeugt. Danach werden Arrays zum Puffern der Ein- und Ausgänge erstellt, wobei der Output-Puffer zirkulär implementiert ist, um asynchrones Einspeichern und Ausle-

sen zu ermöglichen. Dies ist nötig, da alle Samples eines Fensters zusammen gespeichert werden, aber einzeln ausgelesen.

Für jede Fourier Transformation mit FFTW muss ein Inputarray, ein Outputarray und ein fftw_plan Objekt erstellt werden.

Danach werden die Parameter mit Werten (Minimum, Maximum, Standardwert etc.) versehen, die grafischen Elemente geladen und beide miteinander verknüpft. Schließlich wird die Grafikoberfläche aktiviert:

```
AttachGraphics(pGraphics);
```

Diese Zeile wurde im Nachhinein auskommentiert, da einige Hostanwendungen Probleme haben das Plug-in mit der grafischen Oberfläche fehlerfrei zu laden (siehe Kapitel 5 Evaluation). Dies hat zu Folge, dass das Plug-in nicht die eigene, sondern eine vom Host bereitgestellte Bedienoberfläche verwendet.

4.2 C++ Funktion ProcessDoubleReplacing

Die Funktion DoubleMSBeamformer::ProcessDoubleReplacing ist der Kern des Plug-ins. Sie verarbeitet die eintreffenden Signalpakete (double** inputs) und schreibt die verarbeiteten Werte in einen Ausgangsarray (double** outputs). Neben diesen beiden Funktionsparametern wird als drittes noch die Anzahl der eintreffenden Samples (int nFrames) übergeben. Es müssen auch genau nFrames Samples in den outputs-Array geschrieben werden.

Zuerst werden die einzelnen Samples über die Funktion MixDoppelMS() matriziert. Diese beinhaltet die Formel zur Matrizierung von Doppel-MS-Signalen:

```
Return niVo*(A + (1. - A)*cos(alpha))
     + niHi*(A - (1. - A)*cos(alpha))
     + acht*(    (1. - A)*sin(alpha));
```

Über eine Schleife wird dann je für Nutz- und Störsignal ein Array halber Fensterlänge mit matrizierten Samples gefüllt.

Dieses Array wird dann in die zweite Hälfte der FFTW-Inputarrays geschrieben. Dessen erste Hälften werden mit Samples befüllt, die schon im letzten Durchlauf eingelesen wurden. Dieses System wird verwendet, da durch die überlagerten Fensterfunktionen jedes Sample zweimal Transformiert werden muss. Es ist also eine Speicherung aller Samplewerte für je eine Generation nötig. Dabei werden immer „neue“ Samples in Array 2 geschrieben und „alte“ Daten aus Array 1 ausgelesen. Später werden die Zeiger der Arrays vertauscht.

In diesem Schritt werden auch Hanning Fensterung und Zero-Padding durchgeführt. Danach können die beiden Signale in die Frequenzdomäne transformiert werden.

Hier kann der eigentliche Algorithmus für die Subtraktion im Spektralbereich von Runow (2015b) durchgeführt werden. Dabei werden zuerst die redundanten Frequenzen über der Nyquist-Frequenz 0 gesetzt. Danach wird eine Schleife über alle Frequenzbänder durchlaufen. Hier werden aus den komplexen Frequenzen zuerst der Absolutwert der beiden Signale und die Phase des Nutzsignals berechnet. Dann wird eine Fallunterscheidung durchgeführt: nur wenn die Amplitude des Störsignalfrequenzbands kleiner ist als die des Nutzsignalfrequenzbands, werden beide voneinander abgezogen. Dabei wird der Faktor mIntensity miteinberechnet, mit dem bestimmt wird wie stark die Frequenzbänder voneinander abgezogen werden.

Danach kann die Rücktransformation durchgeführt werden. Die synthetisierten Zeitsignale werden überlagert in den Output-Puffer gespeichert.

Dann können die beiden Einlesearrays beider Signale, wie schon erwähnt, getauscht werden, um die „neue“ Hälfte im nächsten Durchgang als „alte“ Hälfte wiederzuverwenden:

```
std::swap(util1, util2);
std::swap(ambi1, ambi2);
```

Als letzten Schritt muss noch der Output-Puffer sampleweise in den Funtionsparameter *double** outputs* geschrieben werden, damit die Host-DAW auf die Samples zugreifen kann.

4.3 Parameter

Die grafische Oberfläche (Abbildung 6) des Plug-ins bietet vier unterschiedliche Parameter: Winkel und Richtcharakteristik des virtuellen Mikrofons, Intensität der Störgeräuschunterdrückung und Richtcharakteristik des Ambi-Signals.

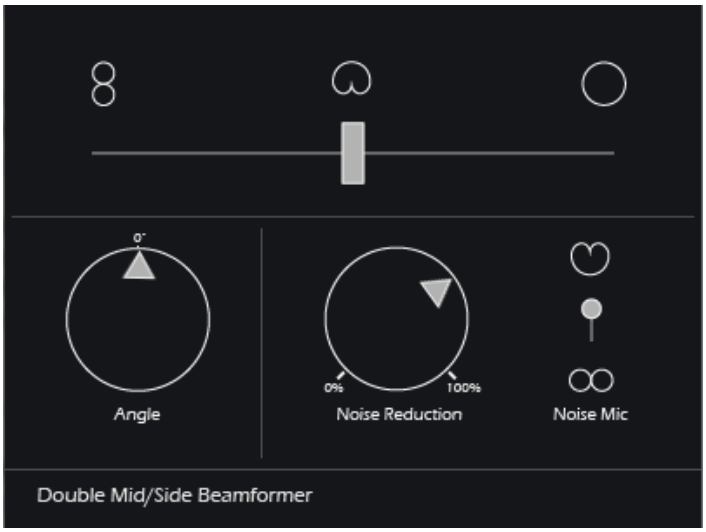

Abbildung 6: Oberfläche des Plug-ins DoubleMSBeamformer (eigene Abbildung)

Der Winkel ist dabei zwischen -180 Grad und 180 Grad frei wählbar und greift direkt auf α zu. Regulär steht er auf 0 Grad, also in Richtung der vorderen Niere.

Die Richtcharakteristik kann zwischen reinem Druckgradient, also Achtercharakteristik über Niere bis zu reinem Druckempfänger, also Kugelcharakterstik frei eingestellt werden. Man greift damit direkt auf den Druckanteil A zu.

Die Intensität der Störgeräuschreduzierung ist in Prozent angegeben und hat beim initiieren des Plug-ins einen Wert von 70%. Dieser Wert hat sich als guter Kompromiss zwischen Störgeräuschreduzierung und zu lauten Artefakten herausgestellt, sollte aber nicht als Standard betrachtet werden.

Die Richtcharakteristik des Ambi-Signals wird über einen Schalter gesteuert, der zwischen Acht und Niere wechselt. Er greift auf Winkel und Druckanteil des Ambi-Signals zu.

Es wird darauf verzichtet Winkel und Druckanteil für das Ambi-Signal frei wählbar zu gestalten, da hier nur bestimmte Kombinationen Sinn ergeben. Wichtig ist dabei, dass in Hauptrichtung des Nutzsignals das Richtungsmaß des Ambi-Signals minimal ist, um keinen Direktschall der Schallquelle aufzunehmen. Dies würde sonst dazu führen, dass ein Anteil des Nutzsignals als Störgeräusch angesehen und dann spektral abgezogen wird.

Diese Forderung erfüllen alle Richtcharakteristiken von Acht (A=0) bis Niere (A=0,5) für einen bestimmten Winkel (bzw. zwei gleichwertige Winkel für A < 0,5). Für die Niere beträgt der Winkel 180 Grad, für die Acht 90 Grad (oder 270 Grad) unterschied zum Winkel des Nutzsignals. Es werden nur diese beiden Optionen implementiert, um die Oberfläche des Plug-ins übersichtlich zu halten und da sie aufgrund des höheren Bündelungsmaßes laut Runow (2015b) geeignete Richtwirkungen sind.

4.4 Fensterlänge

Die Fenstergröße des Plug-ins ist fest auf 1024 Samples gesetzt. Dies bietet einen guten Kompromiss zwischen Signalverzögerung und Klangqualität, andere Größen können aber durchaus erwünscht sein. Dies hängt von den Eigenschaften des Störgeräuschs ab. Generell bringt ein breites Fenster gute spektrale Auflösung; es kann aber nicht so schnell auf zeitliche Änderungen im Störgeräusch reagieren. Das Plug-in bietet nicht die Möglichkeit die Fenstergröße der FFT zu verändern. Dies würde dazu führen, dass das Plug-in bei jedem Wechsel der Fenstergröße neu lädt. Es ist aber möglich das Plug-in mit einer anderen Fenstergröße zu exportieren. Dazu muss in der C++ Datei *DoubleMSBeamformer.cpp* der Wert *winsize* auf die gewünschte Fenstergröße gesetzt werden. Zusätzlich sollte in der Headerdatei *resource.h IPLUG_LATENCY* auch dementsprechend angepasst werden. Es sind Versionen des

Plug-ins mit unterschiedlichen Fenstergrößen auf der beigelegten CD vorhanden um einen einfachen Vergleich der unterschiedlichen Fenstergrößen zu ermöglichen (DoubleMSBeamformer_512.dll, DoubleMSBeamformer_2048.dll).

Nach dem Zero-Padding wird der FFT die doppelte Fensterlänge übergeben. Für das Plug-in mit 1024 Samples Fensterlänge also 2048 Samples.

4.5 Input/Output

Das Plug-in benötigt drei Eingänge und einen Ausgang. Damit das Plug-in aber mit allen DAWs kompatibel ist, wird es als Quadro-Plug-in initiiert. Inputkanal 4 und Outputkanäle 2-4 werden nicht berechnet und übergeben kein Signal. Die Doppel-MS-Kanäle müssen wie folgt an das Plug-in übergeben werden:

$$Niere\ Vorne \rightarrow Input\ 1$$

$$Acht\ Seite \rightarrow Input\ 2$$

$$Niere\ Hinten \rightarrow Input\ 3$$

Output 1 übergibt dann das matrizierte Mono-Signal des virtuellen Mikrofons.

Verschiedene DAWs binden Mehrkanal-Plug-ins unterschiedlich ein. Bei manchen kann das Plug-in direkt auf eine Stereospur geladen und die drei Kanäle der einzelnen Monospuren auf das Plug-in geroutet werden. Ein Beispiel dafür ist *Cockos Reaper* (Abbildung 7). Hier kann man die drei Spuren der Doppel-MS Aufnahme auf den Kanal routen, der das Plug-in enthält. Über das Fenster *Plug-in pin connector* kann dann das Routing der einzelnen Kanäle über eine Matrix mit den Plug-in Eingängen vornehmen.

Abbildung 7: IO-Routing des DoubleMSBeamformers in Reaper 4 (eigene Abbildung)

Aufgrund der Quadro Ein- und Ausgänge wird das Plug-in aber von manchen DAWs als Surround-Plug-in interpretiert. Dort benötigt man einen Quadro-Surround-Bus, bei dem die einzelnen Spuren auf drei Ecken des Quadro-Raums gelegt werden müssen. Ein Beispiel dafür ist Magix Samplitude (Abbildung 8).

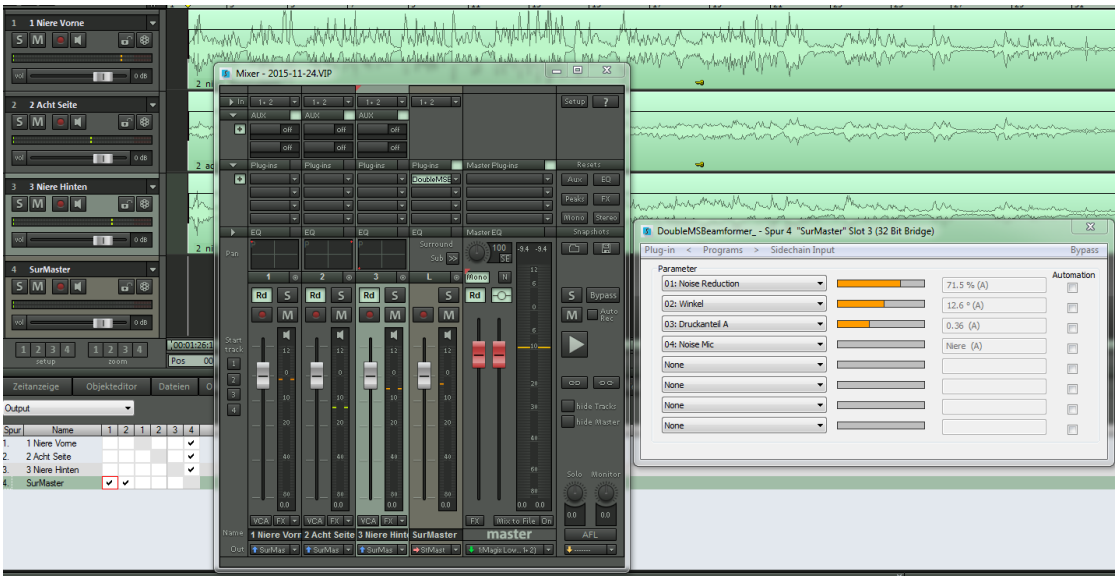

Abbildung 8: IO-Routing des DoubleMSBeamformers in Samplitude Pro X2 (eigene Abbildung)

In diesem Fall müssen die Doppel-MS Kanäle über das zweidimensionale Panning Tool den Eingängen des Plug-ins zugewiesen werden (Abbildung 9):

$$Niere\ Vorne \rightarrow L$$

$$Acht\ Seite \rightarrow R$$

$$Niere\ Hinten \rightarrow LS$$

Abbildung 9: Panning der Doppel-MS Kanäle bei Benutzung eines Quadro-Surround-Kanals in Samplitude Pro X2 (eigene Abbildung)

5 Evaluation

Das Plug-in setzt die Vorgaben wie gewünscht um. Die gesamte Berechnung erfolgt problemlos in Echtzeit. Durch die Fensterung des Signals können die Parameter verändert werden, ohne dass ein hörbares Klicken entsteht. Richtcharakteristik und Winkel des virtuellen Mikrofons können nach Belieben verändert werden. Der Schalter für die Richtcharakteristik des Ambi-Signals erlaubt einen guten A/B-Vergleich der beiden Methoden.

Probleme zeigen sich bei der grafischen Oberfläche, die in manchen Host-Programmen zum Absturz des Programms führt. Wegen dieses schwerwiegenden Fehlers wird in der Hauptversion des Plug-ins auf die grafische Oberfläche verzichtet. Alle DAWs haben eine Standardoberfläche für Plug-ins ohne eigene Grafikelemente. Damit bleibt das Plug-in trotzdem bedienbar. Auf die Verarbeitung des Signals und damit auf den Klang hat dies keine Auswirkungen. Die Version mit Grafikoberfläche (siehe Abbildung 6) ist als DoppelMSBeamformer_GUI.dll auf der beigelegten CD vorhanden. In Cockos Reaper funktioniert auch diese Version, allerdings ist sie nur eingeschränkt zu empfehlen.

Bei Vergleich mit dem offline Matlab-Algorithmus kann qualitativ kein Unterschied zwischen den Aufnahmen festgestellt werden. Bei beiden funktioniert das Beamforming gut und die Störgeräuschreduktion verstärkt zusätzlich die Richtwirkung. Bei sehr halligen Aufnahmen wird der Diffusschallanteil deutlich reduziert. Allerdings treten auch die gleichen Artefakte bei hoher Störgeräuschreduzierung auf.

6 Fazit

Das während dieser Arbeit entstandene VST-Plug-in „DoubleMSBeamformer“ kann den am Anfang der Arbeit gestellten Anforderungen gerecht werden. So ist es damit möglich, aus einem Doppel-MS Signal ein Signal eines virtuellen Mikrofons zu matrizieren. Dabei sind dessen Winkel und Richtcharakteristik erster Ordnung frei wählbar und in Echtzeit ansteuerbar. Die Störgeräuschreduzierung im spektralen Bereich verstärkt die Richtwirkung noch zusätzlich. Im Vergleich mit der offline-Berechnung in Matlab kann qualitativ kein Klangunterschied festgestellt werden, obwohl die gesamte Berechnung in Echtzeit durchgeführt wird.

Die Implementierung als VST-Plug-in sorgt dafür, dass es für viele Hostprogramme kompatibel ist. Eine einfache Nachbearbeitung der Richtwirkung bei Doppel-MS Aufnahmen ist damit also gewährleistet. Die Störgeräuschreduzierung erlaubt durch die bessere Richtwirkung, bei der Aufnahme weiter entfernt von der Schallquelle zu sein, als bei herkömmlichen Doppel-MS Aufnahmen. Dieses Plug-in ist also als Tool für bestimmte Aufnahmesituationen verwendbar, in denen man mit herkömmlicher Mikrofonierung nicht weiter kommt.

Doppel-MS Mikrofonarrays erfassen zusätzlich die Richtungsinformation der Schallquelle. Mit geeigneten Berechnungen könnte in Echtzeit eine Richtungserfassung durchgeführt werden. In Zukunft wäre es also denkbar eine Version zu implementieren, in der die Richtwirkung automatisch der Schallquelle folgt, damit eine manuelle Nachbearbeitung nicht mehr nötig ist. Die Funktionen des hier vorgestellten Plug-ins könnten dann auch für Broadcast- und Liveanwendungen verwendet werden.

Literaturverzeichnis

Dichreiter, M. (2014). Handbuch der Tonstudiotechnik. (8., überarb. und erw. Aufl. ed., Vol. 1). München [u.a.]: Saur.

Görne, T. (2007): Mikrofone in Theorie und Praxis (8. Auflage). Aachen: Elektor-Verlag.

Görne, T. (2014): Tontechnik; Hören, Schallwandler, Impulsantwort und Faltung, digitale Signale, Mehrkanaltechnik, tontechnische Praxis (4. Auflage). München: Hanser-Verlag.

Runow, B. und Curdt, O. (2014): Mikrofonarrays in der professionellen Audioproduktion. Tagungsbericht 28. Tonmeistertagung, (S. 263-269). Köln.

Runow, B. (2015a). Koinzidente Mikrofonarrays. Tübingen. http://www.runow.info/publications/downloads/Runow%20-%20Koinzidente%20Mikrofonarrays.pdf (aufgerufen am 25.11.2015)

Runow, B. (2015b): Zwei Verfahren zur Störgeräuschreduktion mit koinzidenten Mikrofonarrays. Tübingen.

Wittek, H. , Haut, C. und Keinath, D. (2006): Doppel-MS – eine Surround-Aufnahmetechnik unter der Lupe. 24. Tonmeistertagung, Leipzig.

Anhang

Inhalt der CD-ROM

„Entwicklung eines VST-Plug-ins für die Matrizierung eines Doppel-MS Mikrofonarrays“-pdf

VST-Plug-ins: DoubleMSBeamformer.dll und drei alternative Versionen (winsize = 512, winsize = 2048 und GUI)

Auf der CD-ROM befinden sich im Ordner „C++ Code“ die beiden Dateien DoubleMSBeamformer.cpp und DoubleMSBeamformer.h sowie die gesamte WDL-OL Umgebung. Dort befindet sich unter
„C++ Code\wdl-ol\IPlugExamples\DoubleMSBeamformer“ die Visual Studio Projektdatei DoubleMSBeamformer.sln des Plug-ins.

Programmcode

C++ Datei DoubleMSBeamformer.cpp

```
/*
DoubleMSBeamformer by Tobias Gutenkunst
based on an algorithm by Bernfried Runow.
Hochschule der Medien Stuttgart 2015

This plugin processes a double M/S input into an output of a virtual
microphone. This setup allows any angle in the horizontal
plane and any first order polar pattern of the virtual microphone.
Spectral noise reduction is used for better beamforming directivity.
The user interface allows control over four parameters:
  -virtual microphone angle
  -virtual microphone polar pattern
  -noise reduction intensity
  -noise polar pattern

Windowing of 1024 samples is applied before FFT. If you want to
change
the window size, change winsize in line 77 and PLUG_LATENCY in
resource.h.

This file uses code of Oli Larkins MyNewPlugin for WDL-OL
implementation
and Mathworks Hanning function for windowing.
```

```
*/



#include "DoubleMSBeamformer.h"
#pragma warning( suppress : 4101 4129)
#include "IPlug_include_in_plug_src.h"
#include "IControl.h"
#include "resource.h"
#include "include/fftw3.h"
#include <iostream>
using namespace std;

const int kNumPrograms = 1;

enum EParams
{
  kIntensity = 0,
  kWinkel,
  kDruckanteil,
  kNoiseMic,
  kNumParams
};

enum ELayout
{
  kWidth = GUI_WIDTH,
  kHeight = GUI_HEIGHT,

  kIntensityX = 175,
  kIntensityY = 130,

  kWinkelX = 26,
  kWinkelY = 130,

  kDruckanteilX = 50,
  kDruckanteilY = 67,

  kNoiseMicX = 322,
  kNoiseMicY = 158,

  kKnobFrames = 128
};

enum noiseMic
{
   kNiere,
```

```
   kAcht
};

DoubleMSBeamformer::DoubleMSBeamformer(IPlugInstanceInfo
instanceInfo)
  :  IPLUG_CTOR(kNumParams, kNumPrograms, instanceInfo),
mIntensity(1.), mWinkel(1.), mDruckanteil(1.),
mNoiseMicIstAcht(false)
{
  TRACE;

  winsize = 1024;  //set winsize
  SetLatency(winsize);
  hws = winsize / 2;
  han = hanning(winsize, 1); //Hanning window

  //Setup IO Buffers
  util1 = new double[hws];
  ambi1 = new double[hws];
  util2 = new double[hws];
  ambi2 = new double[hws];
  for (int i = 0; i < hws; i++) util1[i] = 0.;
  outputBufferSize = winsize * 2;
  outputBuffer = new double[outputBufferSize];
  for (int i = 0; i < outputBufferSize; i++) outputBuffer[i] = 0.;

  //setup fftw input & output arrays + plan
  fftInUtil = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) *
winsize * 2);
  fftOutUtil = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) *
winsize * 2);
  pUtil = fftw_plan_dft_1d(winsize * 2, fftInUtil, fftOutUtil,
FFTW_FORWARD, FFTW_ESTIMATE);

  fftInAmbi = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) *
winsize * 2);
  fftOutAmbi = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) *
winsize * 2);
  pAmbi = fftw_plan_dft_1d(winsize * 2, fftInAmbi, fftOutAmbi,
FFTW_FORWARD, FFTW_ESTIMATE);

  inverseFftIn = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) *
winsize * 2);
  inverseFftOut = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) *
winsize * 2);
  inverseP = fftw_plan_dft_1d(winsize * 2, inverseFftIn,
inverseFftOut, FFTW_BACKWARD, FFTW_ESTIMATE);
```

```
 //arguments are: name, defaultVal, minVal, maxVal, step, label
  GetParam(kIntensity)->InitDouble("Noise Reduction", 70., 0., 100.0,
0.1, "%");
  GetParam(kWinkel)->InitDouble("Winkel", 0., -180., 180.0, 0.1,
"°");
  GetParam(kDruckanteil)->InitDouble("Druckanteil A", 0.5, 0., 1.0,
0.01, "");
  GetParam(kNoiseMic)->InitEnum("Noise Mic", 0, 2);
  GetParam(kNoiseMic)->SetDisplayText(0, "Niere");
  GetParam(kNoiseMic)->SetDisplayText(1, "Acht");

  IGraphics* pGraphics = MakeGraphics(this, kWidth, kHeight);
  pGraphics->AttachBackground(BACKGROUND_ID, BACKGROUND_FN);

  IBitmap knob_intens = pGraphics->LoadIBitmap(KNOB_INTENS_ID,
KNOB_INTENS_FN, kKnobFrames);
  IBitmap knob_angle = pGraphics->LoadIBitmap(KNOB_ANGLE_ID,
KNOB_ANGLE_FN, kKnobFrames);
  IBitmap slider = pGraphics->LoadIBitmap(SLIDER_ID, SLIDER_FN);
  IBitmap schalter = pGraphics->LoadIBitmap(SWITCH_ID, SWITCH_FN, 2);

  pGraphics->AttachControl(new IKnobMultiControl(this, kIntensityX,
kIntensityY, kIntensity, &knob_intens));
  pGraphics->AttachControl(new IKnobMultiControl(this, kWinkelX,
kWinkelY, kWinkel, &knob_angle));
  pGraphics->AttachControl(new IFaderControl(this, kDruckanteilX,
kDruckanteilY, 300, kDruckanteil, &slider, EDirection::kHorizontal));
  pGraphics->AttachControl(new ISwitchControl(this, kNoiseMicX,
kNoiseMicY, kNoiseMic, &schalter));

  //AttachGraphics(pGraphics); // Die grafische Oberfläche wird nicht
hinzugefügt, da das Plug-in sonst instabil wird

  if (GetAPI() == kAPIVST2) // for VST2 we name individual outputs
  {
    SetInputLabel(0, "Front");
    SetInputLabel(1, "Side");
    SetInputLabel(2, "Back");
    SetInputLabel(3, "-");
    SetOutputLabel(0, "Output");
    SetOutputLabel(1, "-");
    SetOutputLabel(2, "-");
    SetOutputLabel(3, "-");
  }

  //MakePreset("preset 1", ... );
  MakeDefaultPreset((char *) "-", kNumPrograms);
```

```
}

DoubleMSBeamformer::~DoubleMSBeamformer()
{
  delete[] util1;
  delete[] ambi1;
  delete[] util2;
  delete[] ambi2;
  delete[] outputBuffer;

  free (han);

  fftw_destroy_plan(pUtil);
  fftw_free(fftInUtil);
  fftw_free(fftOutUtil);

  fftw_destroy_plan(pAmbi);
  fftw_free(fftInAmbi);
  fftw_free(fftOutAmbi);

  fftw_destroy_plan(inverseP);
  fftw_free(inverseFftIn);
  fftw_free(inverseFftOut);
}

void DoubleMSBeamformer::ProcessDoubleReplacing(double** inputs,
double** outputs, int nFrames)
{
  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* in3 = inputs[2];
  double* out = outputs[0];


  for (int s = 0; s < nFrames; ++s, ++in1, ++in2, ++in3, ++out,
++frameIndex)
  {
    if (frameIndex<hws) //Buffer füllen bis genug Samples für FFT da
sind
    {
      util2[frameIndex] = MixDoppelMS(mDruckanteil, mWinkel, *in1,
*in3, *in2);
      if(mNoiseMicIstAcht) ambi2[frameIndex] = MixDoppelMS(.0,
mWinkel + PI/2, *in1, *in3, *in2);    //ambi-Signal ist eine zu util
um 90° gedrehte Acht
      else          ambi2[frameIndex] = MixDoppelMS(.5, mWinkel +
PI,   *in1, *in3, *in2);    //ambi-Signal ist eine zu util um 180°
gedrehte Niere
```

```
     }else    //Buffer ist Voll. FFT kann starten
     {
       for (int i = 0; i < hws; ++i)
       {
         //von Index 0 bis hws mit 0 auffüllen
         fftInUtil[i][0] = 0.;
         fftInUtil[i][1] = 0.;
         fftInAmbi[i][0] = 0.;
         fftInAmbi[i][1] = 0.;
         //von hws bis winsize mit den "alten" Samples füllen und
Hanning Fensterung durchführen
         fftInUtil[i + hws][0] = util1[i] * han[i];
         fftInUtil[i + hws][1] = 0.;
         fftInAmbi[i + hws][0] = ambi1[i] * han[i];
         fftInAmbi[i + hws][1] = 0.;
         //von winsize bis winsize+hws mit den "neuen" Samples
füllen und Hanning Fensterung durchführen
         fftInUtil[i + winsize][0] = util2[i] * han[i + hws];
         fftInUtil[i + winsize][1] = 0.;
         fftInAmbi[i + winsize][0] = ambi2[i] * han[i + hws];
         fftInAmbi[i + winsize][1] = 0.;
         //bis zum Ende mit 0 auffüllen
         fftInUtil[i + winsize + hws][0] = 0.;
         fftInUtil[i + winsize + hws][1] = 0.;
         fftInAmbi[i + winsize + hws][0] = 0.;
         fftInAmbi[i + winsize + hws][1] = 0.;
       }
       fftw_execute(pUtil);
       fftw_execute(pAmbi);
       fftOutUtil = spektrum_ap(fftOutUtil, winsize);
       fftOutAmbi = spektrum_ap(fftOutAmbi, winsize);
       for (int i = 0; i < winsize * 2; ++i) //für jedes Frequenzband
       {   //bre_vf Algorithmus
         // Einlesen der Amplituden
         double amplitude_virt = sqrt(pow(fftOutUtil[i][0], 2.) +
pow(fftOutUtil[i][1], 2.)); //Absolutwert
         double amplitude_ambi = sqrt(pow(fftOutAmbi[i][0], 2.) +
pow(fftOutAmbi[i][1], 2.));
         // ...und des Winkels
         double winkel_virt = atan2(fftOutUtil[i][1],
fftOutUtil[i][0]);

         // Filter instanzieren
         double filter;
         // Fallunterscheidung
         if (amplitude_ambi < amplitude_virt)filter =
amplitude_ambi;
         else if (amplitude_ambi >= amplitude_virt) {
```

```
            filter = amplitude_virt;
          }
          else filter = 0;
          double amplitude_out = amplitude_virt - filter*mIntensity;
          if (amplitude_out < 0.) amplitude_out = 0.;
          inverseFftIn[i][0] = amplitude_out*cos(winkel_virt);
          inverseFftIn[i][1] = amplitude_out*sin(winkel_virt);
        }
        fftw_execute(inverseP);
        for (int i = 0; i < hws; ++i)
inverseFftOut[i][0] = inverseFftOut[i][0] * han[i];
        for (int i = winsize + hws; i < winsize * 2; ++i)
inverseFftOut[i][0] = inverseFftOut[i][0] * han[i - winsize];
        for (int i = 0; i < winsize * 2; ++i)
        {
          outputBuffer[(outIndexWrite + i) % outputBufferSize] +=
inverseFftOut[i][0];
        }
        outIndexWrite = (outIndexWrite + hws) % outputBufferSize;

        std::swap(util1, util2); //aus neu wird alt... arrays
austauschen um die zweite hälfte wiederzuverwenden
        std::swap(ambi1, ambi2);

        frameIndex = 0;
        util2[frameIndex] = MixDoppelMS(mDruckanteil, mWinkel, *in1,
*in3, *in2);                        // Das erste Sample im neuen Array
wird hier matriziert
        if (mNoiseMicIstAcht)ambi2[frameIndex] = MixDoppelMS(.0,
mWinkel + PI / 2, *in1, *in3, *in2);    //ambi-Signal ist eine zu
util um 90° gedrehte Acht
        else          ambi2[frameIndex] = MixDoppelMS(.5, mWinkel +
PI, *in1, *in3, *in2);    //ambi-Signal ist eine zu util um 180°
gedrehte Niere
     }
     *out = outputBuffer[outIndexRead] / (winsize * 2);  // FFTW gibt
das Signal bei der Rücktransformation mit Faktor winsize*2 zurück
     outputBuffer[outIndexRead] = 0.;
     outIndexRead = (outIndexRead + 1) % outputBufferSize;
//outputBuffer ist zirkulär
   }
}

void DoubleMSBeamformer::Reset()
{
  TRACE;
  IMutexLock lock(this);
}
```

```
void DoubleMSBeamformer::OnParamChange(int paramIdx) //is called when
a parameter control is changed
{
  IMutexLock lock(this);

  switch (paramIdx)
  {
  case kIntensity:
    mIntensity = GetParam(kIntensity)->Value() / 100.;
    break;
  case kWinkel:
    mWinkel = GetParam(kWinkel)->Value() * PI / 180.0;
    break;
  case kDruckanteil:
    mDruckanteil = GetParam(kDruckanteil)->Value();
    break;
  case kNoiseMic:
    mNoiseMicIstAcht = GetParam(kNoiseMic)->Value();
    break;
    default:
      break;
  }
}

double DoubleMSBeamformer::MixDoppelMS(double A, double alpha, double
niVo, double niHi, double acht)
{
  return niVo*(A + (1. - A)*cos(alpha))
      + niHi*(A - (1. - A)*cos(alpha))
      + acht*(    (1. - A)*sin(alpha));
}

fftw_complex *spektrum_ap(fftw_complex *spektrum, const int &winsize)
{
  // Anpassen des Spektrums für spektrum; Nullsetzen des redundanten
Anteils
  // Die vordere Hälfte des Spektrums wird bis auf den ersten Wert
  // (Gleichanteil)in der Amplitude verdoppelt
  for (int i = 1; i < winsize * 2; i++)
  {
    if (i < winsize)
    {
      spektrum[i][0] = spektrum[i][0] * 2.;
      spektrum[i][1] = spektrum[i][1] * 2.;
    }
    else
    {
```

```
       spektrum[i][0] = 0.; //Die zweite Hälfte wird gleich null
gesetzt.
       spektrum[i][1] = 0.;
     }
   }
   return spektrum;
}

/*  function w = hanning(varargin)
%   HANNING   Hanning window.
%   HANNING(N) returns the N-point symmetric Hanning window in a
column
%   vector.  Note that the first and last zero-weighted window
samples
%   are not included.
%
%   HANNING(N,'symmetric') returns the same result as HANNING(N).
%
%   HANNING(N,'periodic') returns the N-point periodic Hanning
window,
%   and includes the first zero-weighted window sample.
%
%   NOTE: Use the HANN function to get a Hanning window which has the
%         first and last zero-weighted samples.ep
itype = 1 --> periodic
itype = 0 --> symmetric
default itype=0 (symmetric)

Copyright 1988-2004 The MathWorks, Inc.
%   $Revision: 1.11.4.3 $  $Date: 2007/12/14 15:05:04 $
*/

double *hanning(int N, short itype)
{
   int half, i, idx, n;
   double *w;

   w = (double*)calloc(N, sizeof(double));
   memset(w, 0, N*sizeof(double));

   if (itype == 1)    //periodic function
     n = N - 1;
   else
     n = N;

   if (n % 2 == 0)
   {
     half = n / 2;
```

```
    for (i = 0; i<half; i++) //CALC_HANNING   Calculates Hanning
window samples.
      w[i] = 0.5 * (1 - cos(2 * PI*(i + 1) / (n + 1)));

    idx = half - 1;
    for (i = half; i<n; i++) {
      w[i] = w[idx];
      idx--;
    }
  }
  else
  {
    half = (n + 1) / 2;
    for (i = 0; i<half; i++) //CALC_HANNING   Calculates Hanning
window samples.
      w[i] = 0.5 * (1 - cos(2 * PI*(i + 1) / (n + 1)));

    idx = half - 2;
    for (i = half; i<n; i++) {
      w[i] = w[idx];
      idx--;
    }
  }

  if (itype == 1)    //periodic function
  {
    for (i = N - 1; i >= 1; i--)
      w[i] = w[i - 1];
    w[0] = 0.0;
  }
  return(w);
}
```

Header-Datei DoubleMSBeamformer.h

```
#ifndef __DOUBLEMSBEAMFORMER__
#define __DOUBLEMSBEAMFORMER__

#pragma warning( suppress : 4101 4129)
#include "IPlug_include_in_plug_hdr.h"
#include "include/fftw3.h"
#include <iostream>
using namespace std;

class DoubleMSBeamformer : public IPlug
{
public:
  DoubleMSBeamformer(IPlugInstanceInfo instanceInfo);
```

```
  ~DoubleMSBeamformer();

  void Reset();
  void OnParamChange(int paramIdx);
  void ProcessDoubleReplacing(double** inputs, double** outputs, int
nFrames);
  double MixDoppelMS(double A, double alpha, double niVo, double
niHi, double acht);

private:
  double mWinkel;
  double mDruckanteil;
  double mIntensity;
  bool mNoiseMicIstAcht;

  int winsize;
  int hws;
  int outputBufferSize;

  int frameIndex=0;
  int outIndexWrite=0;
  int outIndexRead=0;

  double *util1;
  double *ambi1;
  double *util2;
  double *ambi2;
  double *outputBuffer;
  double *han;

  int utilIndex;
  int ambiIndex;

  fftw_complex *fftInUtil;
  fftw_complex *fftOutUtil;
  fftw_plan pUtil;
  fftw_complex *fftInAmbi;
  fftw_complex *fftOutAmbi;
  fftw_plan pAmbi;
  fftw_complex *inverseFftIn;
  fftw_complex *inverseFftOut;
  fftw_plan inverseP;
};
double *hanning(int N, short itype);
fftw_complex *spektrum_ap(fftw_complex *spektrum, const int
&winsize);

#endif
```