Akustische Holografie und Holofonie WF, LU

Directivity Visualizer
Dokumentation

Paul Bereuter
Clemens Frischmann
Felix Holzmüller

Betreuer: DI Dr.rer.nat. Franz Zotter

Graz, 25. August 2020

institut für elektronische musik und akustik

Zusammenfassung

Seit einiger Zeit gibt es den Ansatz, über Aufnahmen mit Mikrofonarrays die Abstrahlung und Richtwirkung von Instrumenten zu schätzen und zu berechnen, wie beispielsweise ausgeführt in [Zot09]. Ziel dieses Projektes war es, basierend auf Überlegungen aus [Zag19], die auf ein Ursignal bezogenen Richtungsfilter betragsmäßig zu berechnen und zu visualisieren. Hierzu wurde, aufbauend auf dem EnergyVisualizer der IEM Plug-Ins [RZGH20], ein VST Plug-In (Compute Zerophase Visualization of Directivity - C0ViD) mittels des JUCE Frameworks [SP20] erstellt, womit diese Abstrahlmuster in Echtzeit als Heatmap dargestellt werden können. Über eine optionale, im Signalfluss vorgezogene Bandpassfilterung können so auch schmalbandig gültige Richtmuster betrachtet werden.

Inhaltsverzeichnis

1 Einleitung

Bereits seit über einem Jahrzehnt wird am IEM an der Messung und Berechnung von Abstrahlmustern geforscht [Zot09], beispielsweise von Instrumenten [BM10, BMKF20], Sängern [BB19] oder Lautsprechern [Eng20]. Hierfür wurde auch ein eigens entwickeltes 64-Kanal Mikrofon-Arrays gebaut [Hoh09]. Bislang konnten diese Aufnahmen und daraus berechnete Daten nur offline analysiert und visualisiert werden [BFR18]. Mit dem hier vorgestellten VST Plug-In „Compute Zero-Phase Visualization of Directivity" (C0ViD) kann die Berechnung und Visualisierung der Richtungsfilter nun in Echtzeit mittels kompatiblen DAWs erfolgen. Abbildungen sowie eine Tabelle der kartesischen Mikrofonpositionen des Mikrofon-Arrays sind in Anhang A ersichtlich. Das Plug-In wurde im JUCE Framework aus dem bereits vorhandenen EnergyVisualizer der IEM-Plug-In Suite [RZGH20] weiterentwickelt. Das kompilierte Plug-In und ein Link zum Quellcode können am IAEM [1] abgerufen werden.

2 Grundlagen

Die Berechnung der Richtungsfilter basiert auf den Überlegungen in der Masterarbeit von Franck Zagala [Zag19]. Im folgenden Abschnitt werden nur die für die Implementierung wichtigen Zusammenhänge skizziert. Auf tiefergreifende Zusammenhänge und Erklärungen sowie die Berechnung der Phase wird deshalb nicht eingegangen, diese können in der Masterarbeit nachgeschlagen werden. Im Sinne der leichten Lesbarkeit werden auch alle Formelzeichen hieraus übernommen.

Die folgenden Beschreibungen beziehen sich jeweils auf ein Frame τ einer STFT. Demnach muss dieser Vorgang für jedes Frame durchgeführt werden.

2.1 Ursignalschätzung

Die Abstrahlung einer Quelle kann als SIMO-System modelliert werden. Hier verfolgt man den Ansatz, das die Richtungsabhängigkeit mit dem Koeffizientenvektor $\boldsymbol{\chi}[k]$ der Kugelflächenfunktionen in der Kurzzeit-Fouriertransformation als

$$\boldsymbol{\chi}[k] = U[k]\boldsymbol{\Psi}[k] \tag{1}$$

mit dem Ursignal $U[k]$ und den Abstrahlfiltern $\boldsymbol{\Psi}[k]$ zu modellieren, wobei k den Index des Frequenzbins darstellt.

Das Ursignal $U[k]$ wird auf Basis der Kurzzeit-Spektren der einzelnen Mikrofonsignale $X_{\tau,\lambda}[k]$ gewonnen, wobei τ den zeitlichen Abschnitt und $\lambda \in \{1, \ldots, \Lambda\}$ den Mikrofonkanal symbolisieren. Hierzu werden die Signale zuerst mittels STFT [2] in den Frequenzbe-

1. `https://iaem.at/kurse/ss20/akustische-holografie-und-holofonie-lu/richtwirkung-ursignal`
2. Auf die gewählten Parameter für die STFT wird in Unterabschnitt 3.1 eingegangen.

reich transformiert und zur Vermeidung von Interferenzen werden ausschließlich Betragsspektren zur Schätzung des Ursignals verarbeitet.

Dazu wird die l^p-Norm der Kurzzeit-Betragsspektren der Mikrofonsignale herangezogen. Sie ist definiert als:

$$|U_\tau[k]| = \left(\sum_{\lambda=1}^{\Lambda} |X_{\tau,\lambda}[k]|^p \right)^{1/p} \tag{2}$$

berechnet werden.

2.2 Berechnung der Richtungsfilter

Mit dem Betrag des Ursignals kann nun das Betragsspektrum des Richtungsfilters $|P_{\tau,\lambda}[k]|$ für die diskreten Richtungen der jeweiligen Mikrofone bestimmt werden. Analog zu Gleichung 1 kann der Zusammenhang im Richtungsbereich allgemein als

$$X_{\tau,\lambda}[k] = P_{\tau,\lambda}[k] U_\tau[k] \tag{3}$$

aufgestellt werden. Typischer Weise wird die Phaseninformation verworfen, und nach Umformung erhalten wir für die Beträge der Kurzzeit-Richtungsfilter

$$|P_{\tau,\lambda}[k]| = \frac{|X_{\tau,\lambda}[k]|}{|U_\tau[k]|}. \tag{4}$$

Die gefundenen Betragsfrequenzgänge der Richtungsfilter werden nun in weiteren Schritten zur Visualisierung des Richtungsfilters herangezogen. Der Ablauf der Verarbeitungsschritte zur Visualisierung wird in Abschnitt 2.4 genauer erläutert.

2.3 Bandpassfilterung des Richtungsfilter in der Frequenzdomäne

Um eine schmalbandige Visualisierung zu ermöglichen, um bspw. einzelne Teiltöne eines Instrumentes zu visualisieruen, wird der Richtungsfilter vor der Visualisierung Bandpass gefiltert. Inspiriert durch die Skirt-Filter, die in [Zag19, S.49] zur Vereinfachung der Richtungsfilter Frequenzgänge verwendet wurden, wurde ein Bandpass-Skirt-Filter entworfen welcher schmalbandige Richtungsfiltervisualisierungen ermöglicht. Der Frequenzgang des Filters $\boldsymbol{\alpha}_\beta[k]$ wurde mittels Exponentialfunktion

$$\begin{gathered} \boldsymbol{\alpha}_\beta[k_c] = \left(e^{-\frac{1}{\beta}} \right)^{\boldsymbol{k_f} - k_c} \\ \beta \in [0,1] \ldots \text{ Filterbreitenparameter} \\ \boldsymbol{k_f} \ldots \text{ Vektor mit diskreten Frequenzbins} \\ k_c \ldots \text{ Center-Frequenzbin des Bandpass-Skirt-Filters} \end{gathered} \tag{5}$$

entworfen. Der Frequenzgang der Bandpass-Skirt-Filter $\boldsymbol{\alpha}_\beta[k_c]$ und der Frequenzgang eines Beispielsignals X sind in Abbildung 1 ersichtlich. Das in Abbildung 1 visualisierte Beispielsignal besteht aus additiv überlagerten Sinussignalen. Der Grundton liegt bei $f_0 = 1000$ Hz, die 5 Obertöne der überlagerten Sinuskomponenten liegen bei den ersten 5 ganzzahligen Vielfachen der Grundfrequenz.

Die Mittenfrequenz des Bandpass-Skirt-Filters wurde in der Abbildung mit $k_c = 24$ gewählt. Der 24. Frequenzbin entspricht bei der zur Erstellung der Abbildung durchgeführten FFT (FFT-Größe: $N_{\text{FFT}} = 512$ samples und Samplingfrequenz: $f_s = 44.1$ kHz) einer Frequenz von $f[k_c] = 1981.05$ Hz. Es sind 3 Varianten des Bandpass-Skirt-Filter-Frequenzganges eingezeichnet, jeweils für unterschiedliche Filterbreiten die mittels Parameter β eingestellt werden können.

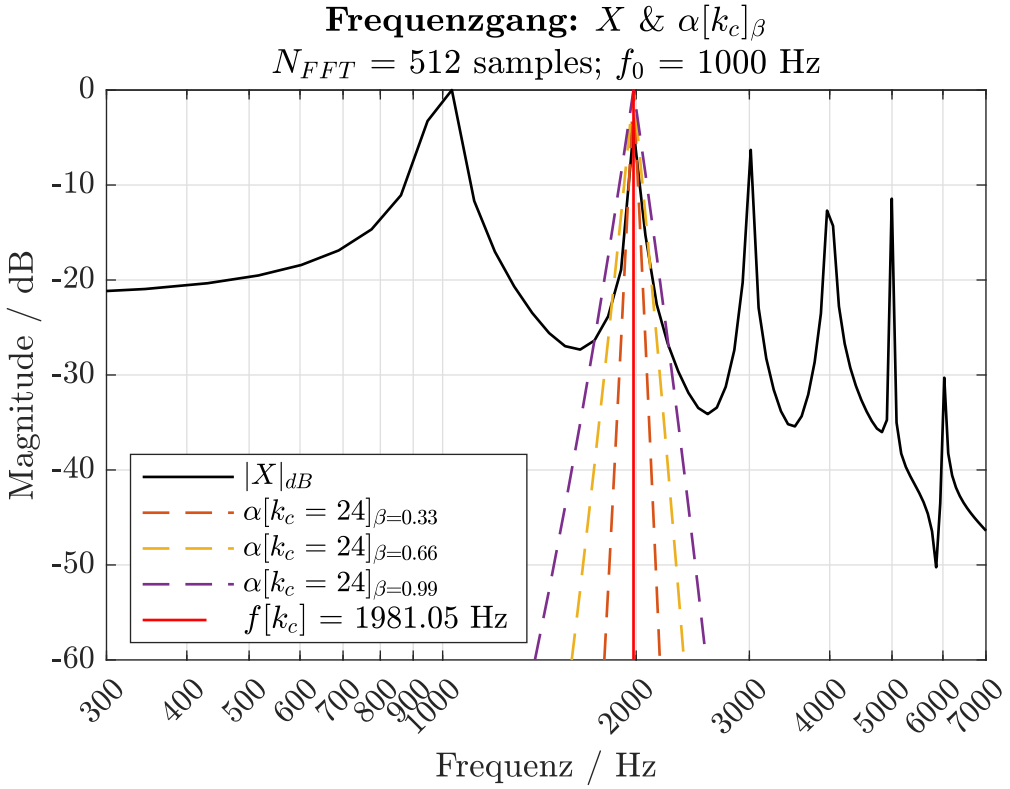

Abbildung 1 – Beispielsignal und Bandpass-Skirt-Filter Frequenzgang

Mittels elementweiser Multiplikation kann nun die Bandpassfilterung im Frequenzbereich durchgeführt werden. Die Multiplikation für einen Frequenzbin kann als

$$\left|\tilde{P}_{\tau,\lambda}[k]\right| = \left(\frac{|X_{\tau,\lambda}[k]|}{|U_\tau[k]|}\right) \cdot \alpha_\beta^k[k_c] \tag{6}$$

notiert werden, wobei $\alpha_\beta^k[k_c]$ nun den Wert des Betragsfrequenzganges des Bandpass-Skirt-Filter am k-ten Frequenzbin darstellt. Die Implementierung der Bandpassfilterung im VST-Plugin mittels `for`-Schleifen wird in Kapitel 3.1.2 näher behandelt.

2.4 Visualisierung des Richtungsfilters

Sind die Betragswerte der Richtungsfilter im Frequenzbereich berechnet und mittels Bandpass gefiltert (siehe Gleichung 6), erfolgt nun die Visualisierung der berechneten Richtungsfilter mit Hilfe von Kugelflächenfunktionen. Für diskrete Schalldrucksignale (Mikrofonsignale) können die Ambisonics-Signale als Lösung eines Minimum-Mean-Square Error (MMSE) Problems berechnet werden [ZF19, p.141-142] als

$$\begin{pmatrix} p(\boldsymbol{\theta_1}) \\ \vdots \\ p(\boldsymbol{\theta_M}) \end{pmatrix} = \begin{bmatrix} Y_0^0(\boldsymbol{\theta_1}) & \dots & Y_N^N(\boldsymbol{\theta_1}) \\ \vdots & \ddots & \vdots \\ Y_0^0(\boldsymbol{\theta_M}) & \dots & Y_N^N(\boldsymbol{\theta_M}) \end{bmatrix} \begin{pmatrix} \gamma_0^0 \\ \vdots \\ \gamma_N^N \end{pmatrix}$$

$$\boldsymbol{p_M} = \boldsymbol{Y_N^T} \boldsymbol{\gamma_N}. \tag{7}$$

$\boldsymbol{p_M}$ ... Schalldrucksignale/Impulsantworten für alle M Mikrofonpositionen
$\boldsymbol{Y_N^T}$ ... transponierte Matrix mit den Kugelflächenfunktionen
$\boldsymbol{\gamma_N}$ ... Ambisonics-Signale Nter Ordnung

Für den Fall, dass die $(N+1)^2$ resultierenden Ambisonics-Signale nicht mit der Anzahl der Mikrofonpositionen M übereinstimmt, kann Gleichung 7 mittels Pseudo-Inversion nach $\boldsymbol{\gamma_N}$ aufgelöst werden. Die Lösung mittels Pseudo-Invertierung beinhaltet die Minimierung des quadratischen Fehlers zwischen diskreten Schalldrucksignalen $\boldsymbol{p_M}$ und dem real auftretenden Schalldruck p (Least-Squares Problem). Die Lösung mittels linksinverser Moore-Penrose-Pseudoinverse ergibt sich zu

$$\boldsymbol{\gamma_N} = \left(\boldsymbol{Y_N Y_N^T}\right)^{-1} \boldsymbol{Y_N p_M} = \left(\boldsymbol{Y_N^T}\right)^{\dagger} \boldsymbol{p_M}. \tag{8}$$

Das diesem Projekt zugrunde liegende Mikrofon-Array besteht aus $M = 64$ Mikrofonen [Hoh09]. Deshalb können für die 64 Mikrofonsignale und für Ambisonics-Signale 7. Ordnung ($N = 7$) die Pseudoinverse aus Gleichung 8 durch die Standardinverse der Matrix $\boldsymbol{Y_N^T}$ ersetzt werden.

$$\textbf{Bedingung: } M = (N+1)^2 = 64$$

$$\boldsymbol{\gamma_N} = \left(\boldsymbol{Y_N^T}\right)^{-1} \boldsymbol{p_M} \tag{9}$$

Diese Art der Berechnung von Ambisonics-Signale muss nicht zwingend auf diskrete Schalldrucksignale angewandt werden. Sie kann auch mit Filterimpulsantworten durchgeführt werden und somit auch auf die kurzzeitspektralen Richtungsfilter aus Gleichung 6. Um Richtungsfilter als zeitvariante Impulsantworten zu erhalten, müssen die Filter aus Gleichung 6 in den Zeitbereich zurück transformiert werden. Der Einfachheit halber wird auf die Phaseninformation der Richtungsfilter verzichtet und die Richtungsfilterimpulsantworten werden somit als nullphasig betrachtet. Die in Gleichung 9 angeführten Zeitsigna-

le werden in unserer Anwendung also als

$$\boldsymbol{p_M} = \mathcal{F}^{-1}\left\{\begin{pmatrix}\left|\tilde{\boldsymbol{P}}_{\tau,1}\right| \\ \vdots \\ \left|\tilde{\boldsymbol{P}}_{\tau,M}\right|\end{pmatrix}\right\} = \begin{pmatrix}\mathcal{F}^{-1}\left\{\left|\tilde{P}_{\tau,1}[0]\right|, \ldots, \left|\tilde{P}_{\tau,1}[N_{FFT}]\right|\right\} \\ \vdots \\ \mathcal{F}^{-1}\left\{\left|\tilde{P}_{\tau,M}[0]\right|, \ldots, \left|\tilde{P}_{\tau,M}[N_{FFT}]\right|\right\}\end{pmatrix} \tag{10}$$

festgelegt, wobei die Vektoren $\left|\tilde{\boldsymbol{P}}_{\tau,1}\right|$ bis $\left|\tilde{\boldsymbol{P}}_{\tau,M}\right|$ die Betragsfrequenzgänge der jeweiligen Bandpassgefilterten Richtungsfilter an den M Mikrofonpositionen beinhalten. Genaueres über die Implementierung von Gleichung 9 im vorgestellten VST-Plug-In und die vom EnergyVisualizer [RZGH20] durchgeführte Dekodierung sind in den Abschnitten 3.1.1 und 3.1.4 angeführt.

3 Implementierung

Die Implementierung basiert auf dem EnergyVisualizer der IEM Plug-Ins [RZGH20] von Daniel Rudrich. Des Weiteren wurden Codeausschnitte des zur Masterarbeit von Franck Zagala [Zag19] entwickelten Plug-Ins „SpInPlIn“ übernommen sowie eine C++Klasse zur mehrkanaligen, geblockten FFT mit Overlap von Daniel Rudrich [Rud19].

3.1 Algorithmen

Basis für alle Berechnungen ist eine blockbasierte Verarbeitung. Die Blocklänge dieser wird durch die DAW bzw. durch den Nutzer vorgegeben. Die grundlegende Verarbeitung passiert in `PluginProcessor.cpp`. Die Matrizen zur Enkodierung in die Ambisonics-Domäne werden im Konstruktor generiert. Weitere Variablen und Arraylängen werden in der `prepareToPlay`-Methode initialisiert.

Die Input-Blocks werden einem `AudioBlock`-Objekt übergeben, welches eine effiziente, pointerbasierte Struktur zur Verwaltung von Sample-Blocks darstellt. Dieses wird nun einem `ProcessContextNonReplacing`-Objekt übergeben, gemeinsam mit einem weiteren Dummy-AudioBlock. Über dieses Objekt können für den folgenden Prozessor die Daten gemeinsam mit Kontext-Informationen übergeben werden. Hier wurde die NonReplacing-Variante gewählt, damit die unveränderten Daten wieder am Ausgang des Plugins für eine allenfalls folgende Verarbeitung anliegen. Dieses Context-Objekt wird nun der `process`-Methode der `FftAndNormProcessor`-Klasse übergeben.

3.1.1 Ursignalgewinnung und Berechnung der Richtungsfilter

Die `FftAndNormProcessor`-Klasse erbt von der `OverlappingFFTProcessor`-Klasse [Rud19]. Diese Klasse verwaltet alle Buffer-Operationen für eine überlappende FFT. Hier wird bereits die FFT-Ordnung und Hopsize-Größe definiert. Als Kompromiss hinsichtlich Effizienz und Genauigkeit hat sich eine FFT-Ordnung [3] von 9 und eine Hopsize-Divider-Ordnung [4] von 2 bewährt [5]. Da hier alle 64 Kanäle transformiert werden müssen, ist dieser Vorgang vergleichsweise rechenaufwändig. Weil wir aber nur reellwertige Eingangssignale erwarten, kann die effizientere `performRealOnlyForwardTransform` verwendet werden. Aus diesen Daten im Interleaved-Format werden nun die Betragsspektren berechnet.

Die Daten werden nun entsprechend der der gewählten l^p-Norm potenziert. Im Sinne der Effizienz wird die Operation zu

$$|X_{\tau,\lambda}[k]|^p = \exp\left(p * \log\left(|X_{\tau,\lambda}[k]|\right)\right) \tag{11}$$

umgeformt. Die Daten werden nun aufsummiert und daraus als Gegenstück zu Gleichung 11 die p-te Wurzel gezogen. Der entsprechende Codeausschnitt zu FFT und Berechnung der Norm ist in Listing 1 gezeigt.

3. Die FFT-Länge entspricht 2^{Ordnung}, Ordnung 9 entspricht also 512 Punkten.
4. Die Hopsize-Divider-Ordnung bestimmt die Hopsize mit $\frac{\text{FFT-Länge}}{2^{\text{Hopsize-Divider-Ordnung}}} = \frac{512}{2^2} = 128$ Punkte
5. Bei einer Samplerate von $44.1\,\text{kHz}$ bzw. $48\,\text{kHz}$

```
110 // transform audio
111 for (int ch = 0; ch < numChIn; ++ch)
112 {
113   fft.performRealOnlyForwardTransform (fftInOutBuffer.getWritePointer
      ↪ (ch), true);
114
115   auto chPtr = fftInOutBuffer.getWritePointer (ch);
116   for (int ii = 0; ii < fftSize / 2 + 1; ++ii)  // Calculate magnitude
      ↪ response
117     chPtr[ii] = std::exp (0.5f * std::log (chPtr[2 * ii] * chPtr[2 *
      ↪ ii] + chPtr[2 * ii + 1] * chPtr[2 * ii + 1]));  // calculate
      ↪ magnitude response
118
119   memcpy (fftData.getWritePointer(ch),
      ↪ fftInOutBuffer.getReadPointer(ch), roundToInt (fftSize/2 + 1));
      ↪  // Copy data for calculating directivity-filters later on
120   for (int ii = 0; ii < fftSize / 2 + 1; ++ii)
121     chPtr[ii] = std::exp(p * std::log (chPtr[ii]));  // exponential
      ↪ part of L-Norm
122
123   FloatVectorOperations::add (lNorm.data(), chPtr, roundToInt
      ↪ (fftSize/2) + 1);  // Sum all spectrograms
124 }
125 for (int ii = 0; ii < fftSize / 2 + 1; ++ii)
126   lNorm[ii] = std::exp (std::log (lNorm[ii] / numChIn) / p);  //
      ↪ calculate p-th root for L-Norm
```

Listing 1 – FFT und Berechnung der l^p-Norm in `fftAndNormProcessor.h`.

Für die Berechnung der Richtungsfilter der diskreten Richtungen wird nun einfach der Quotient vom Betragsspektrum an der jeweiligen Mikrofonposition und der l^p-Norm gebildet und mit dem Frequenzgang des Bandpass-Skirt-Filters multipliziert. Das Ergebnis wird als Realteil im Interleaved-Format gespeichert. Da die Phaseninformation nicht von Belang ist, wird der Imaginärteil null gesetzt, wir bekommen also ein nullphasiges Ergebnis. Nun wird eine IFFT durchgeführt, damit das Signal später leicht weiterverarbeitet werden kann. Dies entspricht dem Rechenschritt, der in Gleichung 10 verdeutlicht ist. Der entsprechende Code ist in Listing 2 gezeigt.

```
127 for (int ch = 0; ch <numChIn;++ch)
128 {
129   auto chFFT = fftData.getWritePointer (ch);
130   auto dirPow = DirFiltPowBuf.getWritePointer(ch);
131   for (int ii = 0; ii<fftSize / 2 + 1; ++ii)
132   {//Calculate Directivity-Filter and Multiply with frequency response
      ↪ of BP-Skirt-Filter ==> Filtering in Frequency Domain
133     dirPow[2*ii] = (chFFT[ii]/lNorm[ii])*alpha[ii];
134     // create complex interleaved vector ==> zero-Phase ==> Imaginary
      ↪ Part is set to 0
135     dirPow[2*ii+1] = 0;
136   }
137   fft.performRealOnlyInverseTransform(DirFiltPowBuf.getWritePointer
      ↪ (ch));
138 }
```

Listing 2 – Berechnung der Richtungsfilter in `fftAndNormProcessor.h`.

3.1.2 Implementierung der Bandpassfilterung

Der im vorherigen Kapitel erwähnte Frequenzgang des Bandpass-Skirt-Filters (im Code als Variable `alpha` notiert), wird bei Veränderung der Filterparameter neu berechnet. Wird ein Filterregler betätigt wird die Funktion `setSkirtFreq()` aufgerufen. Alle Filterparameter ($f[k_c]$, β und `Bypass`) werden der Funktion übergeben. In einem ersten Schritt wird überprüft ob der Filter überhaupt aktiv geschalten ist.

Listing 3, Zeile 54 - 72:
Ist der Filter aktiv (`Bypass` == 0.0f) wird zunächst der zur eingestellten Mittenfrequenz nächstliegende FFT-Frequenzbin gesucht (Variable `skirtBin` im Code). Dies wird mit einer iterativen Überprüfung des Abstandes der FFT-Bin-Frequenzen und der eingestellten Filter-Center-Frequenz (`skirtFreq`) durchgeführt (Zeile 58 bis 65 in Listing 3). Ist der zur eingestellten Center-Frequenz naheliegenste FFT-Frequenzbin gefunden, wird der Frequenzgangsvektor (`alpha`) mit `0.0f` initialisiert ehe er in der for-Schleife (Zeile 68 -71) mit dem in Gleichung 5 definierten Frequenzgang befüllt wird. Auch hier wird die Potenz der Exponentialfunktion mittels der in Gleichung 11 angeführten rechenärmeren Variante berechnet.

Listing 3, Zeile 73 - 76:
Soll der Filter überbrückt werden (`Bypass == 1.0f`) wird der Frequenzgang des Bandpass-Skirt-Filters mit $1.0f$ gefüllt (Zeile 74 in Listing 3), somit hat dies bei der Multiplikation mit den geschätzten Richtungsfilter (Gleichung 6) keine Auswirkungen. Die Filterregler und deren Funktion werden in Kapitel 3.1.5 genauer beschrieben.

```
52 void setSkirtFreq (const float newSkirtFreq, const float beta, const
     ↪ float Bypass)
53 {
54   if (Bypass == 0.0f){
55     skirtFreq = newSkirtFreq;
56     //find nearest bin to skirtFreq
57     float minOffset = 100.0f;
58     for (int k = 0; k < frequencies.size(); k++)
59     {
60       if (fabs (frequencies[k] - skirtFreq) < minOffset)
61       {
62         skirtBin = k;
63         minOffset = fabs (frequencies[k] - skirtFreq);
64       }
65     }
66     // create alpha vector for BP-skirt filtering
67     std::fill(alpha.begin(),alpha.end(),0.0f);
68     for (int k = 0; k < alpha.size(); k++)
69     {// alpha contains frequency-Response of BP-Skirt filter
70       alpha[k] =
     ↪ std::exp(fabs(k-skirtBin)*std::log(std::exp(-1.0f/beta)));
71     }
72   }
73   else{//bypass filter ==> set BP-Frequency response to 1 for all
     ↪ frequencies
74     std::fill(alpha.begin(),alpha.end(),1.0f);
75   }
76 }
```

Listing 3 – Berechnung des Bandpass-Skirt-Filter-Frequenzganges in `fftAndNormProcessor.h`

3.1.3 Ambisonische Enkodierung

Die Enkodierung der zeitvarianten Richtungsfilter in Ambisonics Koeffizienten mittels Kugelflächenfunktionen erfolgt, wie in Gleichung 9 notiert, mittels Multiplikation mit einer invertierten Matrix.

Listing 4, Zeile 69 - 77:

Da die Enkodierungsmatrix nur einmal berechnet werden muss, wird diese im Konstruktor des `DirectivityVisualizerAudioProcessor` der Datei `PluginProcessor.cpp` vorbereitet. Weil in späterer Folge beim Enkodieren das Matrix-Vektor-Produkt mittels der `addFrom()` Methode durchgeführt wird, sollte die invertierte Matrix als Array of Arrays vorliegen. Deshalb wird in den Zeilen 69-72 von Listing 4 das 64×64 Elemente lange Array of Arrays `SH1` geleert. Danach wir die in der Datei `efficientSHvanilla.cpp` enthaltene Methode `SHEval()` aufgerufen. Die `efficientSHvanilla.cpp` Implementierung ermöglicht effiziente Auswertungen der Kugelflächenfunktionen an gegebenen kartesischen Koordinaten [Slo13] (Zeile 76 in Listing 4). Die dazu notwendigen kartesischen

Mikrofonpositionen (im Code `MicArrayXPos`, `MicArrayYPos` und `MicArrayZPos`) wurden als Vektoren in Datei `MicArrayXYZ.h` vermerkt. Die Datei `MicArrayXYZ.h` wird im Header-File `PluginProcessor.h` eingebunden und die Variablen können so in der Datei `PluginProcessor.cpp` referenziert werden. Das dadurch erhaltene Array of Arrays `SH1` entspricht nun der Matrix $\boldsymbol{Y_N}$ aus Gleichung 7. Um die korrekte Enkodierungsmatrix zu erhalten muss diese also noch dementsprechend transponiert und invertiert werden.

Listing 4, Zeile 78 - 98:

Zur Transposition und Inversion der Enkodierungsmatrix wird die Eigen-Library herangezogen [GJ+10]. Eine `Eigen`-Matrix wird erstellt (Zeile 79, Listing 4), die mit Hilfe von 2 verschachtelten for-Schleifen aus dem Array of Arrays `SH1` befüllt wird (Zeilen 80 bis 86 in Listing 4). Mit der In-Place-Methode `.transposeInPlace()` der `Eigen`-Matrix wird die Matrixtransposition durchgeführt (Listing 4, Zeile 88). Die Matrix `MatToInv` im Code entspricht nun der Matrix $\boldsymbol{Y_N^T}$ aus Gleichung 7. Der letzte Schritt ist nun die Inversion, diese wird mit der Methode `.inverse()` durchgeführt. Die Variable `InvMat` im Code entspricht nun der fertigen Enkodierungsmatrix $(\boldsymbol{Y_N^T})^{-1}$ (Listing 4, Zeile 90). Mit zwei verschachtelten for-Schleifen wird nun wiederum die korrekte Enkodierungsmatrix in das Array of Arrays `SH1` zurück geschrieben (Listing 4, Zeile 92 bis 98).

```
69 for (int ii=0; ii< nMics; ++ii)
70 {
71   FloatVectorOperations::clear(SH1[ii], 64);
72 }
73 // calculate transposed encoder-matrix
74 for (int micCh = 0; micCh < nMics; ++micCh)
75 {
76   SHEval (7, MicArrayXPos[micCh], MicArrayYPos[micCh],
     ↪ MicArrayZPos[micCh], SH1[micCh]);
77 }
78 // copy encoder matrix to Matrix-Class of Eigen-Lib to invert it.
79 Eigen::MatrixXf MatToInv(nMics,64);
80 for (int ii = 0; ii < nMics; ++ii)
81 {
82   for (int jj = 0; jj < 64; ++jj)
83   {
84     MatToInv(ii,jj) = SH1[ii][jj];
85   }
86 }
87 // Transpose matrix
88 MatToInv.transposeInPlace();
89 // Invert Matrix
90 auto InvMat = MatToInv.inverse();
91 // copy inverted encoder Matrix back to Array of Arrays.
92 for (int ii = 0; ii < nMics; ++ii)
93 {
94   for (int jj = 0; jj < 64; ++jj)
95   {
96     SH1[ii][jj] = InvMat(ii,jj);
97   }
98 }
```

Listing 4 – Ambisonics Enkodierung mittels Kugelflächenfunktionen, Enkodierung der nullphasigen Richtungsfilterimpulsantworten in `PluginProcessor.cpp`

Die weitere Verarbeitung der Enkodierung geschieht in der ersten Hälfte der `processBlock` Methode von `PluginProcessor.cpp`.

Listing 5, Zeile 177 - 193:
Nachdem in Zeile 177 die in Kapitel 3.1.1 beschriebene Signalverarbeitung auf einen 64 kanäligen Signalblock der Eingangsdaten angewandt wurde, wird in Zeile 178 der verarbeitete Audio-Buffer mit den nullphasigen Richtungsfilterimpulsantworten aus `fftAndNormProcessor.h` übergeben. Es werden dann wiederum zwei verschachtelte for- Schleifen gestartet. In der ersten for-Schleife wird eine Kopie des Audio-Buffers mit den nullphasigen Richtungsfilterimpulsantworten erstellt und der ursprünglich übergebene Buffer (`DirFiltPowBuf`) geleert (Listing 5, Zeile 183-185). Mit der AudioBuffer-Methode `.addFrom()` wird innerhalb der zweiten for-Schleife die eigentliche Enkodierung der nullphasigen-Richtungsfilterimpulsantworten mittels Enkodierungsmatrix (oder Array of Arrays `SH1`) durchgeführt. Die nun enkodierten nullphasigen ambisonischen Richtungsfilterimpulsantworten werden wieder zurück in den Audio-Buffer `DirFiltPowBuf` geschrieben. Sind beide for-Schleifen durchgelaufen wurde das in Gleichung 9 angeführte Matrix-Vektor-Produkt berechnet.

```
177 normProc.process (context);
178 auto DirFiltPowBuf = normProc.getDirPowBuf();
179
180 // SH-Encoding on All Microphone-Signals/Buffer-Channels
181 for (int i = 0; i < 64; ++i)
182 {
183   bufferCopy.copyFrom (i, 0, DirFiltPowBuf.getReadPointer (i),
      ↪ DirFiltPowBuf.getNumSamples());
184 }
185 DirFiltPowBuf.clear();
186 for (int micCh = 0; micCh < nMics; ++micCh)
187 {
188   auto inpReadPtr = bufferCopy.getReadPointer(micCh);
189   for (int ch = 0; ch < 64; ++ch)
190   {// apply SH-Encoding ==> Dot-Product can be realized with addFrom
      ↪ for each channel/microphone-Signal
191     DirFiltPowBuf.addFrom(ch,0,
      ↪ inpReadPtr,DirFiltPowBuf.getNumSamples(),SH1[micCh][ch]);
192   }
193 }
```

Listing 5 – Ambisonics Enkodierung mittels Kugelflächenfunktionen, Erstellung der Enkodermatrix in `PluginProcessor.cpp`

Die auf die weiteren nullphasigen ambisonischen Richtungsfilterimpulsantworten angewendeten Verarbeitungsschritte sind bereits in ähnlicher Form im EnergyVisualizer aus [RZGH20] implementiert. Diese werden in Abschnitt 3.1.4 näher behandelt.

3.1.4 Dekodierung und Gewichtung

Für die Visualisierung der Richtmuster wurde das Konzept der EnergyVisualizers übernommen. Hierbei wird ein Ambisonics-Signal auf 480 virtuelle Lautsprecher dekodiert, die RMS-Werte berechnet, mit der max-r_E Gewichtung beaufschlagt und als Energieverteilung über eine Hammer-Aitoff Projektion im Format einer „Weltkarte“ dargestellt. Die Dekodierung und Gewichtung der nullphasigen ambisonischen Richtungsfilterimpulsantworten passieren in der zweiten Hälfte der `proccesBlock` Methode in `PluginProcessor.cpp` und sind in Listing 6 dargestellt. Die Visualisierung erfolgt mittels OpenGL.

```
194 if (! doProcessing.get() && !
    ↪ oscParameterInterface.getOSCSender().isConnected())
195   return;
196
197 const int L = DirFiltPowBuf.getNumSamples();
198 const int workingOrder = jmin (isqrt (DirFiltPowBuf.getNumChannels())
    ↪ - 1, int (floor (sqrt (input.getMaxSize()) - 1)));
199
200 const int nChAmbisonics = squares[workingOrder+1];
201 copyMaxRE (workingOrder, weights.data());
202 FloatVectorOperations::multiply (weights.data(),
    ↪ maxRECorrection[workingOrder] * decodeCorrection
    ↪ (workingOrder), nChAmbisonics);
203
204 const float oneMinusTimeConstant = 1.0f - timeConstant;
205 for (int i = 0; i < nSamplePoints; ++i)
206 {
207   FloatVectorOperations::copyWithMultiply (sampledSignal.data(),
    ↪ DirFiltPowBuf.getReadPointer (0), decoderMatrix(i, 0) *
    ↪ weights[0], DirFiltPowBuf.getNumSamples());
208   for (int ch = 1; ch < nChAmbisonics; ++ch)
209     FloatVectorOperations::addWithMultiply (sampledSignal.data(),
    ↪ DirFiltPowBuf.getReadPointer (ch), decoderMatrix(i, ch) *
    ↪ weights[ch], L);
210
211   // calculate rms
212   float sum = 0.0f;
213   for (int i = 0; i < L; ++i)
214   {
215     const auto sample = sampledSignal[i];
216     sum += sample * sample;
217   }
218
219   rms[i] = timeConstant * rms[i] + oneMinusTimeConstant * std::sqrt
    ↪ (sum / L);
220 }
```

Listing 6 – Ambisonics Dekodierung in `PluginProcessor.cpp`

Das User-Interface inkl. Regler und die Energievisualisierung werden im Kapitel 3.1.5 genauer beschrieben.

3.1.5 User-Interface und Visualisierung

Das User-Interface mit Eingabereglern und Energievisualisierung ist in Abbildung 2 dargestellt.

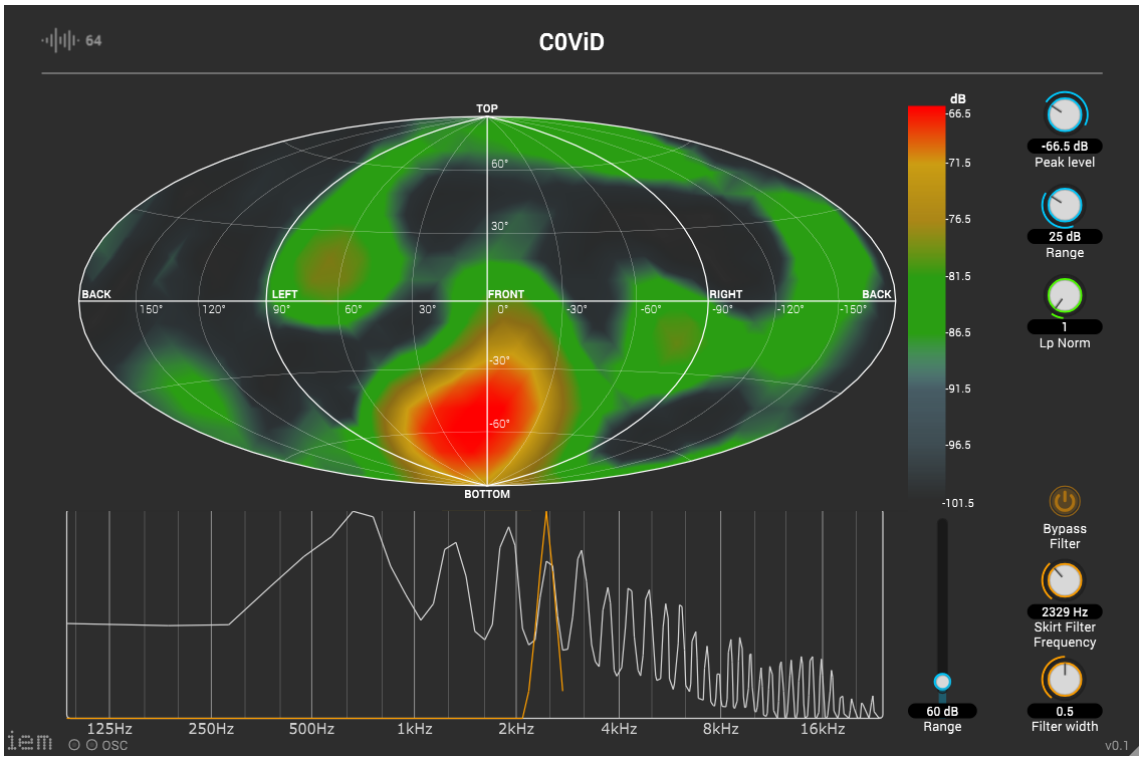

Abbildung 2 – GUI des entworfenen VST-Plugins

Eingaberegler:
Am rechten Rand befindet sich eine Regler-Sektion, in welcher sowohl Einstellungen zur Energievisualisierung als auch zur spektralen Visualisierung der l^p-Norm getroffen werden können. Die Regler sind farblich gruppiert. Die ersten beiden Regler in blau („Peak level“ und „Range“) dienen zur Einstellung der Energievisualisierung. „Peak level“ kann zwischen $-100 \ldots 10\,\mathrm{dB}$ und „Range“ zwischen $10 \ldots 60\,\mathrm{dB}$ über einen Regler eingestellt werden. „Peak level“ legt fest, bei welchem Pegel die obere Grenze des Farbbalkens liegt. „Range“ legt die dargestellte Dynamik fest, es kann bestimmt werden, welche Pegelunterschiede in der Hammer-Aitoff-Energievisualisierung erfasst werden. Über einen Klick auf den Farbbalken kann auch zwischen den Farbkartierungen „Parula“ und „Jet“ umgeschaltet werden.

Der grüne Regler gibt an, in welcher Ordnung die l^p-Norm berechnet wird, siehe Varibale p in Gleichung 2. Ein hoher Wert für p spiegelt sich auch in der Energievisualisierung wieder, in Form einer Pegelabschwächung. In der normalisierten Frequenzspektrumsdarstellung der l^p-Norm (im unteren Drittel des User-Interfaces) führt eine höhere Ordnung

in der Regel zu höheren Spitzen im höherfrequenten Bereich.

Die orange eingefärbten Regler im unteren Drittel der Reglersektion steuern die Filterparameter. Der On/Off Button ermöglicht ein Überbrücken der Bandpass-Filterflanke. Der „Skirt Filter Frequency"-Regler steuert die Mittenfrequenz des Bandpass-Filters. Mit dem „Filter width"-Regler kann die Filterbreite gesteuert werden. Dieser steuert direkt den Parameter β und kann im Wertebereich $0 \ldots 1$ festgelegt werden. Vergleicht man die Regler mit den Parameter der Gleichung 5, wird mit „Skirt Filter Frequency" der Frequenzbin k_c festgelegt. „Skirt Filter Frequency" kann Werte im Bereich zwischen $20 \ldots 6500\,\mathrm{Hz}$ annehmen. Wird die Bandpassfilter-Mittenfrequenz höher als $6500\,\mathrm{Hz}$ gewählt, sind die resultierenden Pegel so niedrig, dass sie in den numerischen Unsicherheiten der verwendeten Zahlenformate verschwinden. Mit der derzeitigen Implementierung können die bandpassgefilterten Richtungsfilter also nur bis $6500\,\mathrm{Hz}$ sinnvoll visualisiert werden.

Die Frequenzspektrumsdarstellung der l^p-Norm zeigt sowohl den Frequenzgang der berechneten l^p-Norm als auch den Frequenzgang des Bandpass-Skirt-Filters $\boldsymbol{\alpha}_\beta[k_c]$. Da die Betragsfrequenzgangsdarstellung einen wesentlichen Teil der Implementierung darstellt, geht Abschnitt 3.1.6 noch näher darauf ein.

3.1.6 Frequenzgangsdarstellung des Ursignals

Zur Darstellung des Betragsfrequenzgangs des Ursignals $|U_\tau[k]|$ wurde die Klasse `SpectrumVisualizerComponent` erstellt. In ihr wurden alle nötigen Berechnungen und die Darstellung selbst definiert. So kann im Header-File `PluginEditor.h` eine Komponente dieser Klasse erstellt werden. Anschließend wurden im zugehörigen .cpp-File `PluginEditor.cpp` mittels der Methoden dieser Klasse die Werte des Betragsspektrums übergeben und die Visualisierung gestartet.
Nachdem die Position der Visualisierung gesetzt wurde, wird im Konstruktor einmalig die Samplerate übergeben (Listing 7, Zeile 151). Diese wird mittels der Methode `.getSampleRate()` des `juce_AudioProcessor.h` abgerufen. Anschließend wird in den Zeilen 152 bis 154 die Zykluszeit auf jedes fünften Frame gesetzt.

```
150 addAndMakeVisible(&spectVisualizer);
151 spectVisualizer.setSampleRate(p.getSampleRate());
152 double fs = p.getSampleRate();
153 auto timerDur = roundToInt((double(scopeSize)/fs)*1000*5);
154 startTimer(timerDur);
```

Listing 7 – Übergabe der Samplingrate in `PluginEditor.cpp`

Die `timerCallback()`-Funktion wird nach jeder Zykluszeit ausgeführt (Listing 8). In den Folgenden Absätzen wir auf diese wiederkehrenden Befehle und deren Bedeutung näher eingegangen.

Datenübertragung:

Die Datenübergabe der beiden Spektren wurde im Timer-Callback definiert (Listing 8, Zeilen 315 bis 318). Mit den Methoden `getSpectrum` und `getSkirt` der Klasse `FftAndNormProcessor` können die l^p-Norm und das Spektrum des Bandpassfilters aufgerufen werden. Sie müssen als konstante Parameter definiert werden, um sie an das Visualisierungsobjekt `spectVisualizer` übergeben zu können. Anschließend werden in der Methode `setData` die Daten auf die dort definierten Variablen gespeichert und die Spektrumslänge errechnet(Listing 9, Zeilen 162 bis 164). In der darauf folgenden for-Schleife werden NaN-Werte durch kleine float-Beträge getauscht, um unterbrochene Spektrallinien zu vermeiden.

```
315 FftAndNormProcessor& normProcessor = processor.getNormProcessor();
316 const std::vector<float> spect = normProcessor.getSpectrum();
317 const std::vector<float> SkirtSpect = normProcessor.getSkirt();
318 spectVisualizer.setData(spect,SkirtSpect);
319
320 spectVisualizer.setFrequencyVector();
321 spectVisualizer.drawNextSpectFrame();
322 repaint();
```

Listing 8 – `timerCallback()` in `PluginEditor.cpp`

```
158 void setData(const std::vector<float> data,const std::vector<float>
    ↪ Skirtdata)
159 {
160   spectData = data;
161   spectSize = data.end() - data.begin();
162   SkirtSpectData = Skirtdata;
163   for (int i = 0; i < spectSize; i++)
164   {
165     if (isnan(spectData[i]))
166     {
167       spectData[i] = 0.01f;
168     }
169
170     if (isnan(SkirtSpectData[i]))
171     {
172       SkirtSpectData[i] = 0.01f;
173     }
174   }
```

Listing 9 – `setData()` in `SpectrumVisualizerComponent.h`

Erstellen des Frequenzvektors:

Der Quelltext der Methode `setFrequencyVector` wird in Listing 10 gezeigt. Diese Methode wurde ebenfalls in der Callback-Funktion des `PluginEditors` aufgerufen (Listing 8, Zeile 320), da hier die Länge des Spektrums bekannt sein muss. Mit der Abfrage nach `alreadySet` wird eine Mehrfachberechnung des Vektors vermieden und so Rechenleistung gespart (Zeile 246). Zuerst wird der Frequenzvektor als linearer Vektor mit der Länge des übergebenen Ursignalspektrums (`spectSize` $= 256$) erstellt und berechnet (Listing

10, Zeilen 248 bis 255). Anschließend kann durch die Exponentialfunktion der Zeile 257 ein logarithmischer Frequenzverktor mit der Länge `scopeSize` $= 512$ erstellt werden.

```
241 void setFrequencyVector()
242 {
243
244   if (!alreadySet)
245   {
246     frequencyVector.resize(spectSize);
247     float deltaFequency = (float)sampleRate / (float)spectSize / 2.0f;
248     for (int i = 0; i < spectSize; i++)
249     {
250       frequencyVector[i] = (1.0f + (float)i) * deltaFequency;
251     }
252
253     for (int i = 0; i < scopeSize; ++i)
254     {
255       skewedFrequencyVector[i] = round(frequencyVector.front() *
    ↪ std::exp((float)i / (float)scopeSize *
    ↪ (std::log(frequencyVector.back() / frequencyVector.front()))));
256     }
257
258     alreadySet = true;
259   }
260 }
```

Listing 10 – `setFrequencyVector()` in `SpectrumVisualizerComponent.h`

Erstellen der Spektrumsdarstellung:

Der nächste Befehl in der `callback`-Funktion des `PluginEditor.cpp`-Files (Listing 8, Zeile 321) ist `drawNextSpectFrame`. Da die Spektrumsanzeige keine absolute Pegelrichtigkeit benötigt, sondern zur qualitativen Frequenzanzeige genutzt wird, werden die übergebenen Spektren der STFT-Frames auf ihr jeweiliges Maximum normiert. Anschließend werden der Minimal- und Maximalwert der Darstellung, mittels dem am Range-Regler eingestellten Wertes, berechnet (Listing 11, Zeilen 219 und 220). Aufgrund der Normierung der Spektren beträgt der Wert der Variable `peakLevel` konstant Null. Nun werden die Spektraldaten, die entlang einer linearen Frequenzachse vorliegen (`frequencyVector`), in eine logarithmische Frequenzdiskretisierung übertragen. Zur Übertragung auf die logarithmische Frequenzdiskretisierung wird der, noch leere Datenvektor durschritten. Die Übertragung erfolgt dann mittels „Sample-And-Hold“. „Sample-And-Hold“ bedeutet in diesem Zusammenhang, dass der erste zu einem linear angeordneten Frequenzbin zugehörige spektrale Wert solange gehalten wird, bis die Frequenz des logarithmisch angepassten Frequenzbins die Frequenz des linear angepassten Frequenzbins überschreitet (Listing 11, Zeilen 222 bis 228). Wird die Frequenz des logarithmisch angepassten Frequenzbins größer als die der linear angepassten Frequenzachse wird ein weiteres Mal der gehaltene Wert geschrieben, ehe anschließend der Index des linearen Frequenz-Vektors `linIndex` inkrementiert wird (Zeilen 229 bis 233). Durch dieses Vorgehen entsteht eine treppenförmige Funktion. Die normalisierten Daten (`spectData`, `SkirtSpectData` $\in [0, 1]$) werden logarithmiert (`dB(spectData)`, `dB(SkirtSpectData)` $\in]-\infty, 0]$) und mit dem Befehl

`jmap` in einen Wertebereich zwischen 0 und 1 projiziert. Dies vereinfacht die weitere Verarbeitung als Darstellung im Plug-In.

```
217 float mindB = peakLevel - dynamicRange;
218 float maxdB = peakLevel;
219 int linIndex = 0;
220 for (int i = 0; i < scopeSize; i++)
221 {
222   if (frequencyVector[linIndex] >= skewedFrequencyVector[i] &&
      ↪ linIndex <= spectSize)
223   {
224     scopeData[i] =
      ↪ jmap(Decibels::gainToDecibels(std::abs(spectData[linIndex]),
      ↪ mindB), mindB, maxdB, 0.0f, 1.0f);
225     SkirtScopeData[i] =
      ↪ jmap(Decibels::gainToDecibels(std::abs(SkirtSpectData[linIndex]),
      ↪ mindB), mindB, maxdB, 0.0f, 1.0f);
226   }
227   else
228   {
229     scopeData[i] =
      ↪ jmap(Decibels::gainToDecibels(std::abs(spectData[linIndex]),
      ↪ mindB), mindB, maxdB, 0.0f, 1.0f);
230     SkirtScopeData[i] =
      ↪ jmap(Decibels::gainToDecibels(std::abs(SkirtSpectData[linIndex]),
      ↪ mindB), mindB, maxdB, 0.0f, 1.0f);
231     linIndex++;
232   }
233 }
```

Listing 11 – `drawNextSpectFrame()` in `SpectrumVisualizerComponent.h`

Anzeigen der Spektrumsdarstellung:
Der letzte Aufruf in der `callback`-Funktion triggert die `paint`-Methode des `Plugin Editors` (Listing 8, Zeile 322). In ihr wird die `drawSpect`-Methode des Visualisierungsobjekts aufgerufen, welche das Spektrum zeichnet. Dieser Aufruf wird in der `paint`-Methode getätigt, da hier die `Graphics`-Klasse des Plugins übergeben werden kann. Mit ihr werden die Farben definiert und die Linien gezogen. In Listing 12 ist die for-Schleife zu sehen, in welcher die Spektrallinien des Ursignals gezeichnet werden. Bei dieser Programmdarstellung wurde für einen kompakten Überblick der Teil, der den Skirt-Filter zeichnet, nicht gezeigt. Solange die aufeinanderfolgenden Magnituden der logarithmischen Spektraldarstellung denselben Wert aufweisen, wird der Index `indexPost` inkrementiert. Sobald sie sich unterscheiden wird eine Linie vom ersten Element `indexPre` $= 0$ zum aktuellen Element `indexPost` definiert. So werden ansteigende/abfallende Verbindungslinien zwischen den Spektralwerten gezeichnet, an Stelle der stückweise konstanten Treppenfunktion. Anschließend werden beide Indizes aktualisiert. Dieser Vorgang wird bis zum Ende des logarithmierten Spektrums wiederholt. Beim definieren des Start- und Endpunktes der Linie wird der jeweilige Wert der des Betrags auf die Höhe des aktuellen Darstellungsfensters projiziert (Listing 12, Zeilen 118 und 119).

```
107 for (int i = 1; i < scopeSize; ++i)
108 {
109   int ii = i - 1;
110   if (scopeData[ii] == scopeData[i])
111   {
112     indexPost = i;
113   }
114   else
115   {
116     Line newSpectLine;
117     g.setColour(Colours::lightgrey);
118     newSpectLine.setStart((float)jmap(indexPre, 0, scopeSize - 1, 0,
      ↪ width) + (float)area.getX(), jmap(scopeData[indexPre], 0.0f,
      ↪ 1.0f, (float)height - 2.0f, 0.0f) + (float)area.getY() + 2.0f);
119     newSpectLine.setEnd((float)jmap(indexPost, 0, scopeSize - 1, 0,
      ↪ width) + (float)area.getX(), jmap(scopeData[indexPost], 0.0f,
      ↪ 1.0f, (float)height - 2.0f, 0.0f) + (float)area.getY() + 2.0f);
120     g.drawLine(newSpectLine);
121     indexPre = i - 1;
122     indexPost = i;
123   }
```

Listing 12 – `drawSpect()` in `SpectrumVisualizerComponent.h`

3.2 Performance

Im Hintergrund laufen bei dieser Implementierung einige teils aufwändige Berechnungen. Alleine die FFT und IFFT aller 64 Eingangskanäle für jeden (überlappenden) Block und die openGL-Visualisierung der Energieverteilung verlangt auch aktuellen Rechnern einiges an Leistung ab. So konnte beobachtet werden, dass es selbst auf aktuellen Windows-PCs (Intel Core i7-8550U: $1.8 \ldots 4\,\mathrm{GHz}$ Quad Core, 16GB RAM, Windows 10 Pro 64bit) während der Benützung zu Aussetzern kommt. Auf rund 5 Jahre alten macOS-Rechnern mit vergleichbarer Leistung (Intel Core i7-4770HQ: $2.2 \ldots 3.4\,\mathrm{GHz}$ Quad Core, 16GB RAM, macOS 10.15.6) lief das Plug-In in Tests jedoch ohne Probleme.

Als Ursache wird vermutet, dass der Betriebssystem-Scheduler demp Plug-In auf macOS dem Plug-In eine höhere Priorität zuordnet und diesem so mehr Rechenleistung im richtigen Augenblick zur Verfügung stellt. Eine alternative Ursache könnte in der automatisch von MacOs parallelisierten Ausführung von Prozessen liegen, während Parallelisierung innerhalb eines Prozesses unter Windows explizite Parallelprogrammierung erfordert. Diese Theorie wird von der macOS-Aktivitätsanzeige untermauert, die auf dem Testgerät eine Auslastung von rund 130 % (kumuliert, bezogen auf einen Kern) aufweist, d.h. der Prozess kann nur mittels Multithreading fehlerfrei ausgeführt werden.

A Appendix

A.1 Mikrofonpositionen des Mikrofon-Arrays (auf 1m normiert)

Mic. Nr.	X / m	Y / m	Z / m	Mic. Nr.	X / m	Y / m	Z / m
1	0,2588	0,0000	-0,9659	33	0,8309	-0,5439	-0,1173
2	-0,0006	1,0000	0,0022	34	0,6995	-0,4775	-0,5317
3	0,1101	0,5216	-0,8461	35	0,7643	-0,5653	0,3103
4	-0,4644	-0,1058	0,8793	36	0,1093	-0,9758	0,1893
5	-0,7290	0,2024	0,6539	37	-0,4104	-0,8142	-0,4106
6	0,8966	0,2282	0,3794	38	0,9299	-0,1544	-0,3338
7	-0,8636	0,0464	-0,5020	39	-0,0350	-0,1293	0,9910
8	-0,3032	-0,9526	0,0268	40	0,8284	0,3165	-0,4622
9	0,3622	0,9022	-0,2343	41	-0,4004	0,8744	-0,2740
10	0,3271	0,7061	0,6280	42	-0,6947	-0,7126	-0,0977
11	-0,1766	0,1623	-0,9708	43	-0,4163	0,8861	0,2037
12	-0,2833	0,2695	0,9204	44	-0,0234	-0,7535	-0,6570
13	-0,0734	0,6066	0,7916	45	0,6277	0,3132	0,7127
14	-0,0477	-0,3219	-0,9456	46	0,1015	-0,5546	0,8259
15	0,3608	-0,4676	-0,8070	47	-0,5881	-0,7201	0,3682
16	-0,1743	-0,8380	0,5171	48	-0,0052	0,8660	-0,5000
17	-0,9706	-0,1861	-0,1524	49	-0,3591	-0,5239	0,7724
18	-0,9625	-0,0156	0,2710	50	-0,3981	-0,5330	-0,7466
19	0,4645	0,6772	-0,5707	51	-0,5000	0,6186	0,6060
20	0,5221	-0,4883	0,6992	52	-0,7391	-0,2924	0,6067
21	-0,5646	0,2665	-0,7812	53	0,9631	0,2638	-0,0538
22	0,7833	-0,1619	0,6002	54	0,4028	0,8927	0,2022
23	0,5068	0,3277	-0,7974	55	0,7040	0,6171	0,3516
24	0,5228	-0,8524	0,0149	56	0,9779	-0,1436	0,1518
25	-0,7149	0,5195	-0,4681	57	-0,0619	0,8880	0,4556
26	0,4070	-0,7906	-0,4575	58	0,6707	-0,0652	-0,7389
27	-0,8757	-0,4432	0,1916	59	0,7437	0,6546	-0,1357
28	0,0696	-0,9665	-0,2472	60	0,2272	0,2939	0,9284
29	-0,8233	0,4663	0,3235	61	-0,7204	0,6924	-0,0393
30	0,4203	-0,1061	0,9011	62	0,3351	-0,8020	0,4944
31	-0,7664	-0,4208	-0,4852	63	-0,5163	-0,1446	-0,8441
32	-0,9454	0,3135	-0,0890	64	-0,3201	0,6299	-0,7076

Tabelle 1 – kartesische Mikrofonpositionen des verwendeten Mikrofonarrays

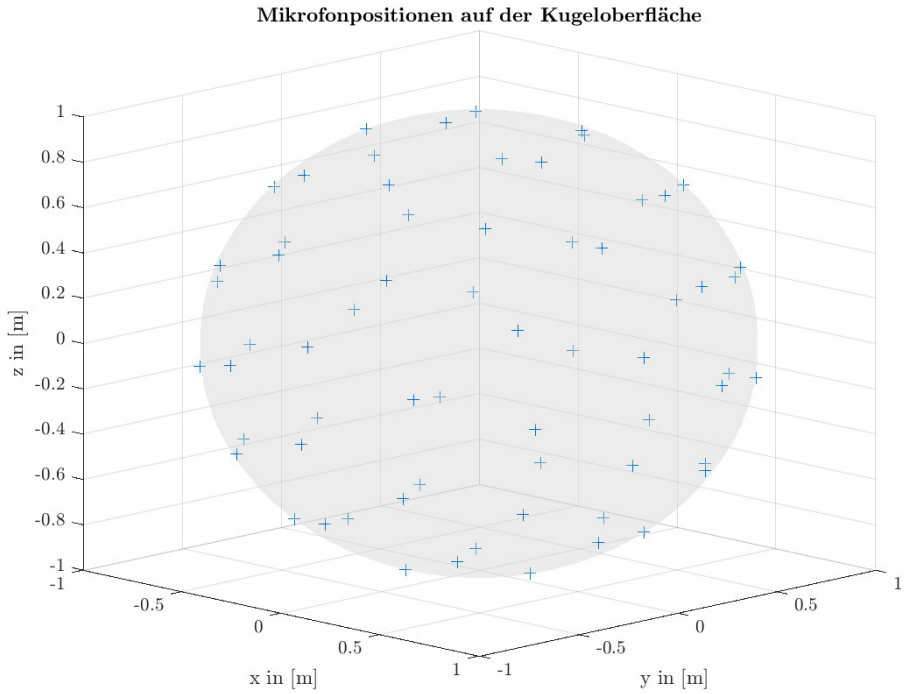

Abbildung 3 – Verteilung der Mikrofonposition auf der Kugeloberfläche

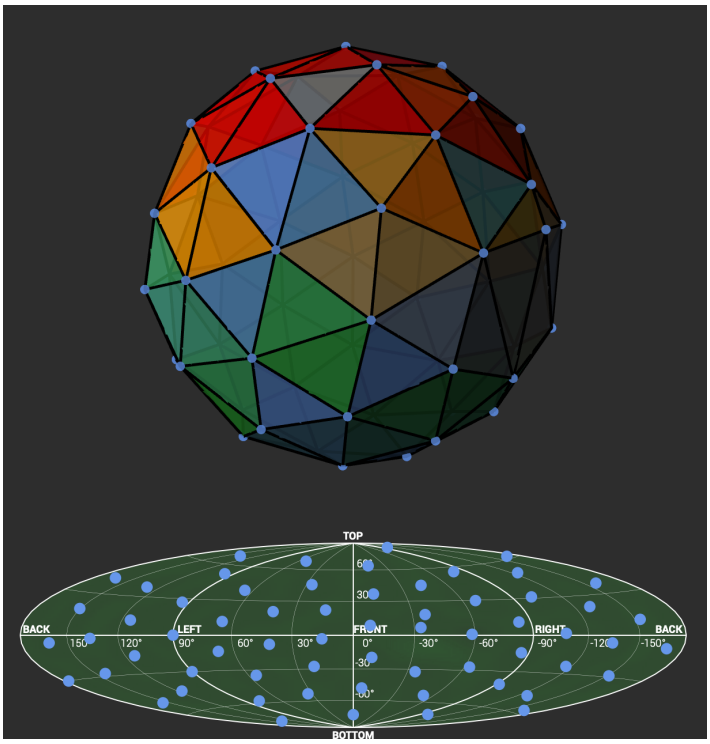

Abbildung 4 – Schematische Darstellung des Mikrofon-Arrays inkl. Hammer-Aitoff Projektion

Literatur

[BB19] BLANDIN, Rémi ; BRANDNER, Manuel: Influence of the vocal tract on voice directivity. In: OCHMANN, M. (Hrsg.) ; VORLÄNDER, M. (Hrsg.): Proceedings of the 23rd International Congress on Acoustics (ICA 2019 Aachen). Aachen (Deutschland), November 2019. – procedure: without peer reviewing

[BFR18] BRANDNER, Manuel ; FRANK, Matthias ; RUDRICH, Daniel: DirPat—Database and Viewer of 2D/3D Directivity Patterns of Sound Sources and Receivers. In: Audio Engineering Society Convention 144, 2018

[BM10] BAUMGARTNER, Robert ; MESSNER, Elmar: Auswirkung der Abstrahlcharakteristik auf die Klangfarbe von Querflöten und Saxofonen. 2010

[BMKF20] BRANDNER, Manuel ; MEYER-KAHLEN, Nils ; FRANK, Matthias: Directivity pattern measurement of a grand piano for augmented acoustic reality. In: LANGER, S. (Hrsg.) ; PEISSIG, J. (Hrsg.): Fortschritte der Akustik, DAGA (Fortschritte der Akustik, DAGA). Berlin (Deutschland), April 2020. – procedure: without peer reviewing

[Eng20] ENGE, Kajetan S.: Listening experiment on the plausibility of acoustic modeling in virtual reality, Diplomarbeit, April 2020. `https://phaidra.kug.ac.at/o:104550`

[GJ+10] GUENNEBAUD, Gaël ; JACOB, Benoît u. a.: Eigen v3. http://eigen.tuxfamily.org, 2010

[Hoh09] HOHL, Fabian: Kugelmikrofonarray zur Abstrahlungsvermessung von Musikinstrumenten, Diplomarbeit, 2009. `http://phaidra.kug.ac.at/o:2195`

[Rud19] RUDRICH, Daniel: OverlappingFFTProcessor. `https://github.com/DanielRudrich/OverlappingFFTProcessor`. Version: September 2019. – Zuletzt abgerufen am 03. Juli 2020

[RZGH20] RUDRICH, Daniel ; ZOTTER, Franz ; GRILL, Sebastian ; HUBER, Markus: IEM Plug-in Suite. `https://plugins.iem.at/`. Version: April 2020. – Zuletzt abgerufen am 02. Juli 2020

[Slo13] SLOAN, Peter-Pike: Efficient Spherical Harmonic Evaluation. In: Journal of Computer Graphics Techniques (JCGT) 2 (2013), September, Nr. 2, 84–83. `http://jcgt.org/published/0002/02/06/`. – ISSN 2331–7418

[SP20] STORER, Jules ; PACE: JUCE - Jules' Utility Class Extensions. `https://juce.com/`. Version: Februar 2020. – 5.4.7, zuletzt abgerufen am 02. Juli 2020

[Zag19] ZAGALA, Franck: Optimum-phase primal signal and radiation-filter modelling of musical instruments. Januar 2019

[ZF19] ZOTTER, F. ; FRANK, M.: Ambisonics - A Practical 3D Audio Theory for Recording, Studio Production, Sound Reinforcement, and Virtual Reality. first edition. Heidelberg (Deutschland) : Springer, 2019

[Zot09] ZOTTER, Franz: Analysis and Synthesis of Sound-Radiation with Spherical Arrays, Diss., September 2009. http://phaidra.kug.ac.at/o:5619