UNIVERSITÄT FÜR MUSIK UND DARSTELLENDE KUNST GRAZ - AUSTRIA

UG

Bitte deutlich leserlich ausfüllen!

Deckblatt einer

w i s s e n s c h a f t l i c h e n Bachelorarbeit

Vor- und Familienname	Matrikelnummer
Martin Pauser	9925925
Studienrichtung	**Studienkennzahl**
Elektrotechnik Toningenieur	V 033 213

Thema der Arbeit:

Ambisonic Decoding für reguläre Lautsprecheraufstellungen und Möglichkeiten der Optimierung für die 5.1 Lautsprecheraufstellungen nach dem ITU Standard

Angefertigt in der Lehrveranstaltung: Musikinformatik 1 SE
(Name der Lehrveranstaltung)

Vorgelegt am: ……
(Datum)

Beurteilt durch: ……..
(LeiterIn der Lehrveranstaltung)

Inhalt

Abstract:

Diese Bachelorarbeit befasst sich im Allgemeinen mit Ambisonics und seiner Dekodierung. Im Speziellen wird davon ausgegangen, dass das zu dekodierende Format im B-Format (Ambisonics 1. Ordnung) vorliegt und in einem der handelsüblichen digitalen Audio Workstations einerseits für diametrische Lautsprecheraufstellungen und andererseits für die 5.1 Surround ITU Standard Lautsprecheraufstellung dekodiert werden soll.

Es werden die relevanten Aspekte der Psychoakustik, digitalen Signalverarbeitung und Programmierung berücksichtigt, die nötig sind, um einen Ambisonic Decoder als Plug-In zu realisieren. Es werden die Auswirkungen auf Druck- und Energievektoren, Begriffe wie Shelvingfilter oder Nahfeld-Kompensation erklärt und eine Reihe für unterschiedliche Situationen optimierte Decoder vorgestellt. Zusätzlich werden Möglichkeiten aufgezeigt mittels TABU-Search-Algorithmus geeignete Dekodier-Koeffizienten für die irreguläre 5.1 Lautsprecheraufstellung nach dem ITU-Standard zu finden.

Im Zuge dieser Arbeit wird ein Decoder-Plug-In exemplarisch als VST-Plug-In mit Hilfe des Steinberg Developer Kits 2.4 (SDK 2.4) und Visual Studio Express programmiert, welches unter anderem in Steinberg Nuendo eingesetzt werden kann.

1 Einleitung

1.1 Problemstellung

- Was ist Ambisonics?
- Wie wird decodiert?
- Welche Möglichkeiten der Decoder-Optimierung gibt es?
- Wie implementiere ich einen optimierten Ambisonic-Decoder in der VST-Architektur von Steinberg?

1.2 Ziel der Arbeit

- Untersuchung von Ambisonic Surround Sound-Systemen 1. Ordnung und Möglichkeiten der Optimierung.
- Implementierung eines Ambisonic Decoders innerhalb der Steinberg Virtual Studiotechnology.

1.3 Struktur der Arbeit

Die Arbeit ist in zwei größere Abschnitte geteilt:

1. Recherche und Diskussion
    a. Kapitel 2: Psychoakustik und räumliche Klangwahrnehmung
    b. Kapitel 3: Ambisonic Surround Sound
    c. Kapitel 4: Ambisonic Decoding
2. Implementierung und Test des als VST-Plug-In realisierten Decoders
    a. Digitale Signalverarbeitung
    b. Implementierung und Test

Der erste Teil der Arbeit soll eine fundierte Grundlage bieten, damit ersichtlich ist, welche psychoakustischen Mechanismen der Lokalisation beim Menschen zugrundeliegen.

Im ersten Unterpunkt von Kapitel 3 wird auf die historische Entwicklung von Aufnahme- und Wiedergabeverfahren eingegangen, um zu verstehen, wie es zu der Entwicklung von Ambisonics kam, wieso Ambisonics heute weitgehend unbekannt ist und warum versucht wird, auf für Ambisonics ungeeignete Lautsprecheraufstellungen, wie den ITU 5.1 Standard, zu

decodieren. In diesem Kapitel werden auch die Grundbegriffe von Ambisonics erklärt.

In Kapitel 4 wird erläutert, wie man auf Basis der psychoakustischen Grundlagen aus Kapitel 2 einen Decoder für ein bestimmtes Wiedergabesetup entwirft, wie genau die Decodierung funktioniert, wofür man Filter braucht und warum. Es wird erklärt warum ITU 5.1 eine ungünstige Aufstellung ist und mit welchen Methoden man heute diese Decoder optimiert, um einen realistischen und natürlichen Eindruck von umgebendem Klang zu erzeugen.

Der zweite Teil der Arbeit zeigt, wie man aus den im ersten Teil erlangten Kenntnissen mittels den Programmiersprachen Matlab und C++ Algorithmen entwickelt, digitale Filter umsetzt und diese in der VST-Architektur implementiert und ein Plug-In erstellt, das zu einer Dynamik Linked Library (dll) kompiliert wird. Am Ende des zweiten Abschnittes wird das Testsetup diskutiert.

1.4 Surround Sound und Ambisonics im Überblick

Seit dem Bestehen von Aufnahme und Wiedergabeverfahren in der Audiotechnik wurde versucht akustische Ereignisse immer besser wiederzugeben. Die Entwicklung geht vom ersten Phonographen von Thomas Edison 1877, über Stereophonie, Quadrophonie, diversen Surround Formaten (5.1, 7.1,…), mit einer unterschiedlichen Anzahl an diskreten Wiedergabekanälen, bis hin zu Ambisonics und Wellenfeldsynthese (WFS).

Anfänglich war die treibende Kraft in dieser Entwicklung Harvey Flechter von Bell Laboratories und Alan Dower Blumlein von EMI (Electric & Musical Industries Ltd.), die beide in den 1930er-Jahren direktionale Aufnahme- und Wiedergabeverfahren untersuchten. Alan Dower Blumleins Patent *Improvements in an relation to Sound-transmission, Sound-recording and Sound-reproduction System* aus dem Jahr 1931 und die Versuche von Bell Labs legten den Grundstein für Stereo (wie wir es heute kennen), Wellenfeldsynthese, Ambisonics und (Dolby)Surround Sound.

Durch die Verfügbarkeit günstiger technischer Geräte, digitaler Speichermedien und Rechenleistung sind Surround Sound-Systeme heute weit verbreitet und kommen in vielen anderen Bereichen wie Spielekonsolen, HomeCinema, Rundfunk und HDTV, DVD-Audio und Blue-Ray zur Anwendung.

Surround Sound im Allgemeinen besagt, dass das wiedergegebene Audiomaterial den Hörer/die Hörerin umgibt. Es soll die Illusion entstehen, dass der Hörer/die Hörerin sich mitten im Geschehen befindet und den virtuellen Schallquellen eine bestimmte Richtung und Entfernung zuordnen kann.

Das Wort Ambisonics kommt aus dem Lateinischen und wird zusammen gesetzt aus „ambio“ (lat. umkreisen) und dem englischen Wort „sonic“ (lat. sono = tönen), welches „den Klang betreffend“ bedeutet, und daher das gleiche aussagt wie Surround Sound. In der technischen Umsetzung sind jedoch erhebliche Unterschiede zu Dolby-Surround Sound-Systemen festzustellen.

In der Literatur sind zwei Arten der Verwendung des Wortes Ambisonics zu finden. Die erste ist der Eigenname Ambisonics als Aufnahme- und Wiedergabeverfahren erfunden von Michael Gerzon, die zweite Art ist die adjektivische Verwendung des Eigennamen, wie bei Ambisonic Surround Sound, also Surround Sound im Sinne von Ambisonics.

Ein Beispiel für die adjektivische Verwendung des Wortes, ist der Titel einer Publikation von M. Gerzon selbst *“Multi-system ambisonic decoder”, Wireless World, Vol. 83, no. 1499, pp. 43-47, (Jul & Aug 1977)*

Man kann also im Englischen „Ambisonic System“ sagen oder kurz „Ambisonics“. In englischer Fachliteratur wird „Ambisonic“ in Verbindung mit einem zweiten Nomen immer groß geschrieben, da es von dem Eigennamen Ambisonics abgeleitet wird.

Surround Sound-Systeme können heute in 2 große Kategorien eingeteilt werden (vgl.(1), Seite 49)**:**

1. Kanalgebundene Systeme, die eine spezifische Lautsprecher Aufstellung definieren (vgl. (2)), wie zum Beispiel:
    - Dolby Stereo
    - Dolby Digital AC-3 (5.1)
    - DTS-ES (6.1)
    - Dolby Digital Plus, DTS-HD Master Audio, Dolby TrueHD, SDDS (7.1)
2. Syntheseverfahren, die ein bestimmtes Aufnahme- und Wiedergabeverfahren – encoding, panning, decoding – vorschreiben:
    - Ambisonics
    - Ambiophonics
    - Wellenfeldsynthese

Der große Unterschied zwischen diesen zwei Kategorien ist, dass bei der ersten Kategorie eine bestimmte Anzahl diskreter Audiokanäle geliefert wird, die im optimalen Fall mit dem vorgeschlagenen Lautsprecher Layout wiedergegeben wird(vgl. (3)). Hier gilt: „Je mehr Kanäle desto besser.“

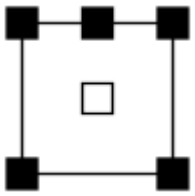

Abbildung 1: Zeichen für Dolby Digital 5.1 auf DVD-Hüllen (Quelle: (4))

Im Fall von Dolby Digital 5.1 sind das drei Front-Kanäle (Left, Center, Right) für Dialog und korrekte Platzierung der on-screen-sounds, zwei

Surround-Kanäle (Left und Right Surround), welche den Hörer/die Hörerin ins Geschehen rücken sollen und einen LFE-Kanal (Low Frequency Effects) mit limitierter Bandbreite für Basseffekte, welche sowohl gehört als auch gefühlt werden können.

Die Priorität der wahrgenommen virtuellen Quellen und deren Lokalistation liegt eindeutig im Gesichtsfeld des Hörers/der Hörerin und bei dem, was sich z.B. auf einer Leinwand abspielt. Wirkung und Zweck der Surround-Kanäle sind lediglich die, dass der Hörer/die Hörerin vom Klang eingehüllt werden soll. Die Wiedergabe ist bei diesem System kanalgebunden. Mehr als fünf verschiedene Kanäle/Audiosignale (und ein Bandbegrenzter) werden hier nicht decodiert. (Die Positionierung virtueller mono Schallquellen erfolgt im Sinne der Intensitätsstereophonie.)

In der zweiten Kategorie liefert eine begrenzte Anzahl an Kanälen eine nach bestimmten mathematischen Vorschriften decodierte beliebig größere Anzahl an unterschiedlichen Ausgangskanälen (Synthese). Jeder einzelne Lautsprecher erhält, abhängig von seiner Position, sein eigenes individuelles Signal. Die Priorität der Lokalisierbarkeit virtueller Quellen ist gleichmäßig um den Hörer/die Hörerin verteilt, das heißt, dass man Quellen, egal aus welcher Richtung sie wahrgenommen werden, gleich gut orten können soll. Das Übertragungsformat und die Decodierung sind hier unabhängig von einander zu sehen.

In dieser Arbeit wird auf Ambisonics aus der zweiten Kategorie von Surround Sound-Systemen eingegangen.

Ambisonics basiert auf den Veröffentlichungen von Michael Anthony Gerzon vom Mathematischen Institut Oxford aus den frühen 70ern und ist eine logische Erweiterung von Blumleins binauralem Wiedergabeverfahren und der Quadrophonie der 60er-Jahre. Anders als zuvor entwickelte Gerzon die *Rational Systematic Desing of Surround Sound Recording and Reproduction Systems* (die später überarbeitete Version hieß General Metatheory of Auditory Localistation(5)), eine Theorie der Theorien, die einen mathematisch wissenschaftlichen Ansatz für die Entwicklung von Surround Sound-Aufnahme- und Wiedergabeverfahren aufbauend auf psychoakustischen Gesichtspunkten ermöglichte.

Bei Ambisonics wird das Schallfeld aus mehreren Kanälen, die in sphärischen Harmonischen (W, X, Y, Z - Ambisonics 1. Ordnung) übertragen werden, an einer zentralen Hörerposition/Hörerinnernposition in der Mitte eines Lautsprecherarrays reproduziert bzw. synthetisiert. In der *„Metatheory of Auditory Localistation“* ging Gerzon von Lautsprecheraufstellungen in gegenüberliegenden Paaren aus (Diametrical Decoder Theorem). Decoding auf 4, 6 oder mehr Lautsprechern in der Horizontalen (2-dimensional) oder mit Höheninformation (3-dimensional, periphon) ist relativ einfach möglich, solange sie gegenüber, paarweise und im gleichen Abstand zum Hörer/zur Hörerin positioniert sind. Für diverse Ambisonic Lautsprecheraufstellungen wird auf Kapitel 3.4 und das Paper „Ambisonic Loudspeaker Arrays“ (6) verwiesen.

Obwohl die Lokalisation bei Ambisonics bei weitem besser ist, als bei üblichen Surround-Systemen, weil dem Hörer/der Hörerin mehrere akustische Stimuli für die Ortung zur Verfügung gestellt werden, ist dieses System wieder in Vergessenheit geraten und eher unbekannt.

Heute jedoch gewinnt Ambisonics zunehmend an Bedeutung, besonders im Bereich der Unterhaltungselektronik und Echtzeitanwendungen bei Computerspielen, da sich der Spieler/die Spielerin in einer virtuellen Umgebung bewegt und Schallereignisse egal aus welcher Richtung kommend (z.B. in einem Ego-Shooter-Spiel) gut orten können sollte. Da es hier möglich ist das Schallfeld aus einzelnen Schallereignissen zu synthetisieren (im Gegensatz zur Aufnahme einer realen Bühnenaufführung), kann in Ambisonics höherer Ordnung (HOA) en- und decodiert werden, was die räumliche Auflösung und damit die Ortung verbessert. In Verbindung mit der Wiedergabe über Kopfhörer, HRTFs und Headtracking-Systemen kann eine äußerst realistische dreidimensionale virtuelle akustische Umgebung geschaffen werden.

2 Psychoakustik und Richtungshören

2.1 Einleitung

Psychoakustik beschäftigt sich unter anderem mit Wirkungen von Schallvorgängen auf das menschliche Erleben und Empfinden und stellt daher die Schnittstelle zwischen physikalischen Gesetzmäßigkeiten und psychologischen Reaktionen dar. Sie beschreibt den Zusammenhang zwischen Schallreizen und menschlicher Empfindung, und spielt daher im Design von Aufnahme- und Wiedergabeverfahren eine zentrale Rolle.

Das Zuordnen von Richtung (Azimut, Elevation) und Entfernung (Abstand *r*) zu einer Schallquelle nennt man Lokalisation, welche auf verschiedenen sich teilweise überlappenden sowie überlagernden Mechanismen basiert. Die Lateralisation bedeutet eine scheinbare Wegbewegung der Schallquelle aus der Mitte des Kopfes hin zu einer Seite bei der Wiedergabe über Kopfhörer. Sie beschreibt die empfundene seitliche Auslenkung dieser Quelle.

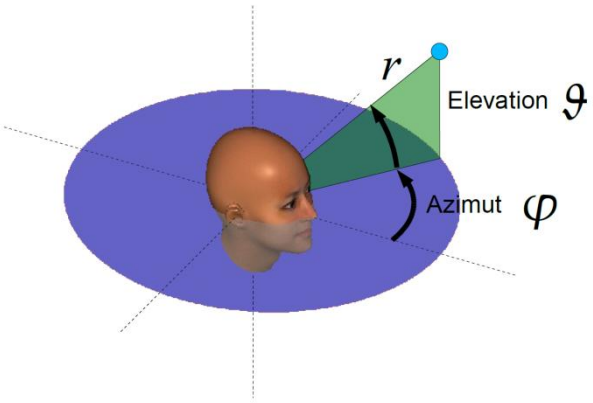

Abbildung 2: Richtungshören,
(Quelle: (7) Seite 14)

Im Folgenden wird erläutert, welche physikalisch messbaren Schallgrößen zu welchen Reizen führen und welche Empfindungen sie im Bezug auf das Richtungshören beim Menschen auslösen.

2.2 Interaural Time Difference (ITD)

Befindet sich eine Schallquelle seitlich der Medianebene (Symmetrieebene des Kopfes), so erreicht das Signal beide Ohren zeitversetzt.

Je weiter die Schallquelle aus der 0°-Richtung ausgelenkt wird, umso mehr Laufzeitunterschied entsteht. Der maximale Laufzeitunterschied an den beiden Ohren beträgt ca. 0,63 ms und nennt sich Interaural Delay.

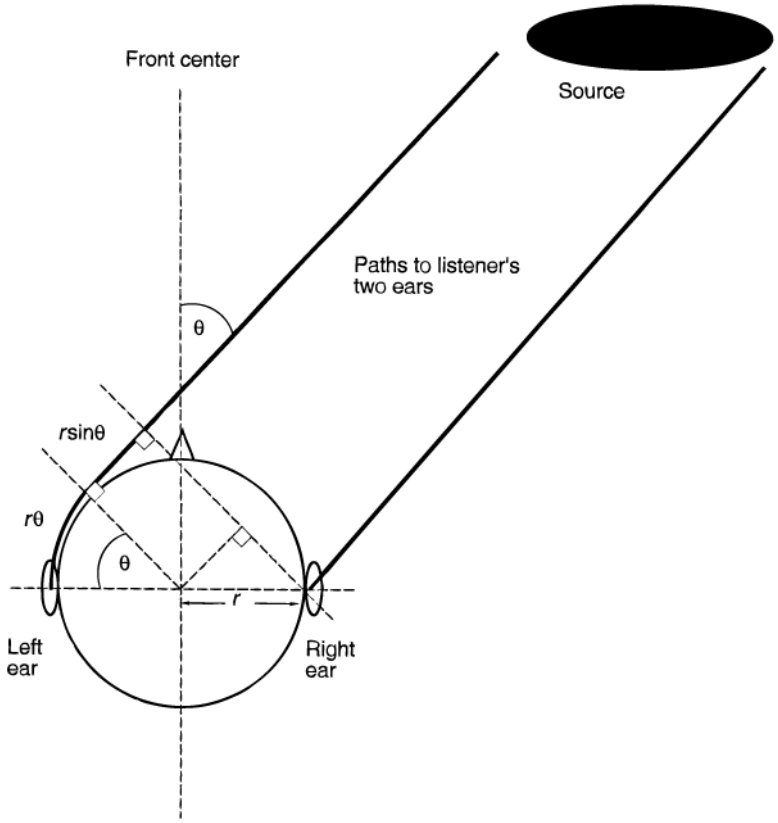

Abbildung 3: Interaural Time Difference (ITD),
(Quelle:(8) Seite 38)

$$\Delta T = \frac{\frac{r}{2}(\theta + \sin\theta)}{c}$$

$\theta[rad]$ … Auslenkung

$c[m/s]$ … Schallgeschwindigkeit

Formel 2.1: ITD

Ob sich eine Schallquelle mit der gleichen Auslenkung vorne oder hinten befindet, kann das Gehör alleine aufgrund der ITD nicht unterschieden. Wenn jedoch der Hörer/die Hörerin seinen/ihren Kopf bewegt, verändert sich dadurch der Laufzeitunterschied bei einer Quelle hinter dem Hörer/der Hörerin gegengleich zu einer Quelle vorne. Die Ortung, basierend auf der ITD, funktioniert gut, solange die Wellenlänge sich in der Größenordnung des Kopfes oder darunter befindet. Hier wird der Schall um den Kopf gebeugt, es findet also keine Abschattung durch ihn statt und die Phase bei zeitverzögerten Sinussignalen ist eindeutig. Sie ist also geeignet für die Lokalistation bei

niedrige Frequenzen oder zur Erkennung der Einhüllenden bei hohen Frequenzen (Signallautstärkeverläufe) geeignet (vgl. (8) Seite 37 ff.).

Der Präzendez-Effekt, auch Gesetz der ersten Wellenfront genannt, ist ebenfalls ein zeitbasierendes System zur Ortung von Schallquellen, besonders bei Schallfeldern mit hohem Diffusanteil. Er besagt, wenn das gleiche Schallsignal zeitverzögert aus unterschiedlichen Richtungen bei einem Hörer/einer Hörerin eintrifft, nimmt dieser/diese nur die Richtung des zuerst eintreffenden Schallsignals wahr. Die um die Anfangszeitlücke (auch als ITDG = Initial Time Delay Gap oder Pre-Dalay bekannt) verzögert eintreffenden Schallsignale werden dann in der Richtung des ersten Signals (der ersten Wellenfront) lokalisiert.

2.3 Interaural Level Difference (ILD)

Die Schallquelle befindet sich hier zwar auch seitlich der Medianebene, doch bei höheren Frequenzen oberhalb von 1600 Hz sind die Abmessungen des Kopfes größer als die Wellenlänge des Schalls und das Signal wird nicht mehr um den Kopf gebeugt. Das Gehör kann die Richtung aufgrund von Phasenmehrdeutigkeit nicht mehr aus der ITD bestimmen. Dafür kommt es zu einer Abschattung des Schalls durch den Kopf, die sich in einem Pegelunterschied an den beiden Ohren bemerkbar macht und ausgewertet werden kann. Schall von rechts besitzt am rechten Ohr einen höheren Pegel als am linken, da der Kopf für ihn ein Hindernis darstellt.

Diese Pegelunterschiede sind stark frequenzabhängig und nehmen mit steigender Frequenz zu. Sie sind ein Maß für die Auslenkung der Quelle aus der 0°-Richtung.

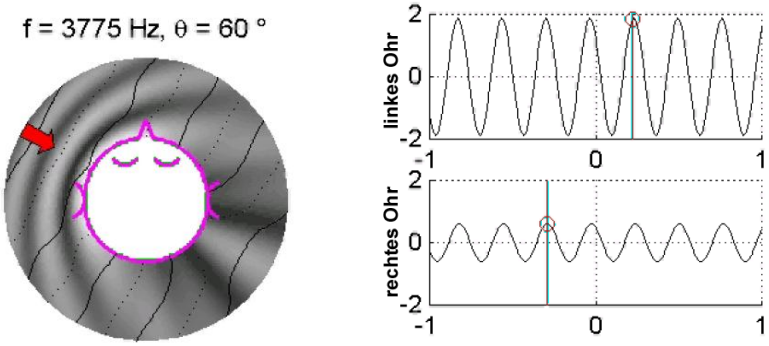

Abbildung 4: ILD,
(Quelle: (7) Seite 18)

2.4 Medianebene und HRFTs

Der „Cone of Confusion“ ist der Kegel, in dessen Richtungen die ITDs gleich sind und dadurch nicht zur Ortung herangezogen werden können. Bei einer in der Medianebene rotierenden Quelle ergeben sich auch bei der ILD kaum Veränderungen.

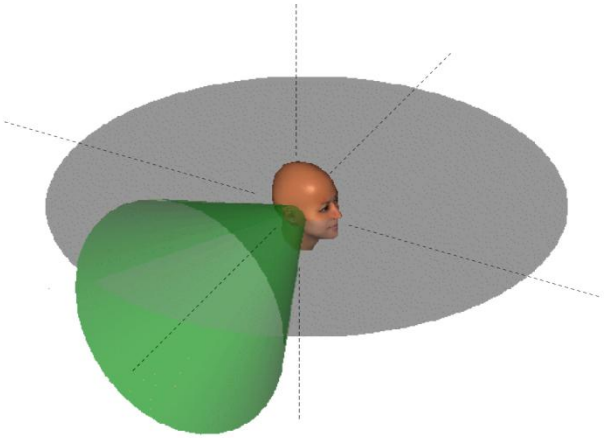

Abbildung 5: cone of confusion,
(Quelle: (7) Seite 33)

Jedoch wirkt die Pinna, das Außenohr des Menschen, als richtungsselektiver Filter. In der Ohrmuschel, im Gehörgang, am Kopf und am Torso werden je nach Schalleinfallsrichtung unterschiedliche Frequenzen durch Reflexion und Interferenz oder Resonanz verstärkt oder abgeschwächt. Das führt dazu, dass jede Richtung seinen eigenen Frequenzgang besitzt, welcher individuell, je nach Form und Größe der eigenen Ohrmuschel verschieden ist, und vom Ohr/Gehirn ausgewertet werden kann.

Das Erkennen der Einfallsrichtung einer Schallwelle basiert auf der Erkennung der eingeprägten richtungsspezifischen Übertragungsfunktionen (Klangmuster), welche als Head Related Transfer Functions (HRTF) bezeichnet werden.

2.5 Zusammenfassung

Bei mehreren, im Bezug auf die Lokalisation einander widersprechenden einzelnen Reizen, wertet das Ohr-Gehirn System die zur Verfügung stehenden Parameter aus und entscheidet sich für die wahrscheinlichste Situation (vgl(8) Seite 44). Es ist also nicht notwendig, bei einem bestimmten

Wiedergabesystem alle Parameter zu berücksichtigen, sondern sich anfangs nur auf die zu konzentrieren, die entscheidungsgebend für die Lokalisation sind, um eine Raumillusion erzeugen. Die für das Ohr entscheidenden Reize werden die Richtung der Lokalisation bestimmen, jedoch wird diese besser ausfallen und öfter korrekt wahrgenommen werden, je mehr einander nicht widersprechende Reize zur Auswertung vorhanden sind.

3 Ambisonic Surround Sound

3.1 Historische Entwicklung

Einer der frühen Versuche von Bell Labs in den 30er Jahren bei der Untersuchung von direktionaler Schallfeldaufnahme und Reproduktion war die „Wall of Sound“. Hier wurden bis zu 80 Mikrofone in einer Reihe vor einem Orchester aufgehängt, welche die jeweils korrespondierenden Audiosignale für ein, in einem anderen Raum befindlichen, gleich aufgestellten, Lausprecherarray lieferten. Die Idee war es, eine von einer Quelle ausgehende Schallwelle in einem Raum aufzunehmen und in einem anderen Raum wiederzugeben, wie aus folgender Abbildung ersichtlich.

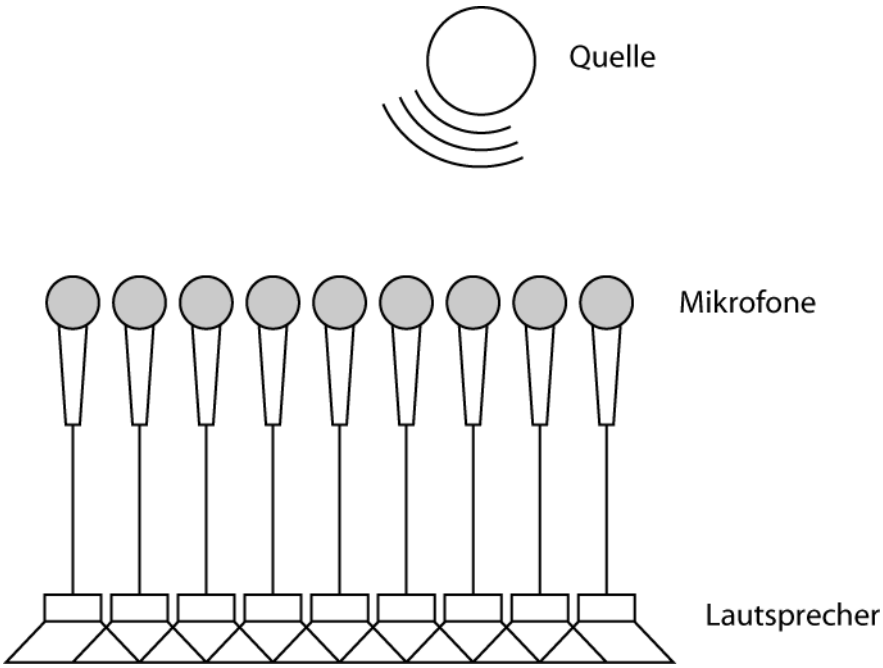

Abbildung 6: "Wall of Sound"

Das Resultat im zweiten Raum war eine gute Abbildung der Schallquelle, die auch bei Bewegungen des Zuhörers/der Zuhörerin nicht wanderte. Ähnlich wie beim Prinzip der Wellenfeldsynthese (WFS) wurde ein Schallfeld in einem anderen Raum synthetisiert. WFS funktioniert nach dem Huygen‘schen Prinzip und besagt, dass jeder Punkt einer Wellenfront als Ausgangspunkt einer neuen Welle, der so genannten *Elementarwelle*, betrachtet werden kann. Die neue Lage der Wellenfront ergibt sich durch Überlagerung (Superposition) sämtlicher Elementarwellen.

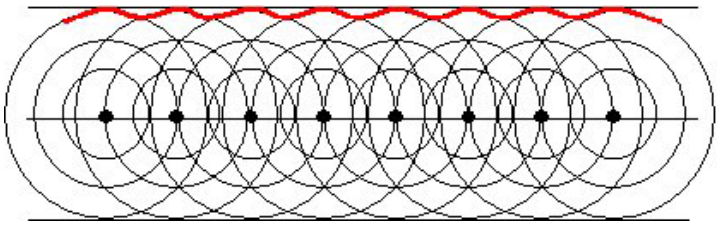

Abbildung 7: Huygensches Prinzip,
(Quelle: http://commons.wikimedia.org/wiki/Huygens%27_principle)

Das gleiche Experiment mit weniger Mikrofonen und Lautsprechern fiel etwas anders aus. Das Problem mit wenigen Lausprechern ist, dass die Schallwelle nicht mehr genau genug rekonstruiert werden kann. Je nach dem, wo sich der Hörer/die Hörerin im zweiten Raum befindet, wird die Position der virtuellen Quelle zu den Lausprechern hingezogen oder kann im Extremfall zwischen ihnen wandern oder als mehrere verschiedene zeitlich verzögerte Quellen wahrgenommen werden. Zu dieser Zeit waren nur omnidirektionale Mikrofone verfügbar, was erklärt, warum Koinzidenzmikrofonieverfahren noch nicht existierten.

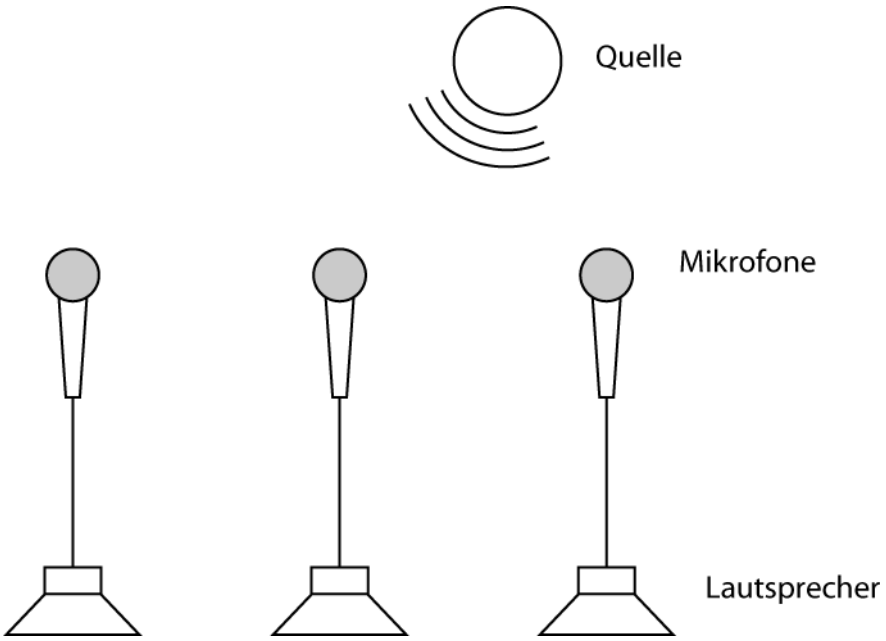

Abbildung 8: "Wall of sound" mit 3 Mikrofonen

Alan Dower Blumlein reichte während seiner Zeit bei EMI (Electric & Musical Industries Ltd.) 1931 sein Patent *Improvements in an relation to Sound-transmission, Sound-recording and Sound-reproduction Systems* ein. Er erkannte die Defizite der Spaced Microphone Technique von Bell und entwickelte eine neue Methode, die er „Binaural Reproduction“ nannte.

Das System beinhaltete zwei omnidirektionale Mikrofone im Abstand der Ohren mit einer Runden Platte dazwischen. Diese Technik war geeignet für Kopfhörerwiedergabe, aber stellte sich als suboptimal für die Wiedergabe über Lautsprecher heraus. Die Phasendifferenz an den Mikrofonen erzeugte und nicht die gewünschte Phasendifferenz an den Ohren des Hörers/die Hörerin bei tiefen Frequenzen. Dies entstand Aufgrund des Übersprechens beider Boxen auf beide Ohren.

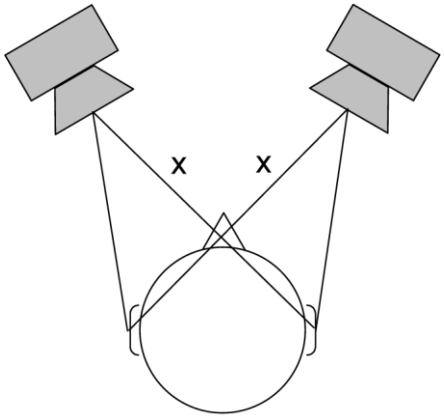

Abbildung 9: Übersprechern der Boxen auf beide Ohren (crosstalk between the ears), bei Abhöre über Stereodreieck, (Quelle: (1) Seite 59)

Er erkannte, dass eine Pegeldifferenz der Signale der Lautsprecher den gewünschten Effekt an den Ohren des Hörers/die Hörerin zur Folge hatte. Seine Erfindung beinhaltete auch einen „Shuffling"-Schaltkreis, der eine Phasendifferenz an den Mikrofonen in eine Pegeldifferenz an den Lautsprechern bei tiefen Frequenzen überführt. Aufgrund der Laufzeitunterschiede und der Schallbeugung um den Kopf kommt es bei einer schräg einfallenden Welle an den Ohren zu einer Phasen-, aber nicht zu einer Pegeldifferenz. Diese Laufzeitdifferenz kann durch eine Pegeldifferenz der Lautsprechersignale erzeugt werden. Bei hohen Frequenzen ergibt sich aufgrund des Kopfschattens eine natürliche Pegeldifferenz.

Blumlein wählte Anfangs diesen umständlichen Weg, da erst ein Jahr später direktionale Bändchenmikrofone mit Achter-Charakteristik verfügbar waren. Mit der Verfügbarkeit dieser Mikrofone verbesserte er seine *Binaural Reproduction Technique.* Er verwendete zwei koinzidente Mikrofone mir 8er-Charakteristik, die um 90° gegeneinander verdreht waren. Dieses Verfahren

war besser mono-kompatibel als das vorige, indem man einfach die beiden Kanäle addierte, woraus sich wieder eine 8er-Charakteristik ergab, die aber nicht wie die beiden ersten 45° links und 45° rechts orientiert waren, sondern nach vorne z.B. auf die Bühne gerichtet war. Der Vorteil lag nicht nur in der Monokompatibilität sondern auch darin, dass die Stereobreite des so aufgenommen Audiomaterials im Nachhinein bei der Wiedergabe durch verschiedene Summierung oder Differenzenbildung verändert werden konnte.

Blumlein erkannte aber auch, dass diese Art der Mikrofonierung und Wiedergabe, die auf das Übersprechen bei tiefen Frequenzen angewiesen war um Laufzeitunterschiede zu erzeugen, auf die Kosten der Lokalisation bei höheren Frequenzen ging, und unterschiedliche Phasendifferenzen bei tiefen und mittleren Frequenzen und somit unterschiedliche Lokalisation, erzeugte (vgl.(9)).

Dieses Problem konnte aber gelöst werden, indem man die Stereobreite wie oben beschrieben durch Veränderung mit Summen und Differenzsignal korrigierte. Diese Technik ist bekannt als *Spatial Equalisation* und wird in Ambisonics Systemen verwendet.

Eine Möglichkeit, künstlich eine Monoquelle im Stereopanorama zu platzieren, wurde auf der Basis von Blumleins Koinzidenzmikrofonie entwickelt, um die Illusion einer echt aufgenommen Situation zu erzeugen. Die Verstärkungen für linken und rechten Kanal um eine Quelle zu „pannen“ werden wie folgt berechnet:

$$LinkeVerstärkung = \sin(\theta + SPos)$$
$$RechteVerstärkung = \cos(\theta + SPos)$$

θ … Quellenposition
SPos … Lautsprecherposition

Formel 3.1: Blumlein-Panning, (Quelle: (1) Seite 65)

Der Parameter SPos wurde verwendet, damit beim Panning der virtuellen Quelle an die Stelle eines Lautsprechers kein Signal mehr aus dem anderen Lautsprecher kommt.

Heute nennt man diese Art der Positionierung im Stereobild „amplitude panning“ oder „pair-wise panning“, die in einer leicht modifizierten Form existiert, da man erkannte, dass der Winkel der Boxen nicht mehr als +/-30 Grad betragen darf und dass die insgesamt abgestrahlte Leistung gleich bleiben muss.

$$LinkeVerstärkung = \sin\left(\theta * \frac{45°}{30°} + 45°\right)$$

Formel 3.2: pair-wise panning

Bei einem θ von 0° ergibt sich eine Verstärkung von ca. -3dB, was einer Halbierung der abgestrahlten Leistung entspricht, die auf beide Lausprecher aufgeteilt wird.

Mitte der 60er-Jahre setzte sich Stereo gegen Mono durch und bald war der Markt auf der Suche nach etwas Neuem, was in man in der Quadrophonie Mitte der 70er-Jahre glaubte gefunden zu haben, welche für ein breites Publikum für die Anwendung zu Hause gedacht war. Man nahm an, dass durch die Erweiterung auf vier Kanäle, die über vier Lautsprecher wieder gegeben wurden, welche quadratisch um den Hörer/die Hörerin positioniert waren, ein Klangfeld erzeugt werden könne. Das war nicht der Fall, da ein Winkel von 90° zwischen den benachbarten Boxen zu groß ist um virtuelle Schallquellen zu erzeugen, die paarweise (pair-wise panning) im Sinne der Intensitätsstereofonie abgemischt waren.

Es entstanden zu dieser Zeit viele nicht miteinander kompatible Medien und Wiedergabesysteme und Versuche abwärtskompatibel zur stereophonen Wiedergabe zu bleiben.

So ein abwärtskompatibles System war auch die Matrix-Quadrophonie (System 4-2-4), bei der mit Hilfe komplexer mathematisch-elektronischer Verfahren unter Ausnutzung von Phasenunterschieden in zwei Stereokanäle codiert und beim Abspielen wieder auf vier decodiert wurde. Herkömmliche stereofone Abspielgeräte konnten beibehalten werden, es wurde lediglich ein Decoder und zugehöriger 4-Kanal-Verstärker mit Lautsprechern benötigt (vgl.(10)).

Obwohl Quadrophonie bald wieder vom Markt verschwand, blieb ein Teil davon bis heute bestehen. Dolby Stereo® (siehe weiter unten) arbeitet mit einem ähnlichen Matrizierungsverfahren.

Ambisonics ist eine Erweiterung und Verbesserung von Blumleins Binaural Reproduction (zumindest konzeptuell) und konnte auch schon über vier Lautsprecher mit sehr guten Ergebnissen wiedergegeben werden.

Gerzon ging davon aus, dass das menschliche Ohr und das Gehirn auf den für die Lokalisation wichtigsten Parameter im jeweiligen Frequenzbereich zurückgreift. Je mehr Parameter dem Ohr/Gehirn zur Verfügung stehen, die konsistent sind, umso besser würde die Lokalisation ausfallen. Aus dieser Theorie entwickelte er ein Lokalisierungsmodell, auf dem das Ambisonics System basiert. Dieses System funktioniert (auch mit nur vier Lautsprechern) besser als herkömmliche Quadrophonie, da (meist) alle Lautsprecher gleichzeitig verwendet werden und zur Lokalisation beitragen. Ambisonics konnte sich am Consumer-Markt jedoch nicht durchsetzen, da die Unterstützung der Musik- und Filmindustrie, die nicht noch ein neues Format wollte, fehlte.

Kurz nachdem Quadrophonie aufkam, trat die Firma Dolby ihren Siegeszug an und revolutionierte die Film- und später auch die Musikindustrie. Dolby entwickelte professionelles Audioequipment und brachte Mitte der 60er-Jahre seine erste Entwicklung Dolby A-type® noise reduction für Musikmagnetbänder auf den Markt. Ende der 60er-Jahre als B-Type Noise Reduction, das Consumer-Format auf den Markt kam, suchte sich die Firma weitere Einsatzfelder für ihre Entwicklung und stieß auf die Filmindustrie. Dolby erkannte, dass ein besseres Mono-Format alleine nicht reichen würde und brachte 1975 Dolby Stereo® auf den Markt. Dolby Stereo war für 35mm Film entwickelt und beinhaltete anstatt des damals üblichen optischen Mono-Tracks zwei Spuren aus denen durch Matrizierung (siehe 4-2-4 Quadrophonie) vier Kanäle (Left, Center, Right, Surround) gewonnen werden konnten - bei weitgehender Stereo- und Mono-Kompatibilität. Außerdem war die Soundqualität aufgrund der implementierten Noise Reduction besser. Das neue Format wurde durch das Dolby Film Program, mit umfassenden Tätigkeiten im Bereich der Beratung von Filmproduzenten, Filmverleihern und Kinobesitzern, und Ausbildung von Technikern, Toningenieuren, verbreitet und erlangte nicht zuletzt durch Filme wie „Star Wars“ und die „Unheimliche Begegnung der dritten Art“ große Akzeptanz und Beliebtheit, und wurde zu *dem* Standard bei optischem analogen Multichannel „Stereo“ Surround Sound-Systemen (vgl. (11)).

Durch die weite Verbreitung von Dolby Stereo, später Dolby Surround Pro Logic (1987) für Heimanwender, verbreitete sich auch die damit verbundene

spezifische Aufstellung der Lautsprecher, welche schließlich für Musikwiedergabe und High Definition Fernsehen von der Internationalen Fernmeldeunion (kurz ITU = International Telecommunication Union) 1993 standardisiert wurde (ITU-R BS.775).

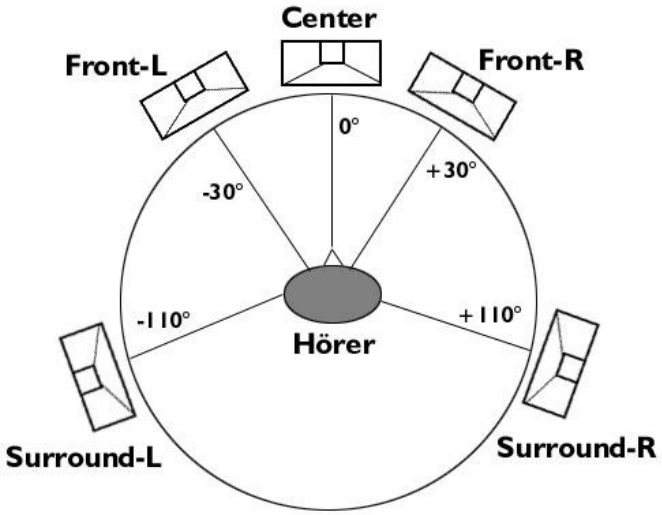

Abbildung 10: ITU 5.1 Standard,
(Quelle: http://de.academic.ru/pictures/dewiki/73/ITU6_Ruhnke.jpg)

Diese oder ähnliche Aufstellungen waren für Ambisonics nicht optimal, da eine einfache Entwicklung eines Ambisonic Decoders für ITU 5.1 nicht mehr möglich war. Gerzon erkannte den Trend des Marktes, reagierte und verfasste zusammen mit Barton das Paper *Ambisonic Decoders for HDTV*, welches bei der Internationalen Konferenz der Audio Engineering Society 1992 in Wien präsentiert wurde. Die Lautsprecheraufstellungen waren damals noch nicht standardisiert, weshalb die in diesem Paper vorgeschlagenen Decoder, heute bekannt als *Vienna Decoder*, nicht für das ITU 5.1 Layout optimiert waren. Außerdem muss man betonen, dass Gerzon nach eigenen Angaben die Lösungen der Gleichungen als „very tedious and messy“, also mühsam und chaotisch bezeichnete. Man weiß heute, dass die Gleichungen damals nicht optimal gelöst wurden (vgl.(1) Seite 59 ff.).

Seit dieser Zeit wird versucht, immer besser optimierte Decoder für die für Ambisonics irreguläre Lautsprecheraufstellung nach dem ITU 5.1 Standard zu entwickeln. Mittels Fitness-Funktionen, welche die Kriterien für eine optimale Decodierung mathematisch formulieren und einem heuristischen Suchalgorithmus, auch bekannt als Tabu-Search-Algoritmus, wird

rechnergestützt die beste Lösung für die Gleichungen gesucht und somit die besten Koeffizienten für einen Decoder ermittelt.

Wenn man sich die Papers der letzten Jahre aus dem Bereich Ambisonics ansieht, erkennt man, dass im Bereich der heuristischen Decoder Optimierung viel Forschungsarbeit geleistet wurde.

Auch wenn bei Ambisonics das Decoding vom Encoding unabhängig ist, also die Anzahl der verwendeten Lautsprecher und deren viele verschiedene Positionierungsmöglichkeiten für die Aufnahme ohne Bedeutung sind und nach Gerzon die Abbildung und Ortung von Schallquellen bei Ambisonics auch bei ungenauer Installation relativ stabil bleibt, können die Boxen nicht vollkommen beliebig im Raum verteilt werden, schon gar nicht bei gleichbleibendem Decoder, was oft missverstanden wird.

3.2 B-Format

Die Erfindung von Ambisonics geht Großteils auf Gerzon zurück und wird oft in einem Atemzug mit der Zerlegung eines Schallfeldes in sphärische Harmonische genannt.

Ähnlich wie bei Blumleins „Binaural Reproduction System“ verwendet auch Ambisonics erster Ordnung mehrere zueinander normal stehende Achtercharakteristiken, aus denen in jede beliebige Richtungen zeigende 8er-Charakteristiken berechnet werden können. Im 2-dimensionalen Fall haben wir dieses System schon bei Blumlein kennengelernt. Im 3-dimensionalen Fall kommt eine weitere Achtercharakteristik entlang der Z-Achse eines kartesischen Koordinatensystems hinzu. Diese drei Kanäle nennt man X, Y und Z gemäß ihrer Ausrichtung im Koordinatensystem. X zeigt nach vorne, Y nach links und Z nach oben. Mit einem weiteren omnidirektionalen Kanal W, welcher als Ambisonics 0.-Ordnung bezeichnet wird, bilden sie gemeinsam das B-Format.

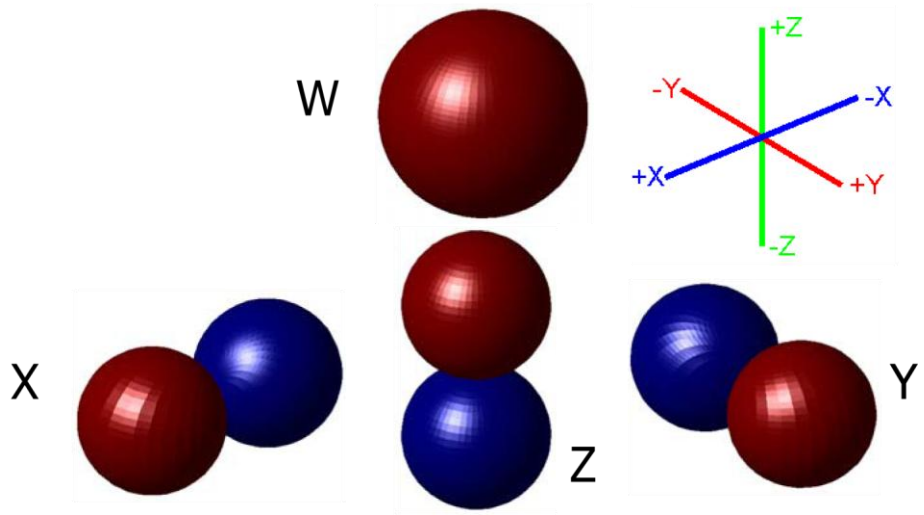

Abbildung 11: 4 Richtcharakteristiken, die zusammen das B-Format bilden (rot In-Phasenanteil), (Quelle: (1) Seite 52)

Aus diesen vier Kanälen kann durch verschieden gewichteter Summation oder Differenzenbildung jede beliebig gerichtete Charakteristik, die zwischen 8er und omnidirektional liegt, errechnet werden. Also auch super-, Hyper-, sub- und normale Nierencharakterristiken sind alle möglich.

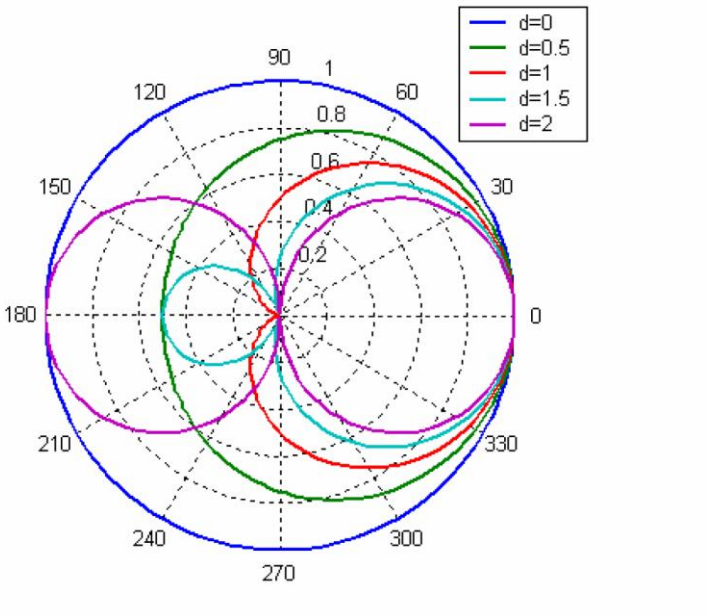

Abbildung 12: Verschiedene Richtcharakteristiken abgeleitet aus 2-dimensionalem Ambisonics 1.Ordnung, (Quelle: (1) Seite 54)

Um eine Monoquelle ins B-Format überzuführen, also zu kodieren, gibt es genau definierte Formeln, die wie folgt lauten:

$$W = mono \cdot \frac{1}{\sqrt{2}}$$
$$X = mono \cdot \cos(\theta) \cdot \cos(\alpha)$$
$$Y = mono \cdot \sin(\theta) \cdot \cos(\alpha)$$
$$Z = mono \cdot \sin(\alpha)$$

$mono$ … das zu encodierende Monosignal
θ …Azimut des Monosignals
α …Elevation des Monosignals

Formel 3.3: B-Format Encodierung, (Quelle: (1) Seite 54)

W wird mit dem Faktor $\frac{1}{\sqrt{2}}$ skaliert, damit die Aussteuerung für diffuse Schallfelder optimal wird.

Ein Signal, welches schon im B-Format encodiert ist, kann leicht um die Achsen des Koordinatensystems gedreht oder in seiner Richtungsdominanz verändert werden:

X-Zoom:

$$W' = W + \frac{1}{\sqrt{2}} \cdot d \cdot W$$
$$X' = X + \sqrt{2} \cdot d \cdot X$$
$$Y' = \sqrt{1 - d^2} \cdot Y$$
$$Z' = \sqrt{1 - d^2} \cdot Z$$

d … Richtungsdominanzparameter [-1,1]
θ … Winkel der Rotation
$W', X', Y' und\ Z'$ sind die neuen veränderten Kanäle

Rotation um die Z-Achse:

$$W' = W$$
$$X' = X \cdot \cos(\theta) + Y \cdot \sin(\theta)$$
$$Y' = Y \cdot \cos(\theta) - X \cdot \sin(\theta)$$
$$Z' = Z$$

Rotation um die Z-Achse:

$$W' = W$$
$$X' = X$$
$$Y' = Y \cdot \cos(\theta) - Z \cdot \sin(\theta)$$
$$Z' = Z \cdot \cos(\theta) + Y \cdot \sin(\theta)$$

Formel 3.4: B-Format, Zoom und Rotation, (Quelle: (1) Seite 65)

Mit diesen Formeln wird klar, was mit der Unabhängigkeit von Encoding (aufnahmeseitig) und Decoding (wiedergabeseitig) gemeint ist. Jeder beliebige Lausprecher kann also mit jeder beliebigen zur Verfügung stehenden Charakteristik betrieben werden, wie aus folgender Formel ersichtlich ist:

$$g_w = \sqrt{2}$$
$$g_x = \cos(\theta) \cdot \cos(\alpha)$$
$$g_y = \sin(\theta) \cdot \cos(\alpha)$$
$$g_z = \sin(\alpha)$$

$$S = 0{,}5 \cdot \left[(2-d)g_w W + d\left(g_x X + g_y Y + g_z Z\right)\right]$$

S … Speakeroutput
θ …Azimut des Lautsprechers
α … Elevation des Lautsprechers
d …Direktivitätsfaktor (0 bis 2)

Formel 3.5: Virtuelle Richtcharakteristik aus B-Format zum Betrieb eines Lautsprechers, (Quelle: (12) Seite 14)

Diese Formeln von Angelo Farina haben eher formalen Charakter, um die Vielseitigkeit des B-Formates darzustellen (vgl. (13) Seite 6 ff.). Sie kürzen den Faktor $\frac{1}{\sqrt{2}}$ aus dem W-Signal wieder heraus, nur zu dem Zweck weitere Berechnungen übersichtlicher zu machen. Klein „g“ mit Index „x“ bis „z“ ist der Richtungsvektor des Lautsprechers aufgespalten in seine kartesischen Komponenten. Die Gewichtungen „g“ mit Index „x“ bis „z“ der Lautsprecher werden wir später beim Testsetup noch näher kennenlernen, sie haben nichts mit der Enkodierung eines Monosignals in B-Format zu tun, obwohl sie sich kaum unterscheiden. Sie werden verwendet, um den Schall aus den einzelnen Lautsprechern in der Mitte des Arrays in ihren Komponenten wieder zu addieren und den resultierenden Gesamtschall zu berechnen. Der Direktivitätsfaktor d bestimmt, mit welcher virtuellen Mikrofoncharakteristik die Box betrieben wird.

Diese Sammlungen an Gleichungen sind nur für einen Decoder mit virtuellen Richtcharakteristiken geeignet und nicht für ein psychoakustisches Decoderdesign, obwohl Bruce Wiggins in seiner PhD Thesis auch Direktivitätsfaktoren für velocity- und energie-optimierte Richtcharakteristiken von 1,15 und 1,33 angibt (vgl. (1) Seite 57).

Diese zwei virtuellen Richtcharakteristiken mit unterschiedlicher Direktivität zum Betrieb eines Lautsprechers sehen wir in folgender Abbildung.

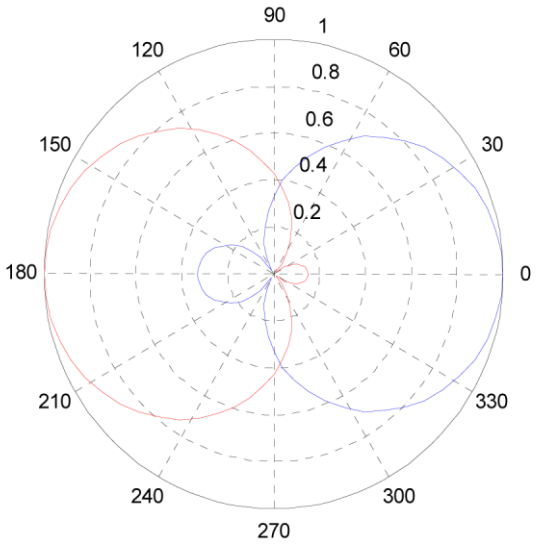

Abbildung 13: Richtcharakteristiken zum Betrieb eines Lautsprechers
d=1,15 (180°) für hohe Frequenzen
d=1,33 (0°) für tiefe Frequenzen
(Quelle: (1) Seite 58)

Die beiden Richtcharakteristiken sind um 180° gegeneinander verdreht, damit sie besser voneinander unterscheidbar sind.

3.3 A-Format

Ambisonics 1. Ordnung kann mit einem Soundfield Mikrofon aufgenommen werden. Dieses Mikrofon besteht aus vier Kapseln mit Sub-Nierencharakteristik, die an jeder Spitze oder Seite eines Tetraeders angeordnet sind.

Abbildung 14: Soundfield Mikrofon, Quelle (1)

Kapsel	Azimut	Elevation
A	45°	35°
B	135°	-35°
C	-45°	-35°
D	-135°	35°

Tabelle 1: Richtungen der Kapseln eines Soundfield Mikrofons

Die Ausgangskanäle (A, B, C, D) diese Mikrofons bilden das A-Format, welches durch Summation und Differenzenbildung, ähnlich wie bei Blumleins Binaural Reproduktion in das B-Format übergeführt werden.

$$W = 0{,}5 \cdot (A + B + C + D)$$
$$X = (A + C) - (B + D)$$
$$Y = (A + B) - (C + D)$$
$$Z = (A + D) - (B + C)$$

Formel 3.6: A-Format zu B-Format

3.4 Ambisonic Loudspeaker Arrays

Das diametrische Decoder Theorem besagt:

Makita und Energie-Vektor-Lokalisation stimmen überein:

1. Wenn alle Lautsprecher den gleichen Abstand zum Zentrum haben.
2. Wenn alle Lausprecher in gegenüberliegenden Paaren angeordnet sind.
3. Wenn die Summe der zwei Signale für gegenüberliegende Paare gleich ist für alle Paare.

Aufbauend auf diesem Theorem kann man diverse, für Ambisonics geeignete, Lautsprecheranordnungen entwerfen.

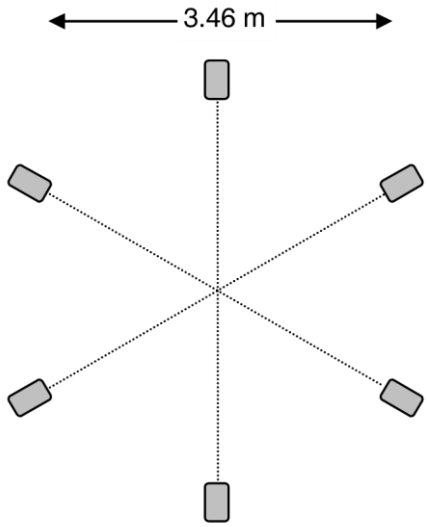

Abbildung 15: Beispiel einer regulären polygonalen 2-dimensionalen Ambisonic Lautsprecheraufstellung, Quelle: (6)

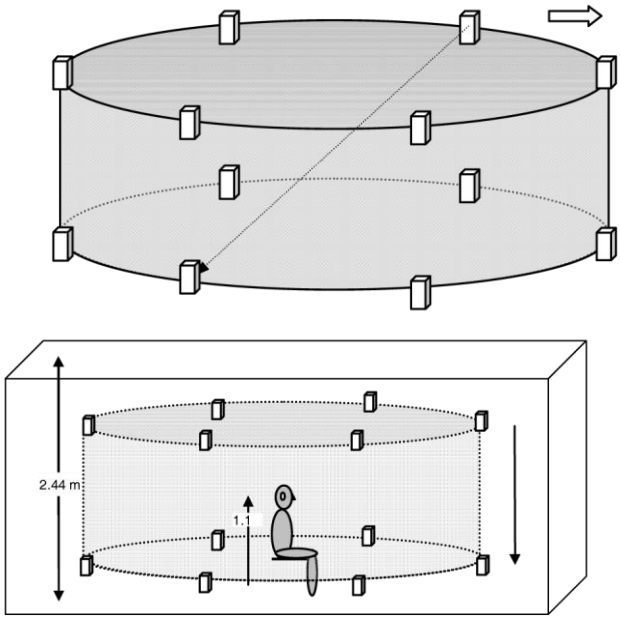

Abbildung 16: Beispiel einer regulären periphonen (3D) Ambisonic Lautsprecheraufstellung mit 12 Boxen, Quelle: (6)

Layouts, bei denen die Boxen in gleichmäßigen Abständen zueinander angeordnet sind, haben eine auch eine gleichmäßige Verteilung der Qualität der Lokalisation, wie zum Beispiel bei einem hexagonalen Arrangement. Wenn ein $1:\sqrt{2}$ Rechteck (in x-Richtung länger) für die Wiedergabe verwendet wird, ist die Lokalisation vorne besser und zu den Seiten etwas schlechter. Die reproduzierten Winkel der Velocity- und Energie-Vektoren stimmen jedoch immer überein.

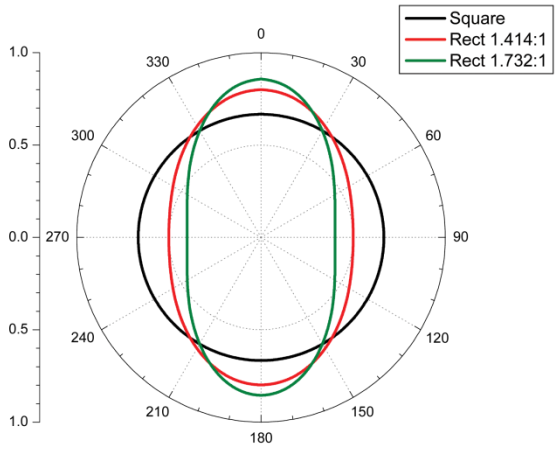

Abbildung 17: Länge des Energie Vektors als Funktion der Quellenrichtung bei einer quadratischen und 2 rechteckigen Lautsprecheraufstellungen, Quelle: (14)

Wenn die Lautsprecher in gegenüberliegenden Paaren angeordnet sind, jedoch nicht im gleichen Abstand zum Zentrum des Arrays, kann durch Verzögerung und Gewichtung (~1/r) im jeweiligen Signalweg ein gleichmäßiger Abstand imitiert werden. Diese Lautsprecheraufstellungen werden so dem Theorem doch noch gerecht.

Verletzt ein Layout den oben genannten zweiten Punkt, wird der Decoderentwurf äußerst kompliziert und es muss ein Kompromiss gefunden werden zwischen Richtungsdominanzen, Gesamtperformance und reproduzierten Energie- und Velocity-Lokalisationswinkeln, um eine optimale Lösung zu finden.

Damit die in den Ambisonic Signalen enthaltenen Richtungsinformationen vollständig in den dekodierten Signalen enthalten sind, wird die Lautsprecheranzahl N so gewählt, dass sie folgende Gleichung mit der Systemordnung M erfüllt (vgl.(15) Seite 6).

$$\text{2D: } N \geq (2M + 1)$$
$$\text{3D: } N \geq (M + 1)^2$$

N … Lautsprecher
M … Systemordnung

Formel 3.7: Mindestanzahl an Lautsprechern bei Ambisonic Decoding, (Quelle: (15) Seite 6)

Bei horizontalem Ambisonics 1. Ordnung ergibt sich also eine theoretische Mindestanzahl von drei Lautsprechern. Ab vier Lautsprechern wird die Decodierung akzeptabel und die Lokalistation verbessert sich je mehr Lautsprecher verwendet werden.

4 Ambisonic Decoding

4.1 Einleitung

Aus der Theorie der Psychoakustik der Richtungswahrnehmung eines Wiedergabeverfahrens leitete Gerzon mathematische Formeln ab, die es ermöglichten, auf einfachem Wege Resultate der Reproduktion zu evaluieren. Die Technologie, die auf dieser Theorie basiert, nannte er Ambisonics.

Aus den 2 Theorien, dass das Gehör bei Frequenzen unter 700 Hz Laufzeitunterschiede (ITD) und bei Frequenzen darüber die Pegeldifferenzen an beiden Ohren (ILD) benutzt, leiten sich die Makita-Theorie und die Energy-Vector-Theorie ab. Die Makita-Lokalisation ist die Richtung, in die der Kopf gedreht werden muss, damit kein Laufzeitunterschied zwischen den Ohren mehr besteht. Das gleiche gilt für die Energy-Vector-Theorie im Bezug auf die Pegeldifferenz.

Nehmen wir an, dass aus vier horizontal im Quadrat aufgestellten Lautsprechern ein Klang kommt, wie in Abbildung 18 gezeigt.

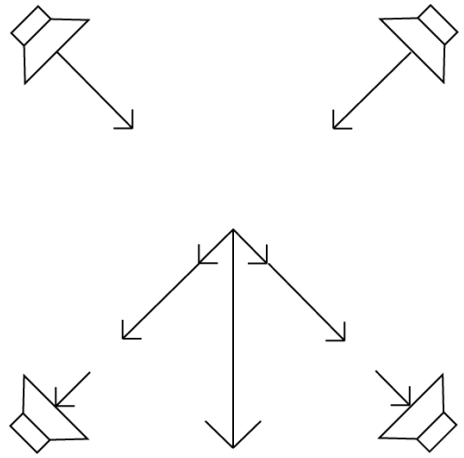

Abbildung 18: Lautsprecher reproduzieren ein Signal

Die Länge des Vektors entspricht der „Menge“ an Schall, die aus der Box kommt. (Bei Anteilen negativer Phase zeigt der Vektor in den Lautsprecher.) In der Mitte des Arrays werden die Vektoren addiert und ergeben den Gesamtschall, dessen Vektor in die Richtung der Makita-Lokalisation zeigt.

Das Bild der Lokalisationen bleibt bei Drehung des Kopfes nur dann stabil, wenn die Größe des resultierenden Vektors gleich der Summe aller Einzelschallereignisse ist. Das ist der Fall, wenn der Schall aus einer einzigen

Quelle/Box kommt. Deshalb wird versucht, das Verhältnis von resultierendem Vektor zu Gesamtschall möglichst nahe 1 zu machen (im besten Fall zu 1), auch wenn der Schall aus verschieden Lautsprechern kommt, wie beim Fall, wenn virtuelle Schallquellen im Raum zwischen den Boxen erzeugt werden sollen. Dieser auf den Gesamtschall bezogene Vektor wird zu einem Maß für die Qualität der Richtungswahrnehmung eines wiedergegeben Signals und somit auch für den Decoder und sollte für gute Decoder in alle Richtungen und für alle Frequenzen möglichst groß gemacht werden (größtmöglicher Wert ist eins) (vgl. (16) Seite 3).

Für tiefe Frequenzen nennen wir diesen auf den Gesamtschall bezogenen Vektor (vector magnitude) r_V. Der Index v steht für velocity, was eigentlich mit Schallschnelle übersetzt wird (physikalisch ist aber der Druckgradient gemeint).

$$\vec{r}_V = \frac{\sum^i G_i \hat{u}_i}{\sum^i G_i}$$

mit G_i als die jeweiligen Lautsprechersignale und $\hat{u}_i$ als Einheitsvektoren in Richtung der jeweiligen Box

Formel 4.1: Velocity-Vektor

Für den „Energy Magnitude“-Vektor für hohe Frequenzen verwenden wir r_E. Dieser Vektor kann nicht zu 1 gemacht werden, da der Schall aus mehreren Quellen kommen muss, sollte aber zu einem maximalen Wert optimiert werden.

$$\vec{r}_E = \frac{\sum^i (G_i)^2 \hat{u}_i}{\sum^i (G_i)^2}$$

Formel 4.2: Energie-Vektor

Die Merkmale eines Ambisonic Decoders oder Systems sind also laut Gerzon:

- Die decodierten Winkel für Vektor $\overrightarrow{r_V}$ und $\overrightarrow{r_E}$ sind gleich und frequenzunabhängig.
- Für tiefe Frequenzen (unter und um 400Hz) hat r_V die Länge 1 für alle wiedergegebenen Richtungen.
- Für mittlere bis hohe Frequenzen (ab 700Hz bis ca.4000Hz) ist die Lände des Vektors $\overrightarrow{r_E}$ maximiert für einen möglichst großen Bereich der 360° Grad der XY-Ebene (oder bei Periphonie natürlich für alle Richtungen in drei Dimensionen)

Jedes Lautsprechersignal wird aus verschiedenen Gewichtungen der Kanäle W, X, Y und Z gewonnen und kann wie folgt mathematisch dargestellt werden (vgl. (16) Seite 11):

$$G_i = W \pm (\alpha_i X + \beta_i Y + \gamma_i Z)$$

G_i …Signal des i-ten Lautsprechers mit $i = 1,2,\dots,n$

$\alpha_i, \beta_i, \gamma_i$ …die Gewichtungen der einzelnen Kanäle

Formel 4.3: Berechnung der Lautsprechersignale

Bei einem horizontalen 4-Lautsprecher-Arrangement, würde die Gleichung in Matrixschreibweise für die decodierten Signale wie folgt aussehen:

$$\begin{pmatrix} G_1 \\ G_2 \\ \vdots \\ G_n \end{pmatrix} = \begin{bmatrix} 1 & \alpha_1 & \beta_1 & \gamma_1 \\ 1 & \alpha_1 & \beta_2 & \gamma_2 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha_n & \beta_n & \gamma_n \end{bmatrix} \cdot \begin{bmatrix} W \\ X \\ Y \\ Z \end{bmatrix}$$

$G_{1,2,\dots,n}$ … gesuchte Lautsprechersignale

Oder in Kurzform:

$\vec{x} = \boldsymbol{D} \cdot \vec{b}$

$\boldsymbol{D}$ … Decodermatrix

$\vec{x}$ … gesuchte Lautsprechersignale

$\vec{b}$ …Sample des im B-Format kodierten Signals

Für ein horizontales Arrangement wird γ zu null und Z scheint daher nicht in den, den Lautsprechern zugeführten Signalen auf und könnte daher weggelassen werden. Was uns zu folgender Gleichung führt:

$$\begin{pmatrix} G_1 \\ G_2 \\ \vdots \\ G_n \end{pmatrix} = \begin{bmatrix} 1 & \alpha_1 & \beta_1 \\ 1 & \alpha_2 & \beta_2 \\ \vdots & \vdots & \vdots \\ 1 & \alpha_n & \beta_n \end{bmatrix} \cdot \begin{bmatrix} W \\ X \\ Y \end{bmatrix}$$

Formel 4.4: Berechnung der Lautsprechersignale in Matrixschreibweise

Die Decoder Matrix D wird durch die Lautsprecheraufstellung der Matrix S (Speakermatrix) bestimmt.

$$\boldsymbol{S} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \\ y_1 & y_2 & \dots & y_n \\ z_1 & z_2 & \dots & z_n \end{bmatrix}$$

Für die einzelnen Lautsprecher können die Koeffizienten wie folgt berechnet werden (vgl.(16)):

$$\begin{pmatrix} \alpha_i \\ \beta_i \\ \gamma_i \end{pmatrix} = \frac{1}{\sqrt{2}} nk \left[\sum_{j=1}^{n} \begin{pmatrix} x_j^2 & x_j y_j & x_j z_j \\ x_j y_j & y_j^2 & y_j z_j \\ x_j z_j & y_j z_j & z_j^2 \end{pmatrix} \right]^{-1} \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$$

Für ein Array mit n $(j = 1,2,\dots,n)$ Lautsprechern mit den

normalisierten Richtungsvektoren $(x_j \quad y_j \quad z_j)$, Index i steht für den i-ten Lautsprecher.
Der Faktor k ist für die jeweilige Optimierung (Energie, Velocity, controlled opposites)

Formel 4.5: Berechnung der Gewichtungsfaktoren für X Y Z, (Quelle: (16) Seite 11)

Mit dem Faktor k kann man das Verhältnis von W zu XYZ einstellen, um unterschiedliche Direktivitäten/Optimierungen für unterschiedliche Gegebenheiten zu erzeugen.

Für ein quadratisches horizontales 4 $(n = 4)$ Lautsprecherarray mit den Richtungsvektoren

$$\boldsymbol{S} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Für die Lautsprecher in der Reihenfolge der Quadranten des Koordinatensystems LeftFront, LeftRear, RightRear, RightFront

und dem Faktor $k = 1$ ergibt sich eine einfache Dekodermatrix D von

$$\boldsymbol{D} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & -1 & 1 & 0 \\ 1 & -1 & -1 & 0 \\ 1 & 1 & -1 & 0 \end{bmatrix}$$

Wir verwenden im Folgenden immer die vollständige Dekodermatrix, da wir Ambisonics im B-Format decodieren, welches immer mit allen Kanälen zur Verfügung steht.

Diese Matrix muss noch mit dem Faktor s skaliert werden, um den im B-Format encodierten Signalvektor $\vec{b}$ in der Mitte des Arrays exakt zu reproduzieren.

$$s = \frac{\sqrt{2}}{n} = 0{,}3536$$

Bei Velocity-optimierung k=1
n …Anzahl der Boxen

$$s \cdot \boldsymbol{D} = \begin{bmatrix} 0{,}3536 & 0{,}3536 & 0{,}3536 & 0 \\ 0{,}3536 & -0{,}3536 & 0{,}3536 & 0 \\ 0{,}3536 & -0{,}3536 & -0{,}3536 & 0 \\ 0{,}3536 & 0{,}3536 & -0{,}3536 & 0 \end{bmatrix}$$

Formel 4.6: Skalierung der Decodermatrix D (gilt für 2D und 3D)

Bei der Matrix ***D*** handelt es sich meist um eine nicht-quadratische Matrix. Die Lösung für die Matrix ***D*** kann auch auf die Bildung einer Pseudoinverse zurückgeführt werden. Die Lautsprechersignale ergeben sich durch Lösung folgender Gleichungen (vgl. (15), Seite 5).

$$\vec{b} = \begin{bmatrix} W \\ X \\ Y \\ Z \end{bmatrix} = S \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \cos(\alpha)\cos(\beta) \\ \sin(\alpha)\cos(\beta) \\ \sin(\beta) \end{bmatrix}$$

S … Monosignal
α … encodierter Azimut
β … encodierte Elevation

$$\vec{x} = [G_1 \quad G_2 \quad \dots \quad G_n]^T$$

$G_{1,2,\dots,n}$ …Lautsprechersignale

$$\vec{b} = \boldsymbol{A} \cdot \vec{x}$$

$$\boldsymbol{A} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \dots & \frac{1}{\sqrt{2}} \\ \cos(\theta_1) & \cos(\theta_2) & \dots & \cos(\theta_n) \\ \sin(\theta_1) & \sin(\theta_2) & \dots & \sin(\theta_n) \\ \sin(\varphi_1) & \sin(\varphi_2) & \dots & \sin(\varphi_n) \end{bmatrix}$$

$\boldsymbol{A}$ … erweiterte Speakermatrix
θ_i …Azimut des i-ten Speakers
φ_i … Elevation des i-ten Speakers
$\vec{b}$ …Sample des B-Format kodierten Signals
s … Skalierung
$\boldsymbol{D}$ … Decodermatrix

$$\vec{x} = \boldsymbol{A}^T \cdot (\boldsymbol{A} \cdot \boldsymbol{A}^T)^{-1} \cdot \vec{b}$$

$$\boldsymbol{A}^T \cdot (\boldsymbol{A} \cdot \boldsymbol{A}^T)^{-1} = s\boldsymbol{D}$$

$$\boldsymbol{D} = \begin{bmatrix} 1 & \alpha_1 & \beta_1 & \gamma_1 \\ 1 & \alpha_2 & \beta_2 & \gamma_2 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha_n & \beta_n & \gamma_n \end{bmatrix}$$

Formel 4.7: Berechnung der skalierten Dekodermatrix A über die Bildung einer Pseudoinversen

Anmerkung: Im Paper von Alois Sontacchi und Robert Höldrich *Schallfeldreproduktion durch ein verbessertes Holophonie – Ambisonic System*(15) wurde die Formel für A anders angegeben. Die Formel für die Matrix A wurde von mir geringfügig verändert. Es handelt sich um den Vektor $\vec{x}$, der transponiert wurde, um die erste Zeile der Matrix A, die mit dem Faktor $\frac{1}{\sqrt{2}}$ bei der Inversenbildung zu einem Velocity Decoder führt und um die dritte Zeile der Matrix A für Periphonie.

Matlabcode zur Berechnung der skalierten Dekodermatrix auf beide Arten ist im Anhang angeführt.

4.2 Reguläre Lautsprecheraufstellung

4.2.1 Velocity optimiert

Für Frequenzen unter und um 400 Hz muss der reproduzierte Vektor r_V zu 1 gemacht werden, damit die Makita-Lokalisation der einer echten Quelle entspricht. Weil die Ambisonic Decoding-Gleichungen so hergeleitet wurden,

dass sie genau dieses Kriterium bewerkstelligen, muss hier auch nicht viel getan werden.

Mit einem $k = 1$ ist der Decoder schon bei gegenüberliegenden paarweise angeordneten Lautsprechern velocity-optimiert.

Nehmen wir an, ein Impuls mit der Amplitude 1 wird in X-Richtung (0°) im B-Format mittels der Formel 3.3 encodiert. Der zugehörige Vektor sieht dann wie folgt aus:

$$\vec{b} = \begin{bmatrix} 0{,}7071 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Mit der velocity-optimierten und skalierten Dekodermatrix für vier Lautsprecher im Quadrat ergeben sich dann die jeweiligen Signale zu

$$\vec{x} = s \cdot \boldsymbol{D} \cdot \vec{b} = \begin{bmatrix} 0{,}3536 & 0{,}3536 & 0{,}3536 & 0 \\ 0{,}3536 & -0{,}3536 & 0{,}3536 & 0 \\ 0{,}3536 & -0{,}3536 & -0{,}3536 & 0 \\ 0{,}3536 & 0{,}3536 & -0{,}3536 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0{,}7071 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0{,}6036 \\ -0{,}1036 \\ -0{,}1036 \\ 0{,}6036 \end{bmatrix}$$

Formel 4.8: Beispiel eines Dekodierungsvorganges für Impuls aus 0°-Richtung bei quadratischer Lautsprecheraufstellung

Um den durch die Boxen wiedergegeben Schall in der Mitte des Arrays wieder zu rekonstruieren, spaltet man ihn in seine kartesischen Komponenten auf und addiert diese Komponentenweise. Dieser Vorgang geht ganz einfach über eine Matrixmultiplikation mit der Speakermatrix (speakerweights).

$$\vec{x}^T \cdot \boldsymbol{S}^T = (0{,}6036 \quad -0{,}1036 \quad -0{,}1036 \quad 0{,}6036) \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \end{bmatrix} = \vec{y}^T$$

$$\vec{y} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Formel 4.9: Rekonstruktion des resultierenden Vektors mittels Matrixmultiplikation

Der rekonstruierte Vektor $\vec{y}$ hat die Länge 1 und die Richtung des encodierten Impulses. Der zugehörige Matlabcode ist im Anhang A zu finden.

4.2.2 Energie optimiert

Bei höheren Frequenzen (700Hz - 4000Hz) muss der Magnitude Vektor r_E möglichst groß gemacht werden. Er kann aber nur je nach Lautsprecher Array unterschiedliche Maximalwerte kleiner 1 annehmen, solange der Schall nicht nur aus einem einzigen Lautsprecher kommt. Bei einem nicht quadratischen rechteckigen Setup wird die Energie-Lokalisation zu den breiteren Seiten, an denen die Lautsprecher weiter auseinander sind, schlechter ausfallen als in die Richtung, wo sie näher zusammen sind (was für r_V nicht gilt).

Um r_E zu maximieren, muss das Verhältnis von W zu X, Y, Z über den Faktor k verändert werden. Der Betrag von Vektor $\overrightarrow{r_E}$ als Funktion von *k* für reguläre 2-D polygonale Aufstellungen mit mindestens vier Lautsprechern ist

$$r_E(k) = \frac{2k}{2k^2 + 1}$$

und hat ein Maximum bei $k = \frac{1}{\sqrt{2}} \approx 0{,}7071 \approx -3{,}1\ dB$

Für reguläre polygonale 3D-Aufstellungen mit mindesten sechs Lautsprechern ergibt sich für

$$r_E(k) = \frac{2k}{3k^2 + 1}$$

ein Maximum von $k = \frac{\sqrt{3}}{3} \approx 0{,}5774 \approx -4{,}77\ dB$.

Decoder mit diesen *k* Werten nennt man Energie-oder „Max-r_E“ Decoder. Damit die Gesamtlautstärke im RMS Sinn gleich bleibt, muss die Absenkung von XYZ und Anhebung von W in einem bestimmten Verhältnis passieren (vgl. (17) Seite 7).

Im 2D-Fall ergibt sich aus

$$W^2 + X^2 + Y^2 = 3 \quad \text{für HF und NF}$$

$$\frac{X}{W} = \frac{Y}{W} = \frac{\sqrt{2}}{2} \quad \text{für HF}$$

für die hohen Frequenzen:

$$W = \sqrt{\frac{3}{2}} \approx +1{,}76\ dB$$

$$X = Y = \sqrt{\frac{3}{4}} \approx -1{,}25\ dB$$

Im 3D-Fall ergibt sich mit

$$W^2 + X^2 + Y^2 + Z^2 = 4 \qquad \text{für HF und NF}$$

$$\frac{X}{W} = \frac{Y}{W} = \frac{\sqrt{3}}{3} \qquad \text{für HF}^1$$

für die hohen Frequenzen:

$$W = \sqrt{2} \approx +3{,}01\ dB$$

$$X = Y = Z = \sqrt{\frac{2}{3}} \approx -1{,}76\ dB$$

Im praktischen Fall heißt das für ein quadratisches Array, dass für ein Signal aus der 0°-Richtung nur Signale aus den vorderen zwei Lautsprechern zur Folge hat.

Falls man sich entscheiden muss zwischen einem velocity- oder energieoptimierten Decoder, ist der Energieoptimierte die bessere Wahl.

4.2.3 Band-Splitting und Shelfing-Filter

Wir sehen an dieser Stelle, dass es nötig ist, für hohe und tiefe Frequenzen eigene Decodermatrizen zu verwenden, um in jedem Frequenzbereich optimal zu decodieren. Der logische Schluss ist, dass beim Dekodieren ein Bandsplitting-Filter eingesetzt werden muss, ähnlich wie bei Crossover Networks bei Mehrweg-Boxensystemen und getrennt einerseits für tiefe und andererseits für mittlere bis hohe Frequenzen dekodiert werden muss. Nach der Decodierung wird das Signal wieder zusammengeführt und über die Lautsprecher ausgegeben.

Bandsplitting-Filter können mit jeweils einem Hoch- und Tiefpass sehr einfach als rekursive IIR-Filter (Infinite Impulse Response-Filter) realisiert werden. Wichtig dabei ist, dass sie die gleiche Phase besitzen, damit es nicht zu Auslöschungen oder Verstärkungen kommt. Diese Kombination aus gleichphasigen Hoch- und Tiefpassfiltern nennt man phase-matched filters (vgl. (18) Seite 12 ff.).

[1] Im Paper (17) Gleichung 12 ist der 3D-Fall falsch angegeben und wurde hier richtig gestellt.

Diese Filter-Kombination besitzt zwar keine lineare Phase, hat aber wenig Verzögerung im Signalweg zur Folge und ist sehr einfach bei zufriedenstellenden Ergebnissen implementierbar.

Eine andere Variante ist die Shelfingfilter-Methode. Hier kann man auch mit nur einem einzigen Decoder (z.B. Velocity-Decoder) und Vorfilterung, d.h. Anhebung oder Absenkung bestimmter Frequenzbereiche, den gewünschten Effekt erzielen. Mit den Shelfing-Filtern wird das gewünschte Verhältnis *k* von W zu XYZ bei hohen Frequenzen eingestellt.

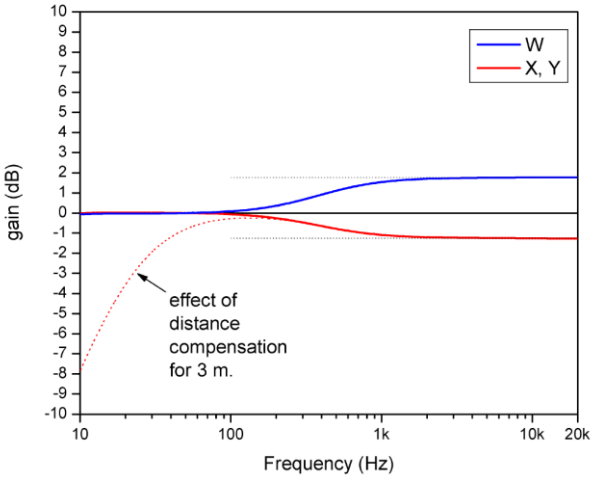

Abbildung 19: Shelfing-Filter Beispiel,
(Quelle: (17) Seite 15)

Diese Vorfilterung wird meist in Form von FFT-Filtern implementiert, die schon ab einer Länge von 256 Samples einen praktisch linearen Phasengang unter 1° und eine Amplitudenveränderung unter 0,2dB besitzen (vgl.(18) Seite 15).

4.2.4 Controlled Opposites

Controlled Opposites bedeutet, dass bei einem Signal, welches in Richtung eines Lautsprechers encodiert wurde, kein gegenphasiges Signal aus dem gegenüberliegenden Lautsprecher kommt. Dieser Decoder wird auch Cardioid-Decoder genannt, da es sich bei der die Box betreibenden virtuellen Richtcharakteristik um eine Nierencharakteristik handelt, die aus der 180°-Richtung kein Signal aufnimmt. Mit dem Faktor *k=0,5* ergibt sich diese Nierencharakteristik für die Dekodermatrix.

Cardioid-Decoder vergrößern den Sweet Spot und sind daher für größeres Publikum gut geeignet. Der Nachteil dabei ist aber, dass die Lokalisation

insgesamt etwas schlechter wird, da die rekonstruierten Druck- und Energie-Magnitude-Vektoren nicht mehr die Länge 1 besitzen.

Normalerweise werden diese Decoder ohne Implementierung von Shelfing-Filtern verwendet.

4.2.5 Nahfeld Kompensation

Oft wird zur Vereinfachung angenommen, dass bei Ambisonics die von den Lautsprechern wiedergegeben Signale ebene Wellen sind. Durch die Lösung der Euler'schen Bewegungsgleichung (hier in x-Richtung)

$$\rho \cdot \frac{\partial v_x}{\partial t} = -\frac{\partial p}{\partial x}$$

$\rho \left[\frac{kg}{m^3}\right]$ …Dichte
v_x …Schallschnelle in x-Richtung
p … Schalldruck

nach t integriert:

$$v_x = -\frac{1}{\rho} \cdot \frac{\partial}{\partial x}\left[\int p\, dt\right]$$

$$\underline{v_x} = -\frac{1}{\rho} \cdot \frac{\partial}{\partial x}\left[\frac{1}{jkc} \cdot \underline{p}\right] = -\frac{1}{\rho} \cdot \frac{(-jk)}{jkc} \cdot \underline{p}$$

$$\underline{v_x} = \frac{\underline{p}}{\rho \cdot c}$$

$k = \frac{2\pi f}{c}$ ….Wellenzahl
c …Schallgeschwindigkeit
f …Frequenz
Mit $\underline{p} = \hat{p} \cdot e^{j\,k(ct-x)}$

Formel 4.10: Eulersche Bewegungsgleichung, Ebene Welle

ergibt sich, dass *v* und *p* bei der Ebenen Welle in Phase sind (vgl. (19) Seite 16 ff.) und es ist keine Nahfeldkompensation nötig. Oft werden auch die Begriffe Distanzkompensation oder NFC(Near Field Correction)-Filter verwendet.

Da Lautsprecher leider keine ebenen Wellen abstrahlen sondern eher Kugelwellen, müssen wir uns die Lösung der Wellengleichung für die Kugelwelle genauer betrachten.

$$v_r = -\frac{1}{\rho} \cdot \frac{\partial}{\partial r}\left[\int \underline{p}\, dt\right]$$

v_r …Schallschnelle in radialer Richtung

mit $\underline{p} = \frac{\hat{p}}{r} \cdot e^{j\,[k(ct-r)+\varphi_1]}$

$$\underline{v_r} = -\frac{1}{\rho} \cdot \frac{\partial}{\partial r}\left[\frac{1}{jkc} \cdot \underline{p}\right] = -\frac{1}{\rho} \cdot \frac{1}{jkc} \cdot \left(-\frac{1}{r} - jk\right) \cdot \underline{p} = \frac{\underline{p}}{\rho \cdot c} \cdot \left(1 + \frac{1}{jkr}\right)$$

$$= \frac{\underline{p}}{Z_0 \cdot kr} \cdot (kr - j) = \left|\underline{v_r}\right| \cdot e^{j\varphi_{pv}}$$

Wobei

$$\varphi_{pv} = \tan^{-1}\left(-\frac{1}{kr}\right)$$

$$\underline{v_r} = \frac{1}{\rho \cdot c} \cdot \frac{\hat{p}}{r} \cdot e^{j[k(ct-r)+\varphi_1]} - j\frac{1}{\rho \cdot c} \cdot \frac{\hat{p}}{k \cdot r^2} \cdot e^{j[k(ct-r)+\varphi_1]}$$

$$v_r = Re\{\underline{v_r}\} = \frac{1}{\rho \cdot c} \cdot \frac{\hat{p}}{r} \cdot \cos[k(ct-r)+\varphi_1] + \frac{1}{\rho \cdot c} \cdot \frac{\hat{p}}{k \cdot r^2} \cdot \sin[k(ct-r)+\varphi_1]$$

Formel 4.11: Euler‘sche Bewegungsgleichung, Kugelwelle

Der erste Term von v_r beschreibt das Fernfeld, in dem sowohl die Schnelle als auch der Schalldruck mit $\frac{1}{r}$ abnehmen. In unendlicher Entfernung ist der Winkel φ_{pv} zwischen Druck und Schnelle theoretisch gleich null, sie sind also in Phase.

Der zweite Term beschreibt das Nahfeld, bei dem die Schallschnelle mit $\frac{1}{kr^2}$ abnimmt. Der Winkel φ_{pv} geht gegen 90°, wobei v_r dem Schalldruck p direkt an der Quelle um 90° nacheilt. Die Größe des Nahfeldes ist von der Wellenzahl *k* (Achtung: k ist hier nicht das Verhältnis von W zu X) und somit von der Frequenz (und der Schallgeschwindigkeit) abhängig (vgl.(19) Seite 46 ff).

Die resultierende komplexe Schallintensität ergibt sich aus dem Produkt von Druck und Schnelle, welche in der Nähe des Lautsprechers einen hohen reaktiven Anteil hat, der nicht abgestrahlt wird. Der Punkt im Raum, an dem Real- und Imaginärteil gleich groß sind (wichtig für Filterentwurf), ist durch Formel 4.12 gegeben.

$$f_g = \frac{c}{2\pi r}$$

Formel 4.12: Grenzfrequenz Nahfeldkompensation

Für den Velocity-Vektor bedeutet dies, dass er bei tiefen Frequenzen größer eins wird, was der Verbreiterung der akustischen Abbildung gleichkommt. Um dem entgegenzuwirken, wird ein Hochpassfilter erster Ordnung für die Kanäle X, Y und Z eingesetzt (vgl.(17) Seite 7).

Die Grenzfrequenz dieses Filters ergibt sich aus Formel 4.12 und liegt bei einem Lautsprecherarray mit 1,5 Metern Durchmesser bei ca. 36 Hz.

Auch bei der Encodierung von Monoquellen sollte laut *Leo Beranek - Acoustic Measurements* eigentlich eine andere Formel verwendet werden.

$$\begin{bmatrix} W \\ X \\ Y \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ D(s) \cdot \cos(\theta) \\ D(s) \cdot \sin(\theta) \end{bmatrix} \cdot Mono$$

mit dem Filter

$$D(s) = \frac{1 + sT}{sT}, T = \frac{r}{c}$$

Formel 4.13: B-Format Encoding laut Leo Beranek – Acoustic Measurements

4.3 Irreguläre Lautsprecheraufstellung

4.3.1 5.0 ITU Standard

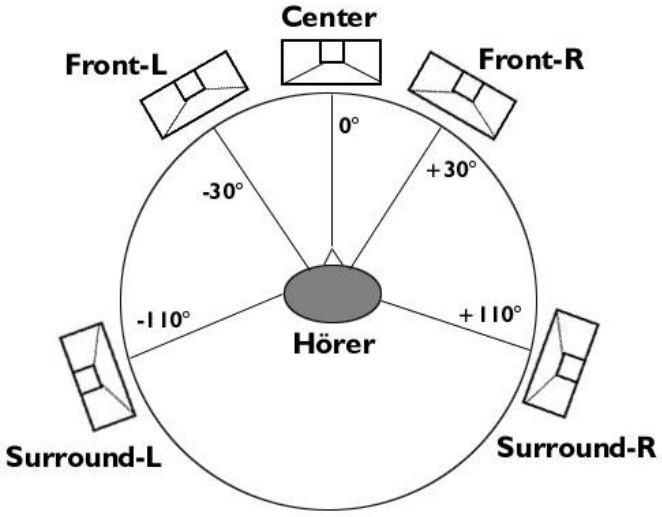

Abbildung 20: ITU 5.1 Surround-Aufstellung

5.1 kanaliger Surround Sound hat sich aufgrund der Erfolge von Dolby Labs sehr weit verbreitet und ist zu *dem* Standard in HDTV-, Homecinema- und Surround-Musikanwendungen geworden. (In großen Kinos werden heutzutage weitaus mehr Kanäle verwendet). Aufstellungen anderer Art mit mehr

Lautsprechern sind im Consumer Elektronik Fachhandel nicht erhältlich und nur in speziellen Fällen vorzufinden (Ambisonics: IEM-Cube und Mumuth Graz, WFS: IOSONO-Kino Ilmenau, Bregenzer und Mörbischer Festspiele).

Um für Konsumenten das optimale Abhören einer Ambisonics Aufnahme zu ermöglichen, ist es aus diesem Grund unerlässlich, einen optimalen Decoder für die ITU 5.1 Aufstellung zu finden.

Das Problem dabei ist, dass es sich bei dieser Aufstellung um eine für Ambisonics irreguläre Aufstellung handelt, die dem Diametric-Decoder-Theorem im zweiten und dritten Punkt widerspricht (Siehe Kapitel 3.4 Ambisonic Loudspeaker Arrays).

Der einfache Entwurf mit den bisher besprochenen Methoden ist nicht mehr möglich. Wenn ein einfacher Cardioid-Decoder verwendet werden würde, wären die Velocity- und Energievektoren, je nach reproduzierter Richtung unterschiedlich lange und hätten nicht die gleiche Richtung.

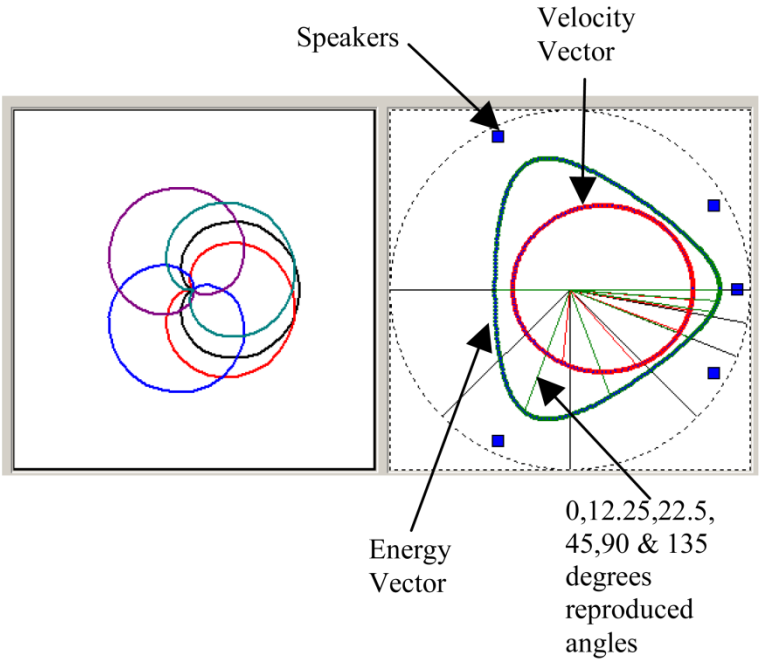

Abbildung 21: ITU 5.1 Cariod Decoder, (Quelle:(1) Seite 138)

Die Bedingungen für einen guten Decoder bleiben jedoch unverändert:

- $|\vec{r}_V|$ und $|\vec{r}_E|$ müssen für alle Richtungen möglichst nahe an 1 sein.
- Die reproduzierten Winkel θ_V für tiefe und θ_E für hohe Frequenzen müssen mit dem encodierten Winkel θ übereinstimmen.
- $P_E = P_V$ und konstant für alle Winkel θ.

Die Gleichungen zur Bestimmung der Decoderqualität lauten:

G_i …Lautsprechersignal
θ_i …Winkel des i-ten Lautsprechers

$$V_x = \sum_{i=1}^{N} G_i \cdot \cos(\theta_i) \,/P_V$$

$$V_x = \sum_{i=1}^{N} G_i \cdot \sin(\theta_i) \,/P_V$$

$$E_x = \sum_{i=1}^{N} (G_i)^2 \cdot \cos(\theta_i) \,/P_E$$

$$E_y = \sum_{i=1}^{N} (G_i)^2 \cdot \sin(\theta_i) \,/P_E$$

$$r_E = \sqrt{E_x^2 + E_y^2} \qquad \theta_E = \tan^{-1}\left(\frac{E_y}{E_x}\right)$$

$$r_V = \sqrt{V_x^2 + V_y^2} \qquad \theta_V = \tan^{-1}\left(\frac{V_y}{V_x}\right)$$

$$P_V = \sum_{i=1}^{N} G_i \qquad P_V = \sum_{i=1}^{N} (G_i)^2$$

Formel 4.14: Formeln zur Messung der Decoderqualität, (Quelle: (1) Seite 141)

Aus dem Umstand heraus, dass die „händische“ Herleitung einer optimalen Lösung bereits für 5 Lautsprecher äußerst schwierig ist, bestand Bedarf an einem System, das automatisch die optimalen Decoder-Koeffizienten berechnet und nach bestimmten Kriterien zwischen „guten“ und „schlechten“ unterscheidet. Die Geschwindigkeit dieses System war von geringer Bedeutung, da es nur einmal für jede Boxenaufstellung einmal die optimale Lösung suchen musste.

4.3.2 Tabu-Search-Algorithmus

Der Tabu-Search-Algorithmus ist ein „intelligenter“ Algorithmus, der innerhalb eines definierten Suchbereiches die beste Lösung sucht. Intelligent deshalb, weil er es vermeidet, eine schon berechnete „schlechte“ Lösung, die in der sogenannten Tabuliste gespeichert wurde, mehr als einmal zu berechnen. Alte Lösungen werden in dieser Liste gespeichert und werden dazu verwendet, die Suche in die richtige Richtung zu treiben. Der Algorithmus wird von sogenannten Fitnessfunktionen, die den Lokalisationsfehler bei einem/einer zentral positionierten Hörer/Hörerin berechnen, in die Richtung eines guten Sets an Decoder-Koeffizienten geleitet (vgl. (1) Seite 142 ff.).

Ausgehend von einem beliebigen Punkt im Suchbereich kann entlang der Koordinatenachsen, die die Decoderparameter repräsentieren, mit einer definierten Schrittgröße in positive oder negative Richtung gegangen werden. Diese beiden Lösungen werden verglichen und die bessere als neuer Startpunkt genommen. So kann mit ein lokales Minimum gefunden werden und mit mehreren Durchläufen ausgehend von verschieden Startpunkten das Beste der lokalen Minima als Lösung genommen werden.

Die Fitnessfunktionen sind das Kernstück dieses Suchalgorithmus, sie messen die Performance des Decoders aufbauend auf Gerzons Metatheorie. Im Zuge der Optimierung werden die einzelnen Sets an Decoderkoeffizienten an die Fitnessfunktionen übergeben und der Lokalisationsfehler (je nach Implementierung RMS oder absolut) in allen Richtungen bestimmt.

Die Fitnessfunktionen (Moorer und Wakefield (20)) lauten:

$$E_{LFAng} = \sum_{i=0}^{n} \left| \theta_i^{Enc} - \theta_i^{V} \right|$$

$$E_{HFAng} = \sum_{i=0}^{n} \left| \theta_i^{Enc} - \theta_i^{E} \right|$$

$$E_{AngMatch} = \sum_{i=0}^{n} \left| \theta_i^{V} - \theta_i^{E} \right|$$

$$E_{LFMag} = \sum_{i=0}^{n} \left| 1 - r_V \right|$$

$$E_{HFMag} = \sum_{i=0}^{n} \left| 1 - r_E \right|$$

$$E_{LFVol} = \frac{1}{n^2} \sum_{i=0}^{n} \sum_{j=0}^{n} \left| 1 - \frac{P_i}{P_j} \right|$$

$$E_{HFVol} = \frac{1}{n^2} \sum_{i=0}^{n} \sum_{j=0}^{n} \left| 1 - \frac{E_i}{E_j} \right|$$

n … Anzahl evaluierten Richtungen für die linke Seite 0-180° ist n=180,

mit Schrittweite 1 für i für die evaluierten Richtungen in Grad

Formel 4.15: Fitnessfunktionen, Quelle: (20)

Es werden die Fehler der reproduzierten Winkel für tiefe und hohe Frequenzen, die Übereinstimmung dieser, die Fehler der Magnitude-Vektoren r_V und r_E und die reproduzierten Lautstärken gemessen.

Diese Funktionen, welche ursprünglich von Bruce Wiggins(1) vorgeschlagen wurden, sind durch Moore und Wakefield verbessert

worden(20), um die Performance des Decoders besser bewerten und dadurch bessere Lösungen für die Decoder-Koeffizienten finden zu können. Die gesamte Fitness ergibt sich aus der Summe der Ergebnisse der einzelnen Fitnessfunktionen.

Der Algorithmus funktioniert nach dem im folgenden Diagramm dargestellten Ablauf:

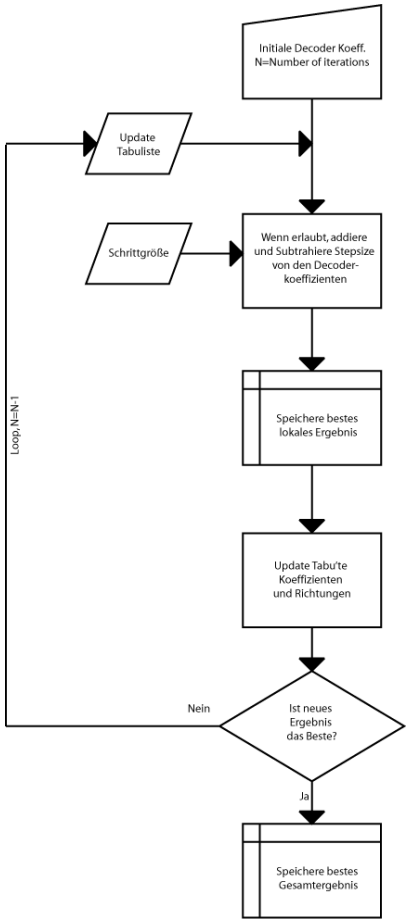

Abbildung 22: Struktur des Tabu Search Alogrithmus

4.3.3 Range Removal, Importance und MAA-Optimierung

Moore und Wakefield haben erkannt, dass einige Fitnessfunktionen (also der Lokalisationsfehler in einem bestimmten Kriterium) einfacher zu minimieren sind als andere und dass sie aufgrund ihres größeren Definitionsbereiches die Suche dominieren. Eine Methode diese Verschiebung auszugleichen, nennt man Range Removal. Diese Methode hat eine gleichmäßige Verteilung des Fehlers auf alle Fitnessfunktionen und somit ein besseres Gesamtergebnis zur Folge. Es werden dadurch besonders die Decoder-Koeffizienten optimiert, die sich auf alle Fitnessfunktionen auswirken (vgl. (21) Seite 3 ff.).

Jedes Ergebnis der Fitnessfunktionen wird als Verhältnis dargestellt zwischen bestem und schlechtestem Wert in vorangegangen Suchvorgängen und befindet sich daher immer im Bereich zwischen 0 und 1. Dadurch wird sichergestellt, dass kein Parameter die Suche dominiert. Das Ergebnis der einzelnen Fitnessfunktionen (objective values) kann also wie folgt als Verhältnis dargestellt werden:

$$FitnessRatio_i = \frac{(objective\_value_i - objective\_value_{min})}{(objective\_value_{max} - objective\_value_{min})}$$

4.16: FitnessRatio, (Quelle: (21) Seite 4)

Diese einzelnen Fitnessverhältnisse können mit einem Gewichtungsfaktor (Importance) versehen werden, um bei der Suche auf bestimmte Kriterien mehr Wert zu legen als auf andere.

Als Vergleich erreichte Bruce Wiggins Decoder (ohne „Range Removal und Importance“) eine Overall-Fitness bzw. einen Gesamtfehler von 1,4148 und Moore/Wakefields Decoder mit „Range Removal und Importance“ 1,3539.

Eine weitere Möglichkeit, einen Decoder an die menschliche Wahrnehmung anzupassen, ist die MAA-Optimierung bzw. das MAA-Optimierungskriteruim. MAA bedeutet Minimum Audible Angle und ist der Winkel, ab dem eine Veränderung der Quellenposition hörbar wird. Dieser Winkel ist im Gesichtsfeld des Hörers/der Hörerin kleiner als zu den Seiten, die Auflösung der menschlichen Lokalisation ist also vorne besser als hinten und zu den Seiten hin am schlechtesten.

Das Prinzip der MAA Optimierung ist die, dass das reproduzierte Schallfeld in drei Bereiche eingeteilt wird. Den jeweiligen Bereichen vorne (0°-59°), Seite (60°-119°) und hinten (120°-180°) werden Gewichtungen zugeordnet, welche indirekt proportional ihrer mittleren MAA sind. Die Fitnessfunktionen bzw. Fitnessratios werden nun mit diesen Gewichtungen bewertet (vgl. (22) Seite 6 ff).

	Gewichtungen
Vorne	1
Seiten	0,3717
Hinten	0,6000

Tabelle 2: MAA-Gewichtungen nach Moorer und Wakefield

Solche für 5.1 ITU Standard MAA-optimierten Decoder verbessern die Performance im vorderen und hinteren Bereich. Dies geschieht natürlich auf Kosten der Wiedergabe im seitlichen Bereich, wo aber der Fehler durch das menschliche Gehör nicht so gut wahrgenommen werden kann.

4.4 Zusammenfassung

Es wurden viele Möglichkeiten der Decodierung und Optimierung vorgestellt, die vorwiegend auf mathematischer Auswertung der reproduzierten Vektoren basieren. Eine weitere Möglichkeit die Bruce Wiggins (1) verwendet ist der Vergleich von mittels HRTF simulierten realen Schallquellen und über Surround Systeme wiedergegeben Quellen. Dieses Verfahren berücksichtigt, dass die wiedergegebenen Signale an zwei verschiedenen Stellen (den beiden Ohren) addiert werden.

Im Bereich der Decoder-Optimierung für irreguläre Aufstellungen, wie dem 5.1 ITU Standard, gibt es noch viel Forschungsmöglichkeit. Ausgedehnte repräsentative Hörtest mit mehreren Decodern gibt es kaum, außerdem könnten die Decoder auf Basis eines umfangreicheren computersimulierten binauralen Lokalisationsmodelles (23) verbessert werden.

5 Digitale Signalverarbeitung

5.1 Einführung

In der heutigen Audiotechnik, vor allem im Bereich der Postproduktion, wird zumindest seit den 90er-Jahren zunehmend auf digitale Bearbeitung am PC oder Mac gesetzt. Erstens aus Kostengründen, da analoges Outboard-Equipment meist teurer ist im Vergleich zu einer digitalen Bearbeitung und zweitens ist die digitale Postproduktion benutzerfreundlicher und es gibt mittlerweile weit mehr Möglichkeiten als in der analogen Domäne.

> *Zeitkontinuierliche Signale werden sind in einem Zeitkontinuum definiert und werden somit durch eine kontinuierliche unabhängige Variable dargestellt. Zeitkontinuierliche Signale werden oft als analoge Signale bezeichnet. Zeitdiskrete Signale sind für diskrete Zeitpunkte definiert, so dass die unabhängige Variable nur diskrete Werte besitzt, d.h. zeitdiskrete Signale werden als Folgen von Zahlen dargestellt. […] Nicht nur die unabhängigen Variablen sind entweder kontinuierlich oder diskret, auch die Signalamplitude kann entweder kontinuierlich oder diskret sein. Digitale Signale sind dadurch gekennzeichnet, dass sowohl die Zeit als auch die Amplitude diskret sind.*
>
> *Signalverarbeitungssysteme lassen sich in der selben Weise wie Signale klassifizieren.*
>
> *Zeitdiskrete Signalverarbeitung, Oppenheim und Schäfer (Seite 35)*

Zeitdiskrete Systeme sind als mathematische Transformation oder als Operator definiert, bei dem die Eingangsfolge $x[n]$ auf die Ausgangsfolge $y[n]$ abgebildet wird.

$$y[n] = T\{x[n]\}$$

Die Ausgangsfolge kann dabei, muss aber nicht, von allen Werten „n“ der Eingangsfolge abhängen. Es können also ein oder mehrere Werte der Folge $x[n]$ zu Bildung der Ausgangsfolge herangezogen werden.

$x[n]$ stellt das zu bearbeitende Audiosignal dar, die Transformation $T\{...\}$ die Bearbeitung (z.B. mittels Plug-In) und $y[n]$ stellt das resultierende zurückgegebene Audiosignal dar.

Eine spezielle, in der Signalverarbeitung besonders wichtige, Klasse zeitdiskreter Systeme sind lineare zeitinvariante Systeme (LTI-Systeme).

Ihre Linearität ist durch das Superpositionsprinzip definiert

$$T\{a\, x_1[n] + \ b\, x_1[n]\ \} = aT\{\ x_1[n]\} + bT\{\ x_2[n]\ \}$$

und ihre Zeitinvarianz durch die Verschiebungsinvarianz. Wenn $x_1[n] = x\big[n - n_0\big]$ muss also $y_1[n] = y\big[n - n_0\big]$ gelten.

Als Konsequenz dieser Eigenschaften ergibt sich, dass LTI-Systeme durch ihre Impulsantwort $h[n]$ vollständig beschrieben werden.

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

Diese Gleichung wird allgemein als Faltungssumme bezeichnet, wobei die Folgen $x[n]$ und $h[n]$ entsprechend obiger Gleichung verknüpft werden, um die Ausgangsfolge $y[n]$ zu bilden. Die zeitdiskrete Faltung wird wie folgt notiert:

$$y[n] = x[n] * h[n]$$

Eine wichtige Unterklasse von LTI-Systemen besteht aus jenen Systemen, die eine lineare Differenzengleichung N-ter Ordnung mit konstanten Koeffizienten folgender Form erfüllt.

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{m=0}^{M} b_m x[n-m]$$

Diese Gleichung kann auch wie folgt dargestellt werden

$$a_0 y[n] + a_1 y[n-1] + \cdots + a_N y[n-N] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-M]$$

Ein einfacher Filter zur gleitenden Mittelwertbildung könnte wie folgt aussehen:

$$h[n] = \begin{cases} \dfrac{1}{M_1 + M_2 + 1}, & -M_1 \leq n \leq M_2 \\ 0, & sonst. \end{cases}$$

Mit $M_1 = 0$ und $M_2 = 2$ ergibt sich

$$h[n] = \frac{1}{3}(\delta[n] + \delta[n-1] + \delta[n-2])$$

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & sonst \end{cases}$$

$$y[n] = \frac{1}{3}x[n] + \frac{1}{3}x[n-1] + \frac{1}{3}x[n-2]$$

Kausale Systeme sind Systeme, bei denen der Wert der Ausgangsfolge $y[n]$ für beliebige Werte n_0 zum Zeitpunkt $n = n_0$ nur von vorangegangenen Werten $n \geq n_0$ der Eingangsfolge abhängt. Zeitdiskrete Echtzeit-Audiosignalverarbeitungs-Systeme sind kausale Systeme.

Linearere Differenzengleichungen mit konstanten Koeffizienten können durch Blockschaltbilder grafisch dargestellt werden.

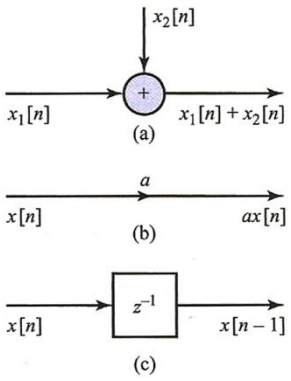

Abbildung 23: Blockschaltbild-Symbole, (Quelle: (24), Seite 425)

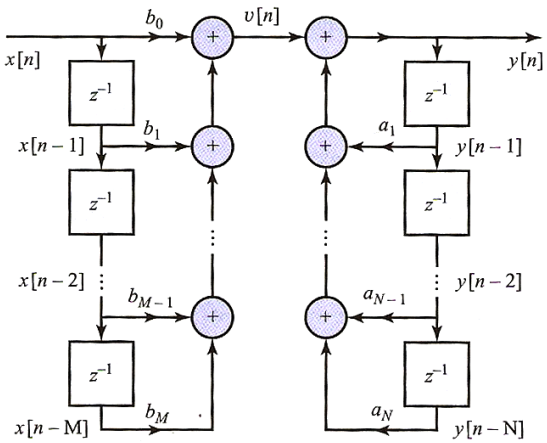

Abbildung 24: Darstellung einer allgemeinen Differenzengleichung N-ter Ordnung mit konstanten Koeffizienten als Blockschaltbild, (Quelle: (24) Seite 428)

Filter, bei denen nur die linke Seite vorhanden ist, also N=0 und M einen endlichen Wert besitzt, nennt man FIR-Filter (Finite Impulse Response-Filter) da ihre Impulsantwort $h[n]$ endliche Länge besitzt.

Filter, bei denen $N > 0$ sind rekursive Filter, die im Allgemeinen eine unendliche Impulsantwort besitzen, weshalb sie IIR(Infinite Impulse Response)-Filter genannt werden.

Digitale Biquad-Filter sind rekursive Filter zweiter Ordnung mit jeweils zwei Polen und Nullstellen. „Biquad“ ist die Abkürzung für „biquadratic“, was bedeutet, dass die Übertragungsfunktion in der Z-Domäne aus dem Verhältnis zweier quadratischer Polynome besteht und daher folgende Form besitzt:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

Rekursive Filter höherer Ordnung können aufgrund ihrer Empfindlichkeit für die Quantisierung der Koeffizienten schnell instabil werden. Deshalb werden sie oft aus Kaskaden seriell geschalteter Biquads realisiert. Bei parallel geschalteten Biquads kann die Veränderung eines Koeffizienten sich auf das Verhalten des gesamten Filters auswirken.

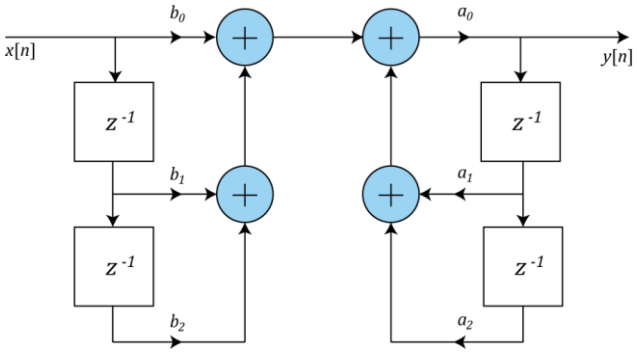

Abbildung 25: Blockschaltbild eines digitalen Biquad-Filters in Direktform 1

5.2 Entwurf frequenzselektiver Filter mittels Bilineartransformation

Tiefpass- oder Hochpass-Filter besitzen ihren Durchlassbereich im unteren bzw. oberen Frequenzbereich.

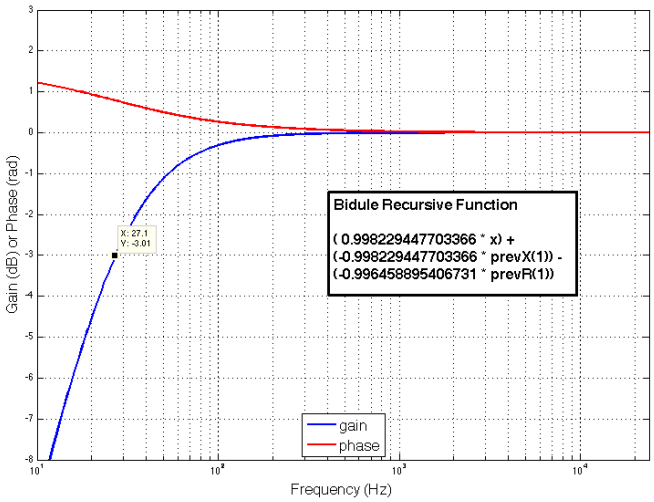

Abbildung 26: Hochpass 1. Ordnung
(Quelle: (17) Seite 21)

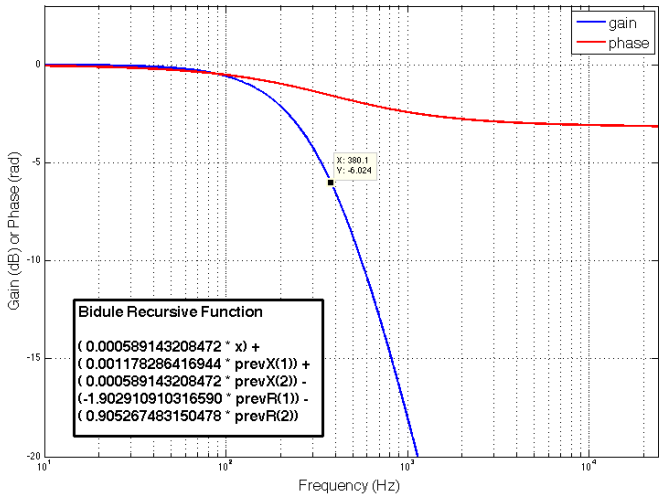

Abbildung 27: Tiefpass-Filter 2. Ordnung
(Quelle: (17) Seite 22)

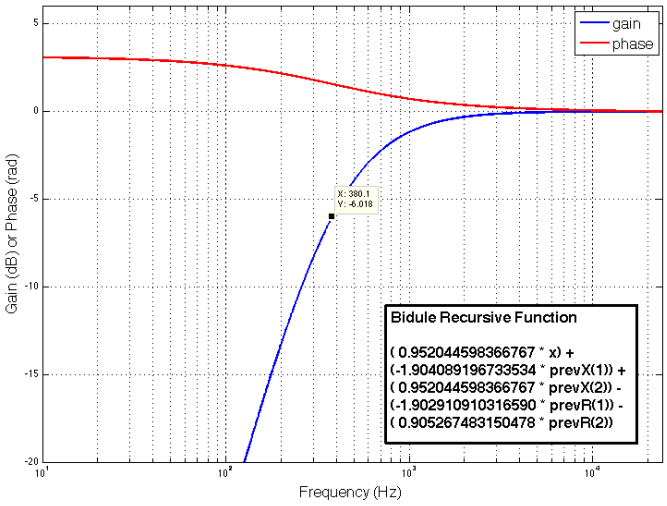

Abbildung 28: Hochpass-Filter 2. Ordnung
(Quelle: (17) Seite 22)

Ihre entnormierten Übertragungsfunktionen 1. und 2. Ordnung lauten:

$$H_{HP\ 1.Ord}\ (s) = \frac{s}{\omega_g + s}$$

$$H_{TP\ 2.Ord}(s) = \frac{\omega_g^2}{s^2 + \frac{\omega_g}{Q_\infty} s + \omega_g^2}$$

$$H_{HP\ 2.Ord}(s) = \frac{s}{s^2 + \frac{\omega_g}{Q_\infty} s + \omega_g^2}$$

Mit der Kreisfrequenz $\omega_g = 2\pi f_g$

Und der Polgüte Q_∞
$Q_\infty = \frac{1}{\sqrt{2}}$ für Butterworth-Approximation

Mit Shelving-Filter können hohe oder tiefe Frequenzen angehoben oder abgesenkt werden. Sie können mit Biquad-Filtern oder durch Parallelschaltung

eines Tiefpasses 1. Ordnung mit der Gleichspannungsverstärkung H_0 und einem System mit der Übertragungsfunktion $H(s) = 1$ einfach realisiert werden.

$$H(s) = 1 + H_{TP}(s) = 1 + \frac{H_0}{s+1}$$

$$H(s) = \frac{s + (1 + H_0)}{s+1} = \frac{s + V_0}{s+1}, V_0 > 1$$

Durch Variation des Parameters V_0 lassen sich hier beliebige Anhebungen ($V > 1$) oder Absenkungen ($V < 1$) einstellen.

Zur Realisierung eines digitalen Filters können die Pol- und Nullstellen der im S-Bereich entworfenen Übertragungsfunktion H(s) mit Hilfe der bilinearen Transformation in die Z-Ebene übertragen werden.

$$s = \frac{2}{T}\frac{z-1}{z+1}$$

$$K = \tan\left(\frac{\omega_g T}{2}\right)$$

$$T = \frac{1}{f_s}, \qquad f_s \ldots Samplingfrequenz$$

Für die Berechnung der Koeffizienten ergibt sich:

Hochpass (1. Ordnung)				
b_0	b_1	b_2	a_1	a_2
$\frac{1}{(K+1)}$	$-\frac{1}{(K+1)}$	0	$\frac{(K-1)}{(K+1)}$	0

Tiefpass (2. Ordnung, $Q_\infty = \frac{1}{2}$)				
b_0	b_1	b_2	a_1	a_2
$\frac{K^2}{1+2K+K^2}$	$\frac{2K^2}{1+2K+K^2}$	$\frac{K^2}{1+2K+K^2}$	$\frac{2(K^2-1)}{1+2K+K^2}$	$\frac{1-\sqrt{2}K+K^2}{1+2K+K^2}$

Hochpass (2. Ordnung, $Q_\infty = \frac{1}{2}$)				
b_0	b_1	b_2	a_1	a_2
$\frac{1}{1+2K+K^2}$	$\frac{-2}{1+2K+K^2}$	$\frac{1}{1+2K+K^2}$	$\frac{2(K^2-1)}{1+2K+K^2}$	$\frac{1-2K+K^2}{1+2K+K^2}$

Höhen-Shelving (Anhebung $V_0 = 10^{\frac{G}{20}}, Q_\infty = \frac{1}{\sqrt{2}}$)				
b_0	b_1	b_2	a_1	a_2
$\frac{V_0+\sqrt{2V_0}K+K^2}{1+\sqrt{2}K+K^2}$	$\frac{2(K^2-V_0)}{1+\sqrt{2}K+K^2}$	$\frac{V_0-\sqrt{2V_0}K+K^2}{1+\sqrt{2}K+K^2}$	$\frac{2(K^2-1)}{1+\sqrt{2}K+K^2}$	$\frac{1-\sqrt{2}K+K^2}{1+\sqrt{2}K+K^2}$

Höhen-Shelving (Absenkung $V_0 = 10^{-\frac{G}{20}}, Q_\infty = \frac{1}{\sqrt{2}}$)				
b_0	b_1	b_2	a_1	a_2
$\frac{1+\sqrt{2}K+K^2}{V_0+\sqrt{2V_0}K+K^2}$	$\frac{2(K^2-V_0)}{V_0+\sqrt{2V_0}K+K^2}$	$\frac{1-\sqrt{2}K+K^2}{V_0+\sqrt{2V_0}K+K^2}$	$\frac{2(K^2-1)}{V_0+\sqrt{2V_0}K+\frac{K^2}{V_0}}$	$\frac{1-\sqrt{2}K+K^2}{1+\sqrt{2}K+\frac{K^2}{V_0}}$

Tabelle 3: Formeln zur Berechnung der Koeffizienten eines Biquad-Filters (Quelle: (25) Seite 136)

Für einen Filter zur Nahfeldkompensation (siehe Kapitel 4.2.5) ergeben sich bei einem Lautsprecherabstand von zwei Metern, einer Schallgeschwindigkeit von 343 m/s und einer Samplingfrequenz von 48 kHz folgende Koeffizienten:

```
a  =[ 1                      -0.996433451030326]
b  =[ 0.998216725515163      -0.998216725515163]
fc = 27.2950727402601
```

Die zugehörigen Berechnungen der Koeffizienten und die Filterung eines Signals mittels IIR-Filter 1. Ordnung finden sich im Anhang unter my_nfcfilter.m

Bei einem Crossover-Network bestehend aus einem Tiefpass und einem Hochpass 2. Ordnung ist es essentiell, dass sie die gleiche Phase und eine 6dB Grenzfrequenz besitzen, sodass es nicht zu Auslöschungen oder

Verstärkungen kommt. Solche Filter eignen sich für die Implementierung eines Bandsplitting bzw. Multisystem Ambisonic Decoders, der getrennt für tiefe und hohe Frequenzen decodiert.

Bei einer Crossover-Frequenz von 380 Hz, einer Güte von $Q_\infty = \frac{1}{2}$ und einer Samplingfrequenz von 48 kHz ergeben sich für die Koeffizienten folgende Werte:

```
b_hp =  0.952044598366767     -1.90408919673353    0.952044598366767
b_lp =  0.00058914320847207    0.0011782864169442  0.00058914320847208
a    =  1                     -1.90291091031659    0.905267483150478
```

Die zugehörigen Berechnungen der Koeffizienten und die Filterung eines Signals mittels Biquad-Filter 1.Ordnung finden sich im Anhang unter my_xfilter.m

6 Implementierung

6.1 Einführung

Mitte der 80er-Jahre entstanden die ersten Midi-Sequencer wie Pro-16 von Steinberg für den Commodore 64 (1984), Pro-24 für den Atari ST (1986) oder Creator (später Notator). Anfang der 90er-Jahre waren Computer so weit, dass Audiosignale aufgenommen werden konnten, und boten somit erstmalig eine Alternative zur analogen Bandmaschine oder digitalen Stand-Alone Harddiskrekordern, digitale Effekte mussten jedoch mit Hilfe spezieller DSP Audiokarten berechnet werden, ein Beispiel dafür sind die TDM Plug-Ins für ProTools.

Das Wort „Plug-In", kommt aus dem Englischen und bedeutet soviel wie "hineinstecken". Ein Plug-In ist ein Computer-Programm, dass ein anderes übergeordnetes Programm (auch „Host" genannt) benötigt, mit dem es interagiert. Es stellt eine bestimmte Funktion bei Bedarf zur Verfügung.

1996 werden die technischen Möglichkeiten revolutioniert. Cubase VST (Virtual Studio Technologie) ist die erste native Software mit Echtzeit-Studioumgebung inklusive EQs, Effekten, Mixing und Automation, welche alle auf einem PC berechnet werden. 1997 wird VST zu einem offenen Standard, der es ermöglichte, Drittanbietern Plug-Ins und Audio-Hardware für die VST-Umgebung zu entwickeln.

Mit steigender Rechenleistung der Computer entwickelten sich mehrere Plug-In-Technologien, um Effekte in Echtzeit am Computer selbst zu berechnen, wie Digidesign's Real Time AudioSuite (RTAS), Linux Audio Developers Simple Plugins (LADSPA) und LV2, Microsoft DirectX und die Steinberg Virtual Studio Technology.

Mittlerweile gibt es unzählige Plattformen zur Musikproduktion und Bearbeitung von digitalem Audiomaterial, die als Plug-In-Host fungieren können, einige werde hier aufgezählt:

<u>Multitracksoftware:</u>

- Cubase
- Energy XT 1.4 / 2
- Nuendo

- ProTools HD
- Reaper
- Samplitude
- Sequoia

Modulare Software, Audio-Editoren und Plug-In Hosts

- Audition
- Wavelab
- Bidule
- Max-MSP
- Pure Data
- VSTHost
- Wavosaur

(Quelle: http://acousmodules.free.fr/hosts.htm, http://www.dancetech.com/article.cfm?threadid=154&lang=0)

Ein Plug-In funktioniert vom Prinzip her genauso wie ein analoges Gerät zur Bearbeitung von Audiosignalen in einem Kanalzug eines Mischpults. Es kann als Insert- oder Sendeffekt oder über AUX-Wege verwendet werden, die Möglichkeiten für Pre- und Post-Schaltungen hängen hierbei vom verwendeten Host ab.

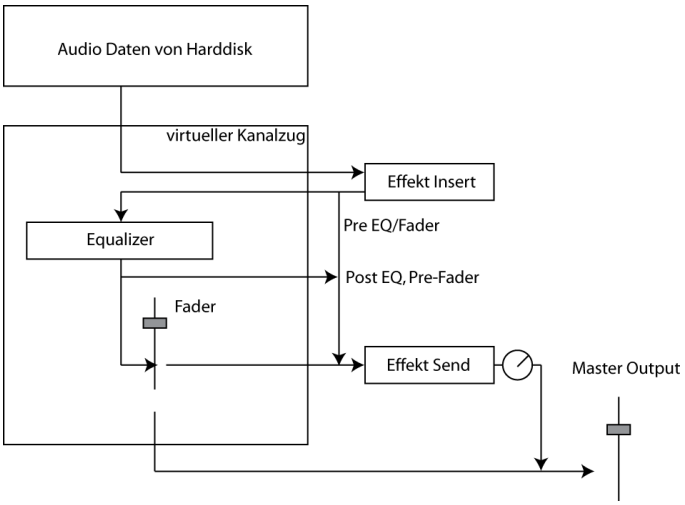

Abbildung 29: Virtueller Kanalzug mit Insert und Send-Effekt-Plug-In

Ein Plug-In ist aus der Sicht des Host-Programmes eine „Black Box“ mit verschiedener Anzahl an Ein- und Ausgängen. Es werden vom Host Zeiger auf die Audiodaten, die in Blocks oder Sequenzen unterteilt sind, an das Plug-In übergeben, welches die digitale Signalverarbeitung übernimmt und den Zeiger

nach der Berechnung an die Host-Applikation zurückgibt. Bei der Signalverarbeitung wird die CPU des Host-Rechners verwendet. Üblicherweise werden keine externen DSP-Karten benötigt. Die Parameter des Plug-Ins können in vielen Fällen zwar durch die Host-Applikation automatisiert werden, sind aber dem Plug-In eigen und können je nach Plug-In sehr unterschiedlich ausfallen.

Die Bedienung eines Plug-Ins erfolgt über das Graphical User Interface (GUI), welches oft analogen Gräten nachempfunden wird. Mit Drehreglern, Schiebereglern und Knöpfen können die Parameter der Bearbeitung eingestellt werden. Oft wird die diese Bearbeitung auch visualisiert wie z.B. mit EQ-Kurven, Kompressionskurven oder Peakmetern für In- und Output. Der kreativen Gestaltung sind hier keine Grenzen gesetzt.

Plug-Ins, die regelbare Parameter, aber kein GUI besitzen, werden in der VST-Umgebung oft ein individuelles User Interface zu Verfügung gestellt, welches von Host zu Host unterschiedlich ausfällt.

Abbildung 30: Individuelles GUI in Wavelab 5

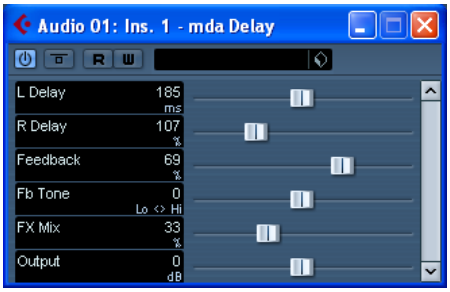

Abbildung 31: Individuelles GUI Cubase 4

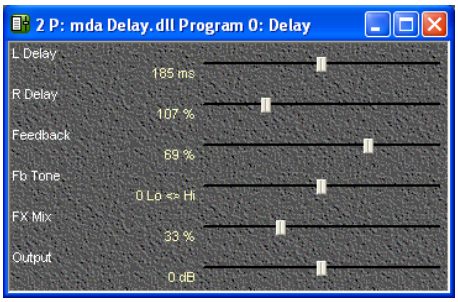

Abbildung 32: Individuelles GUI VST-Host

Im Programm „VST-Host“ können viele weitere Plug-In-Informationen angezeigt werden:

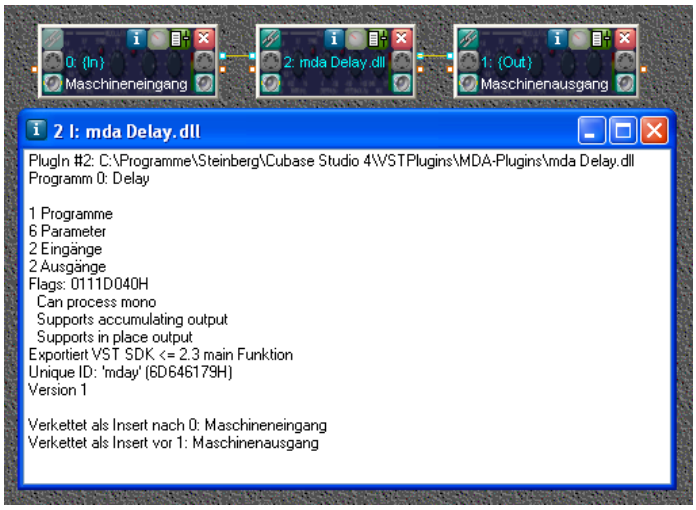

Abbildung 33: Erweiterte Information in VST-Host

Die Implementierung einfacher Plug-Ins kann somit in zwei Stufen erfolgen. Erstens die Entwicklung der DSP-Algorithmen und zweitens die Visualisierung und grafische Gestaltung. Diese Trennung ist oft auch sehr wichtig, da in der Softwareentwicklung für jede dieser zwei Stufen unterschiedliche spezialisierte Entwicklungsteams eingesetzt werden.

Der Source-Code eines VST-Plug-Ins ist plattformunabhängig, jedoch hängt die Dateistruktur des Plug-Ins von der jeweiligen Plattform ab, so ist unter Windows, ein VST-Plug-In eine multi-thread DLL (Dynamic Link Library), unter Mac OS X ein Bundle und unter BeOS und SGI (MOTIF und UNIX) eine Shared Library.

6.2 Steinberg Virtual Studio Technology und Software Developer Kit

Wie zuvor besprochen, gibt es viele unterschiedliche Technologien um ein Audio-Plug-In zu realisieren (VST, RTAS, LADSPA, DirectX), jedoch ist die VST-Architektur die, die am besten portabel ist und von fast allen Host-Programmen unterstützt wird. Alle in Kapitel 6.1 aufgezählten Programme können VST-Plug-Ins verwenden. Sollten gewisse Hosts VST nicht unterstützen, so gibt es in den meisten Fällen zumindest sogenannte „VST Wrapper“, welche die Verwendung ermöglichen, indem sie die VST für eine andere Architektur zur Laufzeit übersetzen.

Seit 1997 gibt es ein frei verfügbares Software Developer Kit (SDK) von Steinberg, welches Entwicklern/Entwicklerinnen ermöglicht, VST-Plug-Ins einfach zu realisieren. Das Steinberg VST SDK besteht aus einer Sammlung aus C++ Header-Files, mit Basisklassen, von denen das spezifische Plug-In abgeleitet werden kann. Diese Files dürfen auf keinen Fall verändert werden, da die Host-Applikation sich darauf verlässt, dass diese so wie sie sind zur Verfügung stehen. Die eigenen Effekte können von den Basisklassen AudioEffect (VST 1.0) und AudioEffectX (VST 2.x Erweiterungen) abgeleitet werden. In den abgeleiteten Klassen können dann beliebige Funktionen verändert oder hinzugefügt werden.

Die wichtigsten virtuellen Funktionen einer abgeleiteten Klasse sind „processReplacing()“ und „processDoubleReplacing()“, in denen die Signalverarbeitung stattfindet.

„processReplacing()“ berechnet die Signalverarbeitungsalgorithmen und überschreibt den Outputbuffer (Audioblock) wie bei einem Insert-Effekt. Optional kann „processDoubleReplacing()“ implementiert werden, um eine 64-bit genaue Verarbeitung bereitzustellen, im Gegensatz zur „normalen“ 32-bit Verarbeitung.

Der Wertebereich eines Audiosignals reicht von +1 bis -1, wobei +1 0dBFS und und 0.5 -6dBFS entspricht.

In dieser Arbeit wird Steinbergs SDK 2.4 verwendet, da die Version 3.0 bisweilen nur von wenigen Hosts verwendet werden kann und die zusätzlichen Features dieser Version in einem einfachen Decoder Plug-In nicht verwendet werden müssen.

6.3 Entwicklungsumgebung und C++ Programmierung

Im Steinberg VST SDK sind nicht nur die Basisklassen enthalten sondern auch vorgefertigte Beispiel-Plug-Ins erstellt für Visual Studio C++ für PC und CodeWarrior für MAC, die es dem Entwickler/ der Entwicklerin einfach machen, die Entwicklungsumgebung einzurichten, eines dieser Beispiele zu kompilieren und darauf aufzubauen.

Visual C++ Express Edition von Microsoft ist ein kostenfreier Compiler, der dazu verwendet werden kann, diese Projektdateien zu öffnen und ein Plug-In zu erstellen.

Um aus dem einfachen im VST SDK enthaltenen Beispiel-Plug-In „AGain“ eine in einem Host verwendbare dll-Datei zu erstellen, muss man nur die Datei „again.vcproj“ die unter „vstsdk2.4\public.sdk\samples\vst2.x\again\win\“ zu finden ist, in Visual Studio Express öffnen und auf „Erstellen“ klicken.

Sobald das erste Beispiel kompiliert ist und in einem Host, wie z.B. dem kostenfreien VST-Host, funktioniert, kann auf den verfügbaren Beispielen aufgebaut und ein eigener DSP-Algorithmus implementiert werden.

Um effizient zu arbeiten, verwendet man am besten immer den Ordner „C:\vstsdk2.4\public.sdk\samples\vst2.x\“, kopiert die Plug-In Beispiele, auf denen man aufbauen möchte in diesem und benennt sie je nach Bedarf um. Man umgeht somit die mühsame Erstellung eines neuen Visual Studio Projektes mit allen nötigen Header-Dateien und Projekteinstellungen, die für die Erstellung einer dll-Datei nötig sind.

6.4 Entwicklung der DSP-Algorithmen und Implementierung

Zuerst stellt sich die Frage, welche Features ein Ambisonic Decoder Plug-In haben sollte.

Ein einfacher Ambisonic Decoder (z.B. Energy Decoder) für Ambisonics 1. Ordnung, der für vier Lautsprecher in einem quadratischen Layout decodiert, ohne Nahfeld-Kompensation und Shelving-Filtern, besteht aus einer einfachen Matrixmultiplikation. Oder einfacher, aus einer gewichteten Addition der Eingangssignale. Der gleiche Decoder mit Nahfeld-Kompensation besitzt einen zusätzlichen Hochpass-Filter erster Ordnung für X, Y und Z. Falls die Decodierung auch noch frequenzabhängig gestaltet werden möchte, können entweder Shelving-Filter und ein Decoder oder Bandsplitting-Filter mit zwei Decodern (Multisystem) jeweils für tiefe oder hohe Frequenzen eingesetzt werden. Bei einer Decodierung für das irreguläre 5.1 ITU Layout wird im optimalen Fall getrennt für tiefe und hohe Frequenzen decodiert, mit zusätzlicher Implementierung einer Nahfeld-Kompensation.

Die Parameter, die zur Verfügung stehen sollten, sind eine Lautsprecher-Abstands-Einstellung für die Cut-Off-Frequenz des Nahfeld-Kompensations-Hochpass-Filters und eventuell ein Drop-Down-Menü zur Auswahl der Boxen-Aufstellung, was soviel bedeutet wie eine Änderung der Decoder-Koeffizienten.

Der Signalflussgraph eines Multisystem Ambisonic Decoders könnte daher wie folgt aussehen:

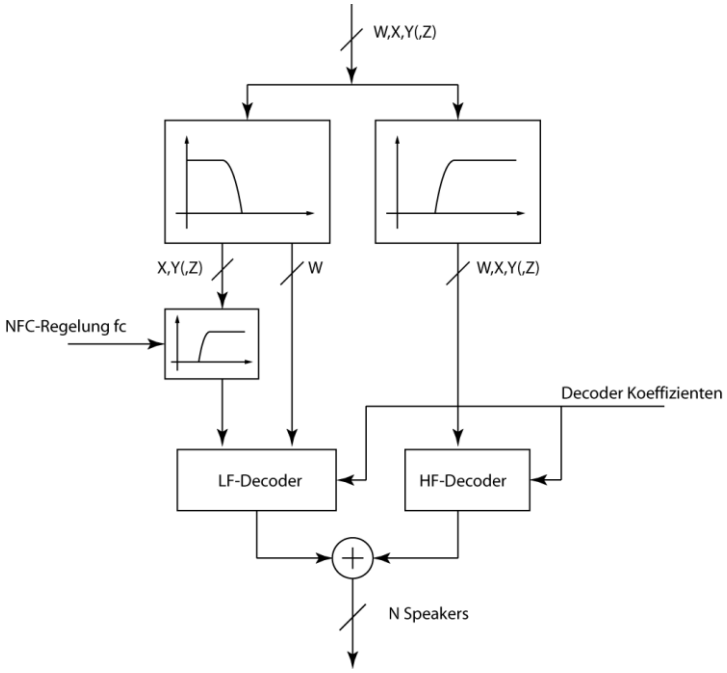

Abbildung 34: Möglicher Multisystem Ambisonic Decoder mit NFC

Zu implementieren sind also:

- Matrixmultiplikation bzw. gewichtete Addition der Eingangssignale
- Hochpass-Filter 1. Ordnung
- Phasengleicher Tief- und Hochpass mit 6dB Grenzfrequenz

Für die Matrix Multiplikation kann auf dem VST SDK-Beispiel „AGain“ aufgebaut werden, da es sich hierbei um nichts Anderes handelt, als eine Gewichtung der Eingangssignale W, X, Y (und optional Z, im Falle von periphoner Decodierung) mit den bestimmten Werten der Koeffizienten k der Decodermatrix.

```
void AGain::processReplacing (float** inputs, float** outputs,
VstInt32 sampleFrames)
{
    float* in1  =  inputs[0];
    float* in2  =  inputs[1];
    float* out1 = outputs[0];
    float* out2 = outputs[1];

    while (--sampleFrames >= 0)
    {
        (*out1++) = (*in1++) * fGain;
        (*out2++) = (*in2++) * fGain;
    }
}
```

C++ Code 1: AGain::processReplacing

```
void AMulti::processReplacing (float** inputs, float** outputs,
VstInt32 sampleFrames)
{
float* w  =  inputs[0];
float* x  =  inputs[1];
float* y  =  inputs[2];
float* z  =  inputs[3];

float* out1 = outputs[0];
float* out2 = outputs[1];
float* out3 = outputs[2];
float* out4 = outputs[3];
float k =  float(0.3535533);

while (--sampleFrames >= 0)
{
            float w =*in1++;
            float x =*in2++;
            float y =*in3++;
            float z =*in4++;
            (*out1++) = (w * k)+ (x *  k )+ (y *  k );
            (*out2++) = (w * k)+ (x *(-k))+ (y *  k );
            (*out3++) = (w * k)+ (x *(-k))+ (y *(-k));
            (*out4++) = (w * k)+ (x *  k )+ (y *(-k));
}
}
```

C++ Code 2: Beispiel für einen Velocity-Decoder, AMulti::processReplacing

Da die Systemidentifikation eines Plug-Ins etwas umständlich ist, wurden die Algorithmen zuerst im Matlab erstellt und geprüft. Die Matlab-Funktion zur Berechnung eines Hochpass-Filters 1. Ordnung für die Nahfeldkompensation ist im Anhang unter my_nfcfilter.m zu finden.

Wenn die in Matlab entwickelten Filter-Funktionen überprüft sind und ordnungsgemäß funktionieren, ist die Übersetzung in C++Code recht einfach. In *„Digital Filter Design and Implementation within the Steinberg Virtual Studio Technology (VST) Architecture“* (26) können wir eine ausführliche Beschreibung zur Filterimplementierung finden. Desweiteren kann man die Filterklasse nfc

(siehe Anhang) mit my_filter.m vergleichen und feststellen, dass nur geringe formale Unterschiede bestehen.

Beim Fall eines Ambisonic Decoders wird aber nicht nur ein Stereosignal gefiltert wie in (26), sondern gleich vier Ambisonic Signale, die durch jeweilige Hoch- und Tiefpass-Filterung acht Signale ergeben. Und im Falle einer zusätzlichen Nahfeld Kompensation ergibt sich eine Kaskade aus zwei Filtern.

Im Zuge dieser Arbeit wurden deshalb drei Klassen zur geeigneten Filterung der Ambisonic Signale erstellt (C++ Code im Anhang).

1. Klasse „nfc“
2. Klasse „hp“
3. Klasse „lp“

Jede dieser Klassen besitzt die Funktion „void setParameter“, die aufgerufen wird, um die Filterkoeffizienten zu berechnen und eine „processing(float sample)“-Funktion, um den jeweiligen Rückgabewert eines gefilterten Samples zu berechnen. Die jeweils vorangegangen x- und y-Werte der rekursiven Filter werden lokal in den Filter-Objekten gespeichert.

Bei Erstellung des Plug-Ins werden diese Filter-Objekte im Header des Plug-Ins deklariert, welche die jeweiligen Samples für die Rekursion speichern. Deklaration in “adecoder.h”:

```
protected:
 lp  W_LF, X_LF, Y_LF, Z_LF;
 hp  W_HF, X_HF, Y_HF, Z_HF;
 nfc X_LF_NFC, Y_LF_NFC, Z_LF_NFC;
```

Im Konstruktor des Plug-Ins werden die Filterkoeffizienten initialisiert

```
 W_HF.setParameter(xoverfreq, fs);
 X_HF.setParameter(xoverfreq, fs);
 Y_HF.setParameter(xoverfreq, fs);
 Z_HF.setParameter(xoverfreq, fs);

 W_LF.setParameter(xoverfreq, fs);
 X_LF.setParameter(xoverfreq, fs);
 Y_LF.setParameter(xoverfreq, fs);
 Z_LF.setParameter(xoverfreq, fs);

 X_LF_NFC.setParameter(Dist, fs);
 Y_LF_NFC.setParameter(Dist, fs);
 Z_LF_NFC.setParameter(Dist, fs);
```

Die Filterung der Signale durch das Plug-In findet später in der processReplacing-Funktion statt, wie hier am Beispiel einer Hochpassfilterung des W-Signals zu sehen ist.

```
float w = *in1++;
float whf = W_HF.process(w);
```

Die Signale werden, wie im Signalflussgraf eines Multisystem-Decoders dargestellt (Abbildung 34), gefiltert und jeweils an einen Decoder für hohe und einen für tiefe Frequenzen übergeben.

Um Phasengleichheit zu erhalten, muss der Ausgang des HF-Decoders phaseninvertiert werden.

Der Source-Code für einen einfachen Energy-Decoder ohne NFC und Crossoxer-Filter (Decoder1) für vier Lautsprecher im quadratischen Layout, einen Multisystem-Decoder mit NFC und Crossover-Filter (Decoder2) und einen Multisystem-Decoder für 5.1 ITU-Standard (Decoder3) sind im Anhang zu finden.

Natürlich besitzt ein Plug-In außer der Funktion „processReplacing“ noch viele andere Funktionen. Einige wichtige werden im Folgenden vorgestellt.

```
AudioEffect* createEffectInstance (audioMasterCallback audioMaster)
{
      return new ADecoder (audioMaster);
}
```

Falls im Host ein Plug-In geladen wird, wird hier ein Zeiger auf diese Effekt-Instanz erzeugt.

```
ADecoder::ADecoder (audioMasterCallback audioMaster)
: AudioEffectX (audioMaster, 1, 1)  // 1 program, 1 parameter only
{
	setNumInputs (4);		// B-Format in
	setNumOutputs (4);		// quadrophonic out
	setUniqueID ('Amb1');	// identify
	canProcessReplacing (); // supports replacing output

	vst_strncpy (programName, "Default", kVstMaxProgNameLen);	//
default program name

	float fs = getSampleRate();
	fDist = (float)0.5;		// init parameter at half of range
	float Dist  = float(0.5)+(fDist*float(2.5));	//		calculate
distance in meters

}
```

Im Konstruktor des Plug-Ins können unterschiedliche Parameter initialisiert werden, wie zum Beispiel:

- Anzahl der Ein- und Ausgänge,
- Samplingrate wird vom Host geholt für spätere Berechungen,
- Parameter des GUIs werden initialisiert,
- Initalisierung der im Header-File deklarierten protected Variablen,…

```
void ADecoder::setParameter (VstInt32 index, float value)
{
	fDist = value;
	update();
}
```

Set Parameter wird aufgerufen, falls einer der Parameter über das GUI verändert wird. Der neue Wert „value“ wird dann an das Plug-In übergeben und dort verarbeitet. In diesem Fall wird der Wert „value“ an „fDist“ übergeben und die Filter-koeffizienten in einer eigenen update()-Funktion erneut berechnet, welche im Header-File natürlich deklariert werden muss.

```
float ADecoder::getParameter (VstInt32 index)
{
	return fDist;
}
```

Gibt den Wert von „fDist“ an das GUI.

```
void ADecoder::getParameterName (VstInt32 index, char* label)
{
	vst_strncpy (label, "dist2spkr", kVstMaxParamStrLen);
}
```

Gibt den Namen des Parameters an das GUI weiter.

```
void ADecoder::getParameterDisplay (VstInt32 index, char* text)
{
	float v=0;
	v=float(0.5)+(fDist*float(2.5));
	float2string (v, text, kVstMaxParamStrLen);
}
```

Da der Parameter „fDist“ nur zwischen 0 und 1 definiert ist und wir aber z.B. die Distanz der Lautsprecher in Metern anzeigen lassen wollen, müssen wir hier den Parameter umrechnen und ihn dann mit „getParameterDisplay“ an das GUI übergeben.

```
void ADecoder::getParameterLabel (VstInt32 index, char* label)
{
      vst_strncpy (label, "meter", kVstMaxParamStrLen);
}
```

Zeigt die gewünschte Maßeinheit des Parameters im GUI an.

6.5 Qualitätsprüfung

Die Qualitätsprüfung stellte sich als recht schwierig heraus. Da die in *„Is My Decoder Ambisonic?“* (17) von Heller und Lee verwendeten Matlab-Testfunktionen auf der dort angegeben Internetseite nicht mehr verfügbar waren, mussten diese nochmals erstellt werden.

Ein Fünf-Kanal-wav-File mit W, X, Y, Z wurde erstellt, welches Impulse in 72 Richtungen im Abstand von 5° im B-Format encodiert. Die fünfte Audiospur enthält Synchronisations-Impulse. Dieses File wird mittels eines Decoder-Plug-Ins decodiert, um gerichtete Impulsantworten zu erhalten. Mit vier Lautsprechern und 72 Richtungen ergeben sich daraus 288 Impulsantworten zur Systemidentifikation. Die Anwendung des Plug-Ins auf das Testsignal wurde in WavoSaur, einem kostenfreien Multichannelfähigen VST-Host durchgeführt.

Für die Auswertung werden die Magnitude-Vektoren $\overrightarrow{r_V}$ und $\overrightarrow{r_E}$ gemäß Formel 4.1 und Formel 4.2 berechnet.

Durch die Matlab-Funktion `imp_dir_spkr=speaker_imp(x_out, 72, 4)` werden die einzelnen Impulsantworten in Array geschrieben mit den Indizes „Samples“ x „Richtungen“ x „Lautsprecher“.

Als nächstes werden mit `fft_dir_spkr=compute_fft_imags(imp_dir_spkr)` die Spektren der einzelnen Impulse berechnet und in ein Array mit den Indizes „Punkte der FFT“x „Richtungen“ x „Lautsprecher“ geschrieben.

Nun werden mit der Funktion `Vpxyz=sum_pxyz(fft_dir_spkr,spkr_weights)` und `Epxyz=sum_pxyz(fft_dir_spkr.*conj(fft_dir_spkr),spkr_weights)` die nicht

normalisierten Vektoren $\overrightarrow{r_V}$ und $\overrightarrow{r_E}$ berechnet. Idizes von Vpxyz sind Frequenz, Richtung und die kartesischen Komponenten (x, y, z).

Mit `rv=r_from_pxyz(Vpxyz)` und `re=r_from_pxyz(Epxyz);` werden die Magnitude-Vektoren indiziert mit Frequenz und Richtung berechnet, welche mit

```
theta=(0:5:355)*pi/180;
rho=rv(freq,:); % rv (freq x richtungen) -> frequenz 2, alle richungen
polar(theta,rho,'--r')
```

in einem Polarplot dargestellt werden können.

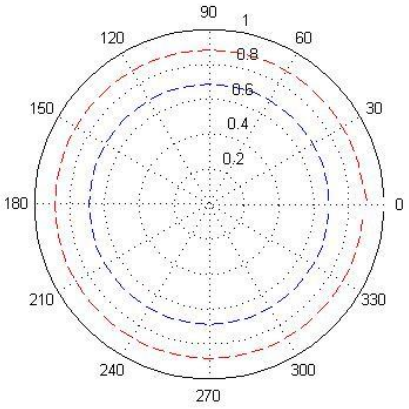

Abbildung 35: Multi-System-Decoder2 517 Hz, rot Velocity, blau Energy

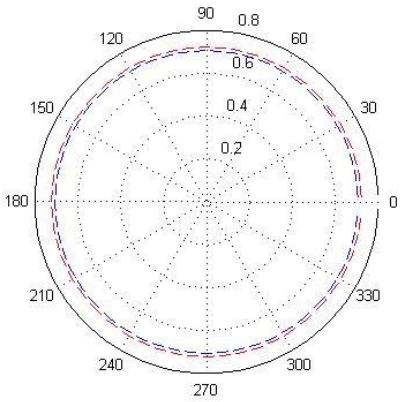

Abbildung 36: Multi-System-Decoder2 2067 Hz, rot Velocity, blau Energy

Funktionstests wurden in WavoSaur und Steinberg Nuendo 3 durchgeführt. Alle Plug-Ins funktionierten einwandfrei. Decoder3 wurde mit gepulstem Rauschen in 20 Richtungen mit Bruce Wiggins 5.1 ITU Decoder verglichen und funktioniert annähernd gut, auch wenn bei Bruce Wiggins Decoder weit mehr Einstellmöglichkeiten über das GUI bereitstehen.

7 Zusammenfassung und Ausblick

Im Zuge dieser Arbeit wurde das Thema Ambisonics und Ambisonic Decoding behandelt. Drei Ambisonic Decoder Plug-Ins, jeweils mit unterschiedlichen Features, wurden exemplarisch in der Steinberg Virtual Studiotechnologie mit Hilfe des VST SDK 2.4 und Visual Studio Express 2008 erstellt.

Diese können in digitalen Workstations, die multichannelfähig sind und die VST-Technologie unterstützen (wie z.B.: Steinberg Nuendo oder WavoSaur), verwendet werden, um B-Format für das jeweils geeignete Lautsprechersetup zu decodieren.

- Decoder1:
  Energie-Decoder für quadratisches Lautsprechersetup mit Gain Regelung
- Decoder2:
  Multisystem-Decoder für quadratisches Lautsprechersetup mit regelbarem Nahfeld-Kompensations-Filter.
- Decoder3:
  Multisystem-Decoder für irreguläres 5.1 ITU Lautsprechersetup mit regelbarem Nahfeld-Kompensations-Filter, mit optimierten Decoder-Koeffizienten von Fons Adriaensens AmbDec.

Das Ziel der Arbeit, das Thema Ambisonics und Ambisonic Decoding zu erläutern und einen Ambisonic Decoder als Plug-In exemplarisch zu realisieren, wurde somit erreicht. Einige Möglichkeiten der Decoderoptimierung für das für Ambisonics irregulären 5.1 ITU-Lautsprecher-Layout wurden erläutert.

Um die erstellten Ambisonic Decoder ausführlicher zu testen, bedarf es jedoch eines umfangreicheren Matlab-Toolsets als in Kapitel 6.5 beschrieben, welches die Darstellung des Frequenzgangs und der Phase ermöglicht. Die genannten Ambisonic Decoder könnten auch über ausgedehnteren Hörtests mit bereits erhältlichen Ambisonic Decodern wie Wiggins „WAD“ oder DecoPro von http://www.gerzonic.net/ (ohne Shelving filter) verglichen und bewertet werden.

Mögliche weitere Schritte der Verbesserung der entworfenen Plug-Ins wären die Implementierung eines GUIs, mit weiteren Parametern für die zur

Einstellung der Cross-Over-Frequenz, ein Ein/Aus-Button für Nahfeld-Kompensation, ein Dropdown-Menü für diverse Decoder-Koeffizienten (Energy, Velocity, Cardioid) und für verschiedene Lautsprecheraufstellungen (quadratisch, rechteckig, hexagonal, ITU), Implementierung der Filter als FFT-Filter und vieles mehr.

Literaturverzeichnis

1. **Wiggins, Bruce.** *An Investigation into the Real-time Manipulation and Control of Three-Dimensional Sound Fields.* University of Derby : Phd Thesis, 2004.

2. **wikipedia.** Surround. *http://en.wikipedia.org.* [Online] [Cited: 12 3, 2009.] http://en.wikipedia.org/wiki/Surround.

3. **Dolby.** http://www.dolby.com. [Online] 2009. [Cited: 12 3, 2009.] http://www.dolby.com/professional/technology/cinema/dolby-digital.html.

4. http://de.wikipedia.org. [Online] http://de.wikipedia.org/wiki/Dolby_Digital.

5. **Gerzon, Michael.** *General Metatheory of Auditory Localisation.* Wien : Audio Engineering Society Preprint, März 1992.

6. **Benjamin, Eric.** *Ambisonic Loudspeaker Arrays.* San Francisco : 125th AES Convention, 2-5 Okt, 2008.

7. **Majdak, Piotr.** Algorithmen in Akustik und Computermusik. *http://piotr.majdak.com/alg/VO/spatial1.pdf.* [Online] 2008. [Cited: 04 18, 2010.]

8. **Rumsey, Francis and Tim, McCormick.** *Sound And Recording.* 6. s.l. : Focal Press, 2009.

9. **Glasgas, Ralph.** http://www.ambiophonics.org. [Online] [Cited: 12 11, 2009.] http://www.ambiophonics.org/AmbioBook/intro.html.

10. **wikipedia.** Quadrophonie. *http://de.wikipedia.org.* [Online] [Cited: 12 11, 2009.] http://de.wikipedia.org/wiki/Quadrophonie.

11. **Dolby.** http://www.dolby.com. [Online] 2009. [Cited: 12 10, 2009.] http://www.dolby.com/about/who-we-are/our-history/history-4.html.

12. **Farina, Angelo, et al.** Ambiophonic Principles for the Recording and Reproduction of Surround Sound for Music. *Ambiophonic Principles for the Recording and Reproduction of Surround Sound for Music.* Parma, ITALY : AES, 2001. 1875 .

13. **Farina, Angelo and Ugolotti, Emanuele.** *Software Implementation of B-Format Encoding and Decoding.* AES Convention Amsterdam : Audio Engineering Society Preprint, Mai 1998.

14. **Benjamin, Eric, Lee, Richard and Heller, Aron.** *Localisation in Horizontal-Only Ambisonic Systems.* San Francisco : 121st AES Convention, Okt. 5-8, 2006.

15. **Sontacchi, Alois and Höldrich, Robert.** Schallfeldreproduktion durch ein verbessertes Holophonie – Ambisonic System. s.l. : IEM.

16. **Gerzon, Michael.** *Practical Periphony: The reproduction of full-sphere sound.* AES Convention London : Audio Engineering Society Preprint, Februar 1980.

17. **Heller, Aaron, Lee, Richard and Benjamin, Eric.** *Is My Decoder Ambisonic.* San Francisco : 125th AES Convention, 2-5 Okt, 2008.

18. **Richard Lee.** ambisonia.com. *http://www.ambisonia.com/Members/ricardo/shelfs.zip/view.* [Online] Mai 26, 2008. [Cited: 10 6, 2009.] http://www.ambisonia.com/Members/ricardo/shelfs.zip/view.

19. **Dr. Graber, Gerhard.** *Raumakustik Skriptum.* SS 2004.

20. **Moore, David and Wakefield, Jonathan.** *The Design and Detailed Analysis of First Order Ambisonic Decorders for the ITU Layout.* Vienna, Austria : 122nd AES Convention, 5-8 Mai, 2007.

21. —. *The Design of Improoved First Order Ambisonic Decoders by the Application of Range Removal and Importance in a Heuristic Search Algorithm.* London : AES 31st International Conference, 25-27 Juni, 2007.

22. —. *Exploiting Human Spatial Resolution in Surround Sound Decoder Desing.* San Francisco : 125th AES Convention, 2-5 Okt, 2008 .

23. **Breebaart, Jeroen.** *Binaural processing model based on contralateral inhibition.* Niederlande : Acoustical Society of America, 2001. 43.66.Pn, 43.66.Ba, 43.66.Dc.

24. **Oppenheim, Alan V. and Schäfer, Roland W.** *Zeitdiskrete Signalverarbeitung.* s.l. : Pearson Studium, 2004.

25. **Zölzer, Udo.** *Digitale Audiosignalverarbeitung.* Wiesbaden : Teubner, 2005. ISBN 3-519-26180-4.

26. **Osorio-Goengaga, Roberto.** *Digital Filter Desing and Implementation within the Steinberg Virtual Studio Technology (VST) Architecture.* New York : 119th AES Convention, 7-10 Okt. 2005.

27. soundfield. [Online] [Cited: 01 4, 2010.] http://www.soundfield.com/.

Abbildungsverzeichnis

Formeln

Tabellen

Anhang

Matlabcode

```
% ------------------decodingbsp1.m-----------------------------------
% -------------------------------------------------------------------
% beispiel eines velocity decoding prozesses im quardatischen layout

%testsignal
b=[ 0.7071,   1,   0,  0;
    0.7071,   0,   1,  0;
    0.7071,  -1,   0,  0;
    0.7071,   0,  -1,  0]

%Decoder-matrix für quardatisches array velocity decode

sD=[ 0.3536,  0.3536,  0.3536, 0;
    0.3536, -0.3536,  0.3536, 0;
    0.3536, -0.3536, -0.3536, 0;
    0.3536,  0.3536, -0.3536, 0]

%speakerfeeds
spkr_feeds=b*sD.'

%speaker_weights = x und y komponenten der lautsprecher

spkr_weights = [  0.7071,  0.7071;
                 -0.7071,  0.7071;
                 -0.7071, -0.7071;
                  0.7071, -0.7071]

%Synthese aus den Speakerfeeds
y=spkr_feeds*spkr_weights
```

```
% -------------------ambidecode1.m-------------------------------------
% ---------------------------------------------------------------------

function ret = ambidecode1(S,k)
% Ambisonic Speaker Decoder: ret = function ambidecode2(S,k)
%           computation after Gerzon
%           then scaling
%
% input:    S   ... speakermatrix [1,1; 1,-1;...]
%                   in diametrically opposed pairs
%                   optional 2D or 3D
%           k   ... X/W - ratio (optional)
%                   1           for velocity (default value)
%                   sqrt(1/2)   for energy
%                   sqrt(3)/3   for energy 3D
%                   0.5         for controlled opposites 2D
%
% output:   matrix with the weights of W X Y Z for each speaker
%               each row corresponds to the weights of the specific
speaker
%
%
% written by martin pauser 20/02/2010


if ~exist ('k')
   k=1;
end

%determinating matrix orientation
dim = size(S);
m = min(dim); %m dimensions
n = max(dim); %n speakers

%coverting matrix to m x n matrix with  m < n
if dim(:,1) > dim(:,2)
    S=S';
end


A = zeros(m:m);
for i = 1:n
    dir = S(:,i)/norm(S(:,i));
    A = A + dir*dir';
    directions(:,i) = dir;
end

factor = (sqrt(m)/n);

a = ones(1,n);
bc = sqrt(1/2)*n*k*inv(A)*directions;
abc = zeros(n,4);


abc = [a;bc]';

%skalierung
%n = number of speaker

%----------2D ----------
if m==2
```

```
    zahler=(sqrt(2)/n)^2*3;   % mal 3 wegen  W X Y
    nenner=sum(abc(1,:).^2);
    rat=sqrt(zahler/nenner);
    ret = abc*rat;
end

%---------3D ----------
if m==3
    zahler=(sqrt(2)/n)^2+(sqrt(3)/n)^2*3;   % mal 3 wegen  X Y Z
    nenner=sum(abc(1,:).^2);
    rat=sqrt(zahler/nenner);
    ret = abc*rat;
end


% ------------------ambidecode2.m-----------------------------------
% ------------------------------------------------------------------

function ret = ambidecode2(S,k)
% Ambisonic Speaker Decoder: ret = function ambidecode2(S,k)
%           computation via Pseudoinverse
%
% input:    S   ... speakermatrix [1,1,(1); 1,-1,(1);...]
%                   in diametrically opposed pairs
%                   optional 2D or 3D
%           k   ... X/W - ratio (optional)
%                   1           for velocity (default value)
%                   sqrt(1/2)   for energy 2D
%                   sqrt(3)/3   for energy 3D
%                   0.5         for controlled opposites 2D
%
% output:   matrix with the weights of W X Y Z for each speaker
%               each row corresponds to the weights of the specific
speaker
%
%              cutoff frequency for near field compensation for each
speaker
%
% written by martin pauser 20/02/2010


if ~exist('k')
    k=1; %velocity decode
end

% m = number of speakers
% n = dimension 2 (2D), 3(3D)
[m,n]=size(S);

if n <= 2
    S(:,3) = 0;
end

if n==3
    if max(S(:,3))==0
        n=2;
    end
end


directions = zeros(m,4);
```

```
for i = 1:m
    dir= S(i,:)./(sqrt(S(i,:)*S(i,:)'));
    fc(i)=348/(2*pi*(sqrt(S(i,:)*S(i,:)')));
    directions(i,:) = [sqrt(1/2),dir];
end

if n==2
   Wrat=sqrt((n+1)/(2*k^2+1));
elseif n==3
   Wrat=sqrt((n+1)/(3*k^2+1));
end

Xrat=Wrat*k;
WXratio = diag([Wrat,Xrat,Xrat,Xrat]);

fc=fc' %cutoff frequency for XYZ for each speaker

ret = pinv(directions')* WXratio';
```

```
% ------------------- my_nfcfilter.m----------------------------------
% --------------------------------------------------------------------
function out = my_nfcfilter(in,d,fs)
%function out = my_nfcfilter(in,d,fs)
%input:     in = input wave vector
%              d   = distance to compensate (optional, default val= 2
meters)
%               fs  = sampling frequency       (optional, default val=
48000Hz)
%
%output:    out = vector, length(out)=length(in)
%
%1st order highpass filter to compensate nearfield effect
%computation via recursiv-filtering
%written by Martin Pauser 2010
if ~exist('d')
   d = 2
end

if ~exist('fs')
    fs = 44100;
end

c  = 343;
fc = c/(2*pi*d)
k= tan(pi*fc/fs);

b(1)=1/(k+1);
b(2)=-b(1);
a(1)=1;
a(2)=(k-1)*b(1);

len=length(in)+1;
y=zeros(len,1);
x=zeros(len,1);
x(2:len)=in;

for n = 2:len
  y(n)=b(1)*x(n)+b(2)*x(n-1)-a(2)*y(n-1);
end

out = y(2:end);
```

```
% ------------------- my_xfilter.m-----------------------------------
% ------------------------------------------------------------------------
function out = my_xfilter(in,fc,fs)
%function [LF, HF] = my_xfilter(in,fc,fs)
%input:     in = input wave vector
%           fc = crossover frequency (optional, default val= 380 Hz)
%           fs = sampling frequency  (optional, default val= 48000Hz)
%
%output:    out n x 2 Matrix [LF,HF], n = length(in)
%           1st column Low Frequency
%           2nd column High Frequency
%
%computation via biquad-filtering
%written by Martin Pauser 2010

if ~exist('fc')
   fc = 380;

end
if ~exist('fs')
    fs = 48000;
end
k= tan(pi*fc/fs);
q=1/2;
%------HF-section-----------------------
b(1)=1/(k*k+(k/q)+1);
b(2)=(-2)*b(1);
b(3)=b(1);
a(1)=1;
a(2)=(2*(k*k-1))/(k*k+(k/q)+1);
a(3)=(k*k-(k/q)+1)/(k*k+(k/q)+1);

len=length(in)+2;
y=zeros(len,1);
x=zeros(len,1);
x(3:len)=in;

for n = 3:len
  y(n)=b(1)*x(n)+b(2)*x(n-1)+b(3)*x(n-2)-a(2)*y(n-1)-a(3)*y(n-2);
end
HF = y(3:end);

%------LF-section-----------------------
b(1)=(k*k)/(k*k+(k/q)+1);
b(2)=2*b(1);
b(3)=b(1);
a(1)=1;
a(2)=(2*(k*k-1))/(k*k+(k/q)+1);
a(3)=(k*k-(k/q)+1)/(k*k+(k/q)+1);

len=length(in)+2;
y=zeros(len,1);
x=zeros(len,1);
x(3:len)=in;

for n = 3:len
  y(n)=b(1)*x(n)+b(2)*x(n-1)+b(3)*x(n-2)-a(2)*y(n-1)-a(3)*y(n-2);
end
LF = y(3:end);

out = [LF, HF];
```

C++ Code

Decoder1

```
//-------------------------------------------------------------------------
// VST Plug-Ins SDK
// Version 2.4          $Date: 2010/15/03 $
//
// Category     : VST 2.x SDK Samples
// Filename     : aenergymain.cpp
// Created by   : Martin Pauser
// Description  : 1. Order Ambisonic Energy Decoder for quadratic
Layout
//
// © 2010, Martin Pauser, All Rights Reserved
//-------------------------------------------------------------------------

#ifndef __aenergy__
#include "aenergy.h"
#endif

//-------------------------------------------------------------------------
AudioEffect* createEffectInstance (audioMasterCallback audioMaster)
{
      return new AMulti (audioMaster);
}


//-------------------------------------------------------------------------
// VST Plug-Ins SDK
// Version 2.4          $Date: 2010/03/15 $
//
// Category     : VST 2.x SDK Amisonic Decoder
// Filename     : aenergy.h
// Created by   : Martin Pauser
// Description  : 1. Order Ambisonic Energy Decoder for quadratic
Layout
//
// © 2010, Martin Pauser, All Rights Reserved
//-------------------------------------------------------------------------

#ifndef __aenergy__
#define __aenergy__

#include "public.sdk/source/vst2.x/audioeffectx.h"

//-------------------------------------------------------------------------
class AMulti : public AudioEffectX
{
public:
      AMulti (audioMasterCallback audioMaster);
      ~AMulti ();

      // Processing
      virtual void processReplacing (float** inputs, float** outputs,
VstInt32 sampleFrames);

      // Program
      virtual void setProgramName (char* name);
```

```
	virtual void getProgramName (char* name);

	// Parameters
	virtual void setParameter (VstInt32 index, float value);
	virtual float getParameter (VstInt32 index);
	virtual void getParameterLabel (VstInt32 index, char* label);
	virtual void getParameterDisplay (VstInt32 index, char* text);
	virtual void getParameterName (VstInt32 index, char* text);

	virtual bool getEffectName (char* name);
	virtual bool getVendorString (char* text);
	virtual bool getProductString (char* text);
	virtual VstInt32 getVendorVersion ();

protected:
	float fGain;
	char programName[kVstMaxProgNameLen + 1];
	};

#endif


//-------------------------------------------------------------------------
// VST Plug-Ins SDK
// Version 2.4          $Date: 2010/03/07$
//
// Category     : VST 2.x SDK Amisonic Decoder
// Filename     : aenergy.cpp
// Created by   : Martin Pauser
// Description  : 1. Order Ambisonic Energy Decoder for quadratic
Layout
//
// © 2010, Martin Pauser, All Rights Reserved
//-------------------------------------------------------------------------

#include "aenergy.h"
#include <math.h>
#include <stdio.h>

#ifndef __aenergy__
#include "aenergy.h"
#endif


//-------------------------------------------------------------------------
AMulti::AMulti (audioMasterCallback audioMaster)
: AudioEffectX (audioMaster, 1, 1)  // 1 program, 1 parameter only
{
	setNumInputs (4);       // B-Format in
	setNumOutputs (4);            // quadrophonic out
	setUniqueID ('Amb1');   // identify
	canProcessReplacing (); // supports replacing output

	fGain = 1.f;                  // default to 0 dB

	vst_strncpy (programName, "Default", kVstMaxProgNameLen);  //
default program name
}

//-------------------------------------------------------------------------
AMulti::~AMulti ()
```

```
{
	// nothing to do here
}

//------------------------------------------------------------------------
void AMulti::setProgramName (char* name)
{
	vst_strncpy (programName, name, kVstMaxProgNameLen);
}

//------------------------------------------------------------------------
void AMulti::getProgramName (char* name)
{
	vst_strncpy (name, programName, kVstMaxProgNameLen);
}

//------------------------------------------------------------------------
void AMulti::setParameter (VstInt32 index, float value)
{
	fGain = value;
}

//------------------------------------------------------------------------
float AMulti::getParameter (VstInt32 index)
{
	return fGain;
}

//------------------------------------------------------------------------
void AMulti::getParameterName (VstInt32 index, char* label)
{
	vst_strncpy (label, "Gain", kVstMaxParamStrLen);
}

//------------------------------------------------------------------------
void AMulti::getParameterDisplay (VstInt32 index, char* text)
{
	dB2string (fGain, text, kVstMaxParamStrLen);
}

//------------------------------------------------------------------------
void AMulti::getParameterLabel (VstInt32 index, char* label)
{
	vst_strncpy (label, "dB", kVstMaxParamStrLen);
}

//------------------------------------------------------------------------
bool AMulti::getEffectName (char* name)
{
	vst_strncpy (name, "Gain", kVstMaxEffectNameLen);
	return true;
}

//------------------------------------------------------------------------
bool AMulti::getProductString (char* text)
{
	vst_strncpy (text, "Gain", kVstMaxProductStrLen);
	return true;
}

//------------------------------------------------------------------------
```

```
bool AMulti::getVendorString (char* text)
{
      vst_strncpy (text, "Martin Plug-Ins", kVstMaxVendorStrLen);
      return true;
}

//-----------------------------------------------------------------------
VstInt32 AMulti::getVendorVersion ()
{
      return 1000;
}


//-----------------------------------------------------------------------
void AMulti::processReplacing (float** inputs, float** outputs,
VstInt32 sampleFrames)
{
      float* in1  =  inputs[0];
      float* in2  =  inputs[1];
      float* in3  =  inputs[2];
      float* in4  =  inputs[3];

      float* out1 = outputs[0];
      float* out2 = outputs[1];
      float* out3 = outputs[2];
      float* out4 = outputs[3];

      float e1 = float (0.433012701892219);
      float e2 = float (0.306186217847897);

      while (--sampleFrames >= 0)
    {
            float w =*in1++;
            float x =*in2++;
            float y =*in3++;
            float z =*in4++;

            (*out1++) = ((w * e1)+ (x *   e2 )+ (y *   e2 ))*fGain;
            (*out2++) = ((w * e1)+ (x * (-e2))+ (y *   e2 ))*fGain;
            (*out3++) = ((w * e1)+ (x * (-e2))+ (y * (-e2)))*fGain;
            (*out4++) = ((w * e1)+ (x *   e2 )+ (y * (-e2)))*fGain;

      }
}
```

Filter-Klassen

```
//------------------------------------------------------------------------
// Filename     : nfc.h
// Created by   : Martin Pauser
// Description  : filterklasse für nahfeld kompensation
//
// © 2010, Martin Pauser, All Rights Reserved
//------------------------------------------------------------------------
#ifndef __nfc__
#define __nfc__

#include "public.sdk/source/vst2.x/audioeffectx.h"

class nfc
{
public:
	nfc(void);
	~nfc(void);

	float process(float sample);
	void setParameter(float dist, float samplerate);

private:
//--------variablen NFC-filter

	double fDist;
	double x1, y1;
	double b0, b1, a1;
	double fs, fc, c, d, k;
};
#endif


//------------------------------------------------------------------------
// Filename     : nfc.cpp
// Created by   : Martin Pauser
// Description  : filterklasse für nahfeld kompensation
//
// © 2010, Martin Pauser, All Rights Reserved
//------------------------------------------------------------------------

#include "nfc.h"
#include <math.h>
#include <stdlib.h>

#ifndef M_PI
#define M_PI		3.14159265358979323846
#endif

nfc::nfc()
{
	x1=0;
	y1=0;
}

nfc::~nfc(void)
{
}
```

```
void nfc::setParameter(float dist, float samplerate)
{
	d = double (dist);
	fs = double (samplerate);

	c  = 343;
	fc = c/(2*M_PI*d);
	k  = tan(M_PI*fc/fs);

	b0 = 1/(k+1);
	b1 = (-b0);
	a1 = (k-1)*b0;
}

float nfc::process(float sample)
{
	double x0 = sample;
	double y0;

	y0 = b0*x0 + b1*x1 - a1*y1;

	/*shift x0 to x1 and y0 to y1*/
	x1=x0;
	y1=y0;

	return float(y0);
}

//-----------------------------------------------------------------------
// Filename     : hp.h
// Created by   : Martin Pauser
// Description  : filterklasse für highpassfilter eines X-Overnetworks
//
// © 2010, Martin Pauser, All Rights Reserved
//-----------------------------------------------------------------------
#ifndef __hp__
#define __hp__

#include "public.sdk/source/vst2.x/audioeffectx.h"

/* whatever sample type you want */
typedef double smp_type;


class hp
{
public:
	hp(void);
	~hp(void);

	float process(float sample);
	void setParameter(float fXfreq, float samplerate);

protected:
	//--------variablen von HP-filter
	double x1, x2, y1, y2;		// buffer for past samples
	double fs, fc, k, q;
	double b0, b1, b2, a1, a2;	// filterkoeffizienten

};
#endif
```

```
//---------------------------------------------------------------------
// Filename     : hp.cpp
// Created by   : Martin Pauser
// Description  : filterklasse für highpassfilter eines X-Overnetworks
//
// © 2010, Martin Pauser, All Rights Reserved
//---------------------------------------------------------------------
#include "hp.h"
#include <math.h>
#include <stdlib.h>

#ifndef M_PI
#define M_PI            3.14159265358979323846
#endif


hp::hp()
{
      x1= double (0);
      x2= double (0);
      y1= double (0);
      y2= double (0);
}

hp::~hp(void)
{
}

void hp::setParameter(float fXfreq, float samplerate)
{
      fc = double(fXfreq);
      fs = double(samplerate);

      k  = double (tan((M_PI*fc)/fs));
      q  = .5;

      //---------------calculate filtercoefficients init--------------
      b0=1/(k*k+(k/q)+1);
      b1=(-2)*b0;
      b2=b0;
      a1=(2*((k*k)-1))/(k*k+(k/q)+1);
      a2=(k*k-(k/q)+1)/(k*k+(k/q)+1);
}

float hp::process(float sample)
{
      double x0= sample;
      double y0;
      y0 = b0*(x0 - 2*x1  + x2) - a1*y1 - a2*y2;

      /* shift x1 to x2, x0 to x1 */
      x2=x1;
      x1=x0;

      /* shift y1 to y2, y0 to y1 */
      y2=y1;
      y1=y0;

      return float(y0);
}
```

```
//-------------------------------------------------------------------------
// Filename     : lp.h
// Created by   : Martin Pauser
// Description  : filterklasse für lowpassfilter eines X-Overnetworks
//
// © 2010, Martin Pauser, All Rights Reserved
//-------------------------------------------------------------------------
#ifndef __lp__
#define __lp__

#include "public.sdk/source/vst2.x/audioeffectx.h"

class lp
{
public:
	lp(void);
	~lp(void);

	float process(float sample);
	void setParameter(float fXfreq, float samplerate);

private:
	//--------variablen LP-filter
	double x1, x2, y1, y2;		// buffer for past samples
	double fs, fc, k, q;
	double b0, b1, b2, a1, a2;	// filterkoeffizienten

};
#endif
```

```
//-------------------------------------------------------------------------
// Filename     : lp.cpp
// Created by   : Martin Pauser
// Description  : filterklasse für lowpassfilter eines X-Overnetworks
//
// © 2010, Martin Pauser, All Rights Reserved
//-------------------------------------------------------------------------
#include "lp.h"
#include <math.h>
#include <stdlib.h>

#ifndef M_PI
#define M_PI		3.14159265358979323846
#endif


lp::lp()
{
	x1=x2=y1=y2=0;
}

lp::~lp(void)
{
}

void lp::setParameter(float fXfreq, float samplerate)
{
	fc = double (fXfreq);
	fs = double (samplerate);

	k  = tan((M_PI*fc)/fs);
	q = .5;
```

```
	//---------------calculate filtercoefficients init--------------
	b0=(k*k)/(k*k+(k/q)+1);
	b1=2*b0;
	b2=b0;
	a1=(2*((k*k)-1))/(k*k+(k/q)+1);
	a2=(k*k-(k/q)+1)/(k*k+(k/q)+1);
}

float lp::process(float sample)
{
	double x0 = sample;
	double y0;

	y0 = b0*(x0 + 2*x1  + x2) - a1*y1 - a2*y2;

	/* shift x1 to x2, x0 to x1 */
	x2=x1;
	x1=x0;

	/* shift y1 to y2, y0 to y1 */
	y2=y1;
	y1=y0;

	return float(y0);
}
```

Decoder2

```
//-------------------------------------------------------------------------
// VST Plug-Ins SDK
// Version 2.4          $Date: 2010/14/03 $
//
// Category     : VST 2.x
// Filename     : adecoder.h
// Created by   : Martin Pauser
// Description  : 1. Order Ambisonic mulitsystem decoder for quadratic
Layout
//
// © 2010, Martin Pauser, All Rights Reserved
//-------------------------------------------------------------------------

#ifndef __adecoder__
#define __adecoder__

#include "public.sdk/source/vst2.x/audioeffectx.h"
#include "nfc.h"
#include "lp.h"
#include "hp.h"


//-------------------------------------------------------------------------
class ADecoder : public AudioEffectX
{
public:
      ADecoder (audioMasterCallback audioMaster);
      ~ADecoder ();

      // Processing
      virtual void processReplacing (float** inputs, float** outputs,
VstInt32 sampleFrames);

      // Program
      virtual void setProgramName (char* name);
      virtual void getProgramName (char* name);

      // Parameters
      virtual void setParameter (VstInt32 index, float value);
      virtual float getParameter (VstInt32 index);
      virtual void getParameterLabel (VstInt32 index, char* label);
      virtual void getParameterDisplay (VstInt32 index, char* text);
      virtual void getParameterName (VstInt32 index, char* text);

      virtual bool getEffectName (char* name);
      virtual bool getVendorString (char* text);
      virtual bool getProductString (char* text);
      virtual VstInt32 getVendorVersion ();

protected:
      //--------variablen NFC-filter
      void update();
      float fDist;

      long size;
      lp  W_LF, X_LF, Y_LF, Z_LF;
      hp  W_HF, X_HF, Y_HF, Z_HF;
      nfc X_LF_NFC, Y_LF_NFC, Z_LF_NFC;
```

```
	char programName[kVstMaxProgNameLen + 1];
};

#endif



//-------------------------------------------------------------------------
// VST Plug-Ins SDK
// Version 2.4		$Date: 2010/14/03 $
//
// Category     : VST 2.x
// Filename     : adecoder.cpp
// Created by   : Martin Pauser
// Description  : 1. Order Ambisonic mulitsystem decoder for quadratic
Layout
//
// © 2010, Martin Pauser, All Rights Reserved
//-------------------------------------------------------------------------

//#include "adecoder.h"
#include <math.h>
#include <stdio.h>

#ifndef __adecoder__
#include "adecoder.h"
#endif

//-------------------------------------------------------------------------
AudioEffect* createEffectInstance (audioMasterCallback audioMaster)
{
	return new ADecoder (audioMaster);
}


//-------------------------------------------------------------------------
ADecoder::ADecoder (audioMasterCallback audioMaster)
: AudioEffectX (audioMaster, 1, 1)  // 1 program, 1 parameter only
{
	setNumInputs (4);		// B-Format in
	setNumOutputs (4);			// quadrophonic out
	setUniqueID ('Amb1');	// identify
	canProcessReplacing (); // supports replacing output

	vst_strncpy (programName, "Default", kVstMaxProgNameLen);  //
default program name

	float fs = getSampleRate();
	fDist = (float)0.5;
	// init parameter at half of range
	float Dist  = float(0.5)+(fDist*float(2.5));   // calculate
distance in meters
	float xoverfreq = 480;

	//----------init Filters-------------------------

	W_HF.setParameter(xoverfreq, fs);
	X_HF.setParameter(xoverfreq, fs);
	Y_HF.setParameter(xoverfreq, fs);
	Z_HF.setParameter(xoverfreq, fs);
```

```
	W_LF.setParameter(xoverfreq, fs);
	X_LF.setParameter(xoverfreq, fs);
	Y_LF.setParameter(xoverfreq, fs);
	Z_LF.setParameter(xoverfreq, fs);

	X_LF_NFC.setParameter(Dist, fs);
	Y_LF_NFC.setParameter(Dist, fs);
	Z_LF_NFC.setParameter(Dist, fs);

}

//-----------------------------------------------------------------------
ADecoder::~ADecoder ()
{
}

//-----------------------------------------------------------------------
void ADecoder::update()
{
   //------NFC-Filter-koeffizienten update hier
	float fs = getSampleRate();
	float Dist  = float(0.5)+(fDist*float(2.5));

	X_LF_NFC.setParameter(Dist, fs);
	Y_LF_NFC.setParameter(Dist, fs);
	Z_LF_NFC.setParameter(Dist, fs);

}

//-----------------------------------------------------------------------
void ADecoder::setProgramName (char* name)
{
	vst_strncpy (programName, name, kVstMaxProgNameLen);
}

//-----------------------------------------------------------------------
void ADecoder::getProgramName (char* name)
{
	vst_strncpy (name, programName, kVstMaxProgNameLen);
}

//-----------------------------------------------------------------------
void ADecoder::setParameter (VstInt32 index, float value)
{
	fDist = value;
	update();
}

//-----------------------------------------------------------------------
float ADecoder::getParameter (VstInt32 index)
{
	return fDist;
}

//-----------------------------------------------------------------------
void ADecoder::getParameterName (VstInt32 index, char* label)
{
	vst_strncpy (label, "dist2spkr", kVstMaxParamStrLen);
}
```

```
//-----------------------------------------------------------------------
void ADecoder::getParameterDisplay (VstInt32 index, char* text)
{
      float v=0;
      v=float(0.5)+(fDist*float(2.5));
      float2string (v, text, kVstMaxParamStrLen);
}

//-----------------------------------------------------------------------
void ADecoder::getParameterLabel (VstInt32 index, char* label)
{
      vst_strncpy (label, "meter", kVstMaxParamStrLen);
}

//-----------------------------------------------------------------------
bool ADecoder::getEffectName (char* name)
{
      vst_strncpy (name, "Ambidecoder2", kVstMaxEffectNameLen);
      return true;
}

//-----------------------------------------------------------------------
bool ADecoder::getProductString (char* text)
{
      vst_strncpy (text, "Ambidecoder", kVstMaxProductStrLen);
      return true;
}

//-----------------------------------------------------------------------
bool ADecoder::getVendorString (char* text)
{
      vst_strncpy (text, "Martin Media Technologies",
kVstMaxVendorStrLen);
      return true;
}

//-----------------------------------------------------------------------
VstInt32 ADecoder::getVendorVersion ()
{
      return 1000;
}

//-----------------------------------------------------------------------
void ADecoder::processReplacing (float** inputs, float** outputs,
VstInt32 sampleFrames)
{
      float* in1  =  inputs[0];
    float* in2  =  inputs[1];
      float* in3  =  inputs[2];
      float* in4  =  inputs[3];

      float* out1 = outputs[0];
    float* out2 = outputs[1];
      float* out3 = outputs[2];
    float* out4 = outputs[3];

      float b0=float(0.93494185246596);
      float a1=float(-1.86769564979658);
      float a2=float(0.872071760067264);

      while (--sampleFrames >= 0)
```

```
    {
            float w = *in1++;
            float x = *in2++;
            float y = *in3++;
            float z = *in4++;

            float out1_HF, out2_HF, out3_HF, out4_HF;
            float out1_LF, out2_LF, out3_LF, out4_LF;

            //highpass filter current sample
            float whf = W_HF.process(w);
            float xhf = X_HF.process(x);
            float yhf = Y_HF.process(y);
            float zhf = Z_HF.process(z);

            //Lowpass filter current sample
            float wlf = W_LF.process(w);
            float xlf = X_LF.process(x);
            float ylf = Y_LF.process(y);
            float zlf = Z_LF.process(z);

            // apply NFC filter
            float xlf_nfc = X_LF_NFC.process(xlf);
            float ylf_nfc = Y_LF_NFC.process(ylf);
            float zlf_nfc = Z_LF_NFC.process(zlf);



            //----decode HIGH FREQUENCY-------------------------
            float e1 = float (0.433012701892219);
            float e2 = float (0.306186217847897);
            (out1_HF) = (whf * e1)+ (xhf *   e2 )+ (yhf *   e2 );
            (out2_HF) = (whf * e1)+ (xhf * (-e2))+ (yhf *   e2 );
            (out3_HF) = (whf * e1)+ (xhf * (-e2))+ (yhf * (-e2));
            (out4_HF) = (whf * e1)+ (xhf *   e2 )+ (yhf * (-e2));

            //----decode LOW FREQUENCY-------------------------
            float v = float (0.353553390593274);
            (out1_LF) = (wlf * v)+ (xlf_nfc *   v )+ (ylf_nfc *   v );
            (out2_LF) = (wlf * v)+ (xlf_nfc * (-v))+ (ylf_nfc *   v );
            (out3_LF) = (wlf * v)+ (xlf_nfc * (-v))+ (ylf_nfc * (-v));
            (out4_LF) = (wlf * v)+ (xlf_nfc *   v )+ (ylf_nfc * (-v));


            //-----putting HF and LF together-----------------
            (*out1++) = out1_LF - out1_HF;
            (*out2++) = out2_LF - out2_HF;
            (*out3++) = out3_LF - out3_HF;
            (*out4++) = out4_LF - out4_HF;
      }
}
```

Decoder3

```
//-------------------------------------------------------------------------
// VST Plug-Ins SDK
// Version 2.4          $Date: 2010/14/03 $
//
// Category     : VST 2.x
// Filename     : adecoder.cpp
// Created by   : Martin Pauser
// Description  : 1. Order Ambisonic mulitsystem decoder for 5.1 ITU
Layout
//
// © 2010, Martin Pauser, All Rights Reserved
//-------------------------------------------------------------------------

//
#include "adecoder.h"
#include <math.h>
#include <stdio.h>

#ifndef __adecoder__
#include "adecoder.h"
#endif

//-------------------------------------------------------------------------
AudioEffect* createEffectInstance (audioMasterCallback audioMaster)
{
      return new ADecoder (audioMaster);
}


//-------------------------------------------------------------------------
ADecoder::ADecoder (audioMasterCallback audioMaster)
: AudioEffectX (audioMaster, 1, 1)  // 1 program, 1 parameter only
{
      setNumInputs (4);       // B-Format in
      setNumOutputs (5);
      setUniqueID ('Amb2');   // identify
      canProcessReplacing (); // supports replacing output

      vst_strncpy (programName, "Default", kVstMaxProgNameLen);  //
default program name

      float fs = getSampleRate();
      fDist = (float)0.5;
      // init parameter at half of range
      float Dist  = float(0.5)+(fDist*float(2.5));    // calculate
distance in meters
      float xoverfreq = 480;

      //----------init Filters-------------------------
      x0=x1=x2=y0=y1=y2=0;

      W_HF.setParameter(xoverfreq, fs);
      X_HF.setParameter(xoverfreq, fs);
      Y_HF.setParameter(xoverfreq, fs);
      Z_HF.setParameter(xoverfreq, fs);

      W_LF.setParameter(xoverfreq, fs);
      X_LF.setParameter(xoverfreq, fs);
```

```
	Y_LF.setParameter(xoverfreq, fs);
	Z_LF.setParameter(xoverfreq, fs);

	X_LF_NFC.setParameter(Dist, fs);
	Y_LF_NFC.setParameter(Dist, fs);
	Z_LF_NFC.setParameter(Dist, fs);

}

//------------------------------------------------------------------------
ADecoder::~ADecoder ()
{
}


//------------------------------------------------------------------------
void ADecoder::update()
{
   //------NFC-Filter-koeffizienten update hier
	float fs = getSampleRate();
	float Dist  = float(0.5)+(fDist*float(2.5));

	X_LF_NFC.setParameter(Dist, fs);
	Y_LF_NFC.setParameter(Dist, fs);
	Z_LF_NFC.setParameter(Dist, fs);

}

//------------------------------------------------------------------------
void ADecoder::setProgramName (char* name)
{
	vst_strncpy (programName, name, kVstMaxProgNameLen);
}

//------------------------------------------------------------------------
void ADecoder::getProgramName (char* name)
{
	vst_strncpy (name, programName, kVstMaxProgNameLen);
}

//------------------------------------------------------------------------
void ADecoder::setParameter (VstInt32 index, float value)
{
	fDist = value;
	update();
}

//------------------------------------------------------------------------
float ADecoder::getParameter (VstInt32 index)
{
	return fDist;
}

//------------------------------------------------------------------------
void ADecoder::getParameterName (VstInt32 index, char* label)
{
	vst_strncpy (label, "dist2spkr", kVstMaxParamStrLen);
}

//------------------------------------------------------------------------
void ADecoder::getParameterDisplay (VstInt32 index, char* text)
```

```
{
	float v=0;
	v=float(0.5)+(fDist*float(2.5));
	float2string (v, text, kVstMaxParamStrLen);
}

//------------------------------------------------------------------------
void ADecoder::getParameterLabel (VstInt32 index, char* label)
{
	vst_strncpy (label, "meter", kVstMaxParamStrLen);
}

//------------------------------------------------------------------------
bool ADecoder::getEffectName (char* name)
{
	vst_strncpy (name, "B-to-5.1-ITU", kVstMaxEffectNameLen);
	return true;
}

//------------------------------------------------------------------------
bool ADecoder::getProductString (char* text)
{
	vst_strncpy (text, "Ambidecoder", kVstMaxProductStrLen);
	return true;
}

//------------------------------------------------------------------------
bool ADecoder::getVendorString (char* text)
{
	vst_strncpy (text, "Martin Media Technologies",
kVstMaxVendorStrLen);
	return true;
}

//------------------------------------------------------------------------
VstInt32 ADecoder::getVendorVersion ()
{
	return 1000;
}

//------------------------------------------------------------------------
void ADecoder::processReplacing (float** inputs, float** outputs,
VstInt32 sampleFrames)
{
	float* in1  =  inputs[0];
	float* in2  =  inputs[1];
	float* in3  =  inputs[2];
	float* in4  =  inputs[3];

	float* out1 = outputs[0];
	float* out2 = outputs[1];     //LF
	float* out3 = outputs[2];   //center speaker
	float* out4 = outputs[3];     //RF
	float* out5 = outputs[4];

	while (--sampleFrames >= 0)
    {

		float w = *in1++;
		float x = *in2++;
```

```
            float y = *in3++;
            float z = *in4++;

            float out1_HF, out2_HF, out3_HF, out4_HF, out5_HF;
            float out1_LF, out2_LF, out3_LF, out4_LF, out5_LF;

            //highpass filter current sample
            float whf = W_HF.process(w);
            float xhf = X_HF.process(x);
            float yhf = Y_HF.process(y);
            float zhf = Z_HF.process(z);

            //Lowpass filter current sample
            float wlf = W_LF.process(w);
            float xlf = X_LF.process(x);
            float ylf = Y_LF.process(y);
            float zlf = Z_LF.process(z);

            // apply NFC filter
            float xlf_nfc = X_LF_NFC.process(xlf);
            float ylf_nfc = Y_LF_NFC.process(ylf);
            float zlf_nfc = Z_LF_NFC.process(zlf);


            //----decode HIGH FREQUENCY-------------------------
            (out1_HF) = (whf * (float) 0.56252)+(xhf *(float) (-
0.21649))+ (yhf *(float)  0.27315  );
            (out2_HF) = (whf * (float) 0.32762)+(xhf *(float)   0.28791
)+ (yhf *(float)  0.27803  );
            (out3_HF) = (whf * (float) 0.19330)+(xhf *(float)   0.28972
)+ (yhf *(float)  0.00000  );
            (out4_HF) = (whf * (float) 0.32762)+(xhf *(float)   0.28791
)+ (yhf *(float) (-0.27803));
            (out5_HF) = (whf * (float) 0.56252)+(xhf *(float) (-
0.21649))+ (yhf *(float) (-0.27315));

            //----decode LOW FREQUENCY-------------------------
            (out1_LF) = (wlf * (float)0.42033 )+ (xlf_nfc *(float)(-
0.31225))+ (ylf_nfc *(float)  0.33020 );
            (out2_LF) = (wlf * (float)0.19770 )+ (xlf_nfc *(float)
0.28782 )+ (ylf_nfc *(float)  0.28882 );
            (out3_LF) = (wlf * (float)0.05803 )+ (xlf_nfc *(float)
0.20597 )+ (ylf_nfc *(float)  0.00000 );
            (out4_LF) = (wlf * (float)0.19770 )+ (xlf_nfc *(float)
0.28782 )+ (ylf_nfc *(float)(-0.28882));
            (out5_LF) = (wlf * (float)0.42033 )+ (xlf_nfc *(float)(-
0.31225))+ (ylf_nfc *(float)(-0.33020));


            //-----putting HF and LF together-----------------
            (*out1++) = out1_LF - out1_HF;
            (*out2++) = out2_LF - out2_HF;
            (*out3++) = out3_LF - out3_HF;
            (*out4++) = out4_LF - out4_HF;
            (*out5++) = out5_LF - out5_HF;
      }
}
```