Darstellungs- und Analysewerkzeug für sphärische Richtwirkungsdaten

Tonignieurs-Projektarbeit, SE

Korbinian Georg Maria Wegler

Betreuung: DI Christian Schörkhuber

Beurteilung: Ass.Prof. DI Dr.rer.nat. Franz Zotter

Graz, 10. März 2020

institut für elektronische musik und akustik

Zusammenfassung

Richtwirkungsdaten treten als Betrag und Phase in mehreren Dimensionen in unzähligen akustischen Anwendungen auf. Als Ambisonics-Signale, Messungen mit Kugelmikrofonanordnungen und Kugellautsprecheranordnungen oder Außenohrübertragungsfunktionen (HRTF) existieren sie als vieldimensionale Datensammlungen, die nicht einfach zu sichten sind. Um die unübersichtliche Datenmenge zu überblicken, werden unter anderem Ballondarstellungen und Polardiagramme verwendet. Im Rahmen dieser Arbeit entsteht eine graphische Benutzeroberfläche (GUI) in python™, die eine Darstellung von Betrag und Phase bestehender Richtwirkungsdaten als Ballondarstellung (in Azimut- und Zenitwinkel), Polardarstellung (in Polarwinkel und einstellbarer Schnittebene) und als Weltkartendarstellung (Mollweide-Projektion) ermöglicht. Beispielhaft sollen mit Hilfe der entstandenen Benutzeroberfläche verschiedene Datensätze analysiert und verglichen werden.

In acoustics there are many uses for directivity data, typically expressed in multidimensional phase and amplitude, in acoustics. Ambisonics data, measurements conducted with spherical microphone arrays and spherical loudspeaker arrangements create a big amount of multidimensional data, that is hard to analyse. In order to deal with that excessive amount of data balloon representations and polar diagrams are used. In this thesis a graphical user interface (GUI) will be developed in python™. This GUI will make it possible to view existing directivity data in balloon representation (in azimuth and zenith), in polar diagram (polar angle and sectional plane) and world map representation (Mollweide projection). A set of existing data will be analyzed and compared using this GUI.

Inhaltsverzeichnis

Einleitung

Diese Arbeit dokumentiert die Entwicklung eines Darstellungs- und Analysewerkzeugs für sphärische Richtwirkungsdaten, genannt 3D Directivity GUI. Richtwirkungsmessdaten werden häufig durch Impulsantwort-Messung in MIMO-Systemen (Multiple Input Multiple Output) in einem räumlichen Raster erhoben. Daraus ergeben sich große Datenmengen in mehreren Dimensionen, die ohne weitere Verarbeitung nicht untersucht und beurteilt werden können. Diese graphische Benutzeroberfläche (engl. Graphical User Interface - GUI) soll eine übersichtliche Darstellung großer Messdatenmengen, abhängig von deren Frequenz, Quelle beziehungsweise Empfänger und Einfallswinkel ermöglichen. Die zweidimensionale Darstellung dieser in aller Regel dreidimensionalen Einfallswinkel, erfordert eine Art der Projektion und soll in dieser Arbeit auf drei verschiedene Arten umgesetzt werden. Jede dieser Darstellungsweisen ermöglicht eine andere Ansicht und damit auch Einsicht in den untersuchten Datensatz. Am Institut für elektronische Musik und Akustik (IEM) der Universität für Musik und darstellende Kunst Graz (KUG) bestehen bereits einige Visualisierungsprogramme für große Messdatenmengen in Matlab. *balloon_holo.m* verfügbar unter [Zau18] ermöglicht die Darstellung eines Polardiagramms und einer Ballondarstellung am IEM verfügbarer Messdaten. *Dir_Pat_Viewer_2D* [Bra19a] erlaubt es Richtwirkungsdaten, die mit einer Doppelringmikrofonanordnung aufgenommen wurden zu analysieren und deren Richtwirkung sowie Frequenzgang anzuzeigen. *Pol_Pat_Spec* [Bra19c] ermöglicht die Wiedergabe von Aufnahmen durch Mehrkanalmikrofonanordnungen und durch ein vertikales und horizontales Polardiagramm die Echtzeitanalyse der Richtwirkung der aufgenommenen Quelle. *Dir_Pat_Viewer_3D* [Bra19b] dient der Erzeugung einer Ballondarstellung und eines Polarplots aus SOFA-Daten. Alle diese Visualisierungstools sind in der kommerziellen Skriptsprache Matlab implementiert. Da aber die meisten Messdaten unter *open-source*-Lizenzbedingungen veröffentlicht sind, ist ein *open-source*-Ansatz um diese darzustellen erstrebenswert. Ziel dieser Arbeit ist es folglich eine GUI in der *open source*-Programmiersprache python™ zu implementieren. Diese soll einerseits, durch gute Grafikanbindung, schnell bedienbar sein und andererseits im Vergleich zu *Dir_Pat_Viewer_3d* um eine Weltkartendarstellung erweitert werden. Die Integration des *Qt*-Frameworks durch *PyQT* und *PyQtGraph* verspricht eine einfach Grafikdarstellung und schnelle Erlernbarkeit der benötigten Werkzeuge. Der Autor hatte zu Beginn dieser Arbeit noch keine Berührpunkte mit python und erlernte die Programmiersprache im Verlauf des Projektes. Als Eingangsformat für die umfassenden Richtwirkungsmessdaten soll das auf räumliche Akustikmessdaten spezialisierte AES69 SOFA-Format [SOF20] verwendet werden. Diese Arbeit gliedert sich

in einen kurzen Theorieteil, eine ausführlichere Beschreibung der Implementierung und eine Evaluation der fertigen Anwendung. Im Rahmen der Evaluation können, anhand bereits vorhandener Datensätze von Aussenohrübertragungsfunktionen und der Ikosaederlautsprecher des IEM, die Darstellungsarten des Programmes getestet werden. Eine kurze Bedienungsanleitung erläutert die wichtigsten Bedienkonzepte. Der Programmcode steht dem IEM in einem Git-Repository[1] zur Verfügung und wird im Anhang dieser Arbeit abgedruckt. Die zur Ausführung notwendigen Bibliotheken sind in einer Anaconda[2]-Umgebung verfügbar.

[1] `https://git.iem.at/s1131471/3DGuiPython_wegler`
[2] [Ana16]

1 Theoretischer Hintergrund

Im Folgenden sollen die theoretischen Zusammenhänge der Visualisierung sphärischer Richtwirkungsdaten näher erläutert werden.

1.1 Messdaten

Sphärische Richtwirkungsdaten gibt es in großer Variation für die verschiedensten Anwendungsfälle. Das IEM stellt eine Vielzahl an Messdaten online zur Verfügung: [ZD11], [Zot19a] und [Zot19b] beinhalten Messdaten handelsüblicher Mikrofon und Lausprecheranordnungen. In [ZSZ18] sind die Richtwirkungsmessungen der Ikosaederlautsprecher IKO1, IKO2 und IKO3 zu finden und [Bra18] verweist auf die Datenkollektion des DirPat-Projektes. Sphärische Richtwirkungsdaten werden häufig als MIMO-Impulsantworten (Multiple Inputs Multiple Outputs) erhoben. Für einen HRTF-Datensatz (engl. Head Related Transfer Function, Aussenohrübertragungsfunktion) beispielsweise ergibt sich für jede gemessene Einfallsrichtung für beide Ohren der Versuchsperson eine Impulsantwort. Schon bei einer durchaus üblichen räumliche Auflösung von $10°$ folgen dabei für ein vollständiges Kugelraster: $2\,\mathrm{Ohren} \times 36\,\mathrm{Azimutwinkel} \times 18\,\mathrm{Elevationswinkel} = 1296\,\mathrm{Impulsantworten}$ und damit ein Datensatz in vier Dimensionen. Bereits diese erste Abschätzung zeigt die Notwendigkeit für strukturierte Speicherung und Datenverarbeitung. Obwohl Impulsantworten sehr geeignet sind zur vollständigen Beschreibung eines Übertragungssystesm, mangelt es ihnen an Interpretierbarkeit. Eine simple Darstellung der Impulsantwort abhängig von der Zeit, ist für die meisten Fälle nicht sehr anschaulich. Mittels einer Fouriertransformation ergibt sich, bei bekannter Abtastfrequenz f_S, aus der Impulsantwort, die frequenzabhängige Übertragungsfunktion in Betrag und Phase. Diese ist anschaulich und leicht zu interpretieren und soll in diesem Analysewerkzeug dargestellt werden. Die meisten Schallausbreitungsphänomene, wie Bündelung oder Streuung sind ohnehin frequenzabhängig.

1.2 Datenformat und Bezugskoordinatensystem

Das AES69 SOFA-Datenformat ist ein Container-Format für räumliche Impulsantworten und wurde 2015 von der Audio Engineering Society als Norm [AES15] definiert. SOFA basiert auf *netCDF* und enthält, neben den reinen Messdaten, auch verschiedene Metadaten. Diese Metadaten beinhalten Angaben über eine Vielzahl an Eigenschaften der enthaltenen Messdaten. Die genauen Spezifikationen des Formates sind in [SOF20]

Parameter	Wertebereich	Vorne, Augenhöhe	Links, Augenhöhe	Hinten, Augenhöhe
Azimut	$0° \dots 360°$	$0°$	$90°$	$180°$
Elevation	$-90° \dots 90°$	$0°$	$0°$	$0°$
Radius	> 0	–	–	–

Tabelle 1: Wertebereiche der Winkel und Größen in Kugelkoordinaten nach [SOF20]

nachzulesen, die wichtigsten seien hier explizit genannt:

Der angeführte Urheber und zugehörige Kontaktdaten ermöglichen die Rückverfolgung des gemessenen Datensatzes und erleichtern die Kontaktaufnahme bei Detailfragen. Das Erstellungsdatum und der Ort, sowie die Version der SOFA-Konvention und die Lizenz dienen der besseren Interpretation. Die Messdaten selbst sind als mehrdimensionale Matrix, zusammen mit der Abtastfrequenz und deren Einheit, sowie optionalen Verzögerungszeiten in einem Unterordner *Data* abgelegt. Als weiterer und gerade im Zusammenhang mit dieser Arbeit wichtigster Teil sind alle Positionen und Richtungen des Messaufbaus, sowie deren Einheiten in einem festgelegten Format hinterlegt. Die Richtungen und deren zugrunde liegendes Koordinatensystem bilden den Ausgangspunkt für jegliche Transformationen zur weiteren Darstellung. Die Positionen werden in die vier Kategorien Listener (Hörer), Receiver (Empfänger), Source (Quelle) und Emitter (Sender) gegliedert. Ein Listener enthält dabei mehrere Receiver, dadurch sind deren Orte relativ zu diesem bestimmt. Das gleiche Konzept gilt für Emitter, die an die zugehörige Source angeheftet sind. Für die jeweiligen Koordinatensysteme kommen die Attribute „cartesian“ (kartesische Koordinaten) oder „spherical“ (Kugelkoordinaten) in Betracht. Eine typische Messanordnung in kartesischen Koordinaten stellt Abbildung 1 dar. Die Koordinatentransformation zwischen diesen beiden Koordinatensystemen ist in Abbildung 2 zu sehen. Aus den festgelegten Koordinatentransformationen ergeben sich für Kugelkoordinaten die folgenden Wertebereiche und deren Bedeutung in Tabelle 1.2. Der Drehwinkel Φ ist dabei in der Draufsicht von der x-Achse aus gemessen und im positiven mathematischen Drehsinn als Azimutwinkel festgelegt. Der Höhenwinkel ist als Elevations- oder Höhenwinkel Θ von der xy-Ebene aus in z-Richtung definiert. Es ergibt sich dadurch ein Wertebereich von $\pm 90°$, vom Horizont aus gemessen. Im Gegensatz dazu hätte eine Zenitwinkeldefinition, die von einem der beiden Pole aus definiert ist, einen Wertebereich von $0° \dots 180°$. Obwohl die SOFA-Konvention eindeutig Elevationswinkel vorgibt, sind beide Definitionen in bestehenden Datensätzen anzutreffen. Diese mögliche Fehlerquelle soll in der Datenverarbeitung dieses Analysewerkzeuges abgefangen werden.

Die Definition der Messdatenquellen zu Sender und Empfänger ermöglicht zwei Konfigurationen: Einerseits mit dem Senderradius größer als dem Empfängerradius, wie bei-

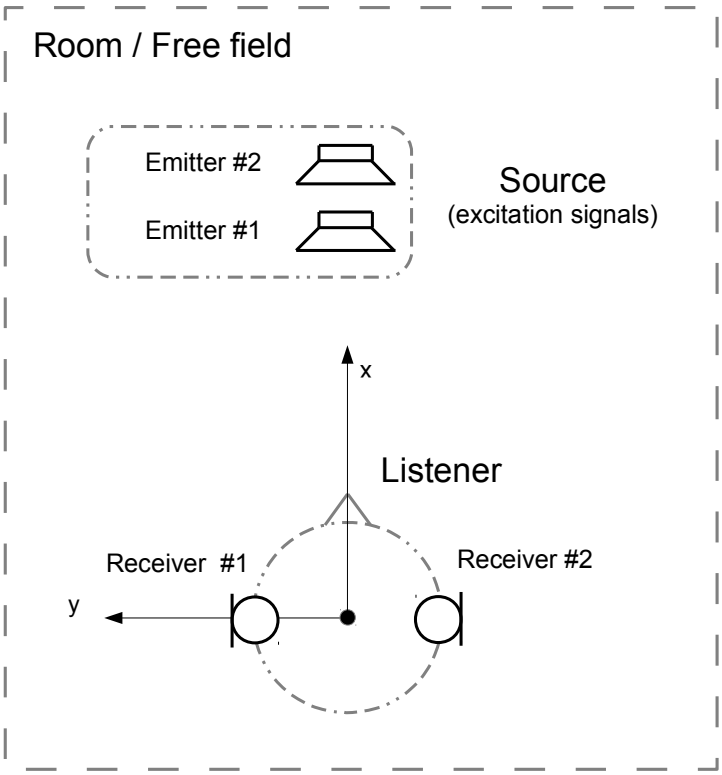


Abbildung 1: Typische Messanordnung (karthesische Koordinaten) nach SOFA-Konvention, aus [SOF20]

spielsweise bei einer HRTF-Messung. Andererseits mit dem Senderradius kleiner als dem Empfängerradius, beispielsweise bei der Richtwirkungsmessung eines Lautsprechers. Zur Analyse beider möglicher Konfigurationen soll das jeweils weiter außen liegende Messgitter zur Richtungsauswertung des jeweiligen Datensatzes verwendet werden. Das ist insofern sinnvoll, als in den meisten Anwendungsfällen das untersuchte Objekt im Zentrum des Messaufbaus steht und das Messraster, sei es aus Sendern oder Empfängern, außen herum aufgebaut wird.

1.3 Verschiedene Darstellungen

Im Rahmen dieser Arbeit sollen drei verschiedene Visualisierungsarten basierend auf drei verschiedenen Koordinatentransformationen untersucht werden:

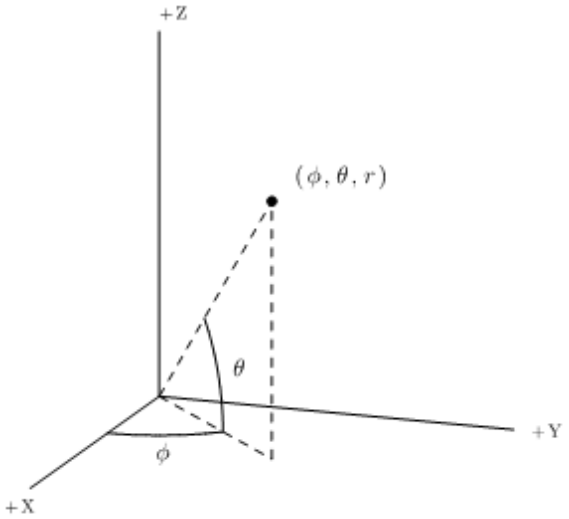


Abbildung 2: Koordinatensysteme nach SOFA-Konvention, aus [SOF20]

1.3.1 Ballondarstellung

Die Ballondarstellung basiert auf einer Vektorrepräsentation des gewählten Messrasters. Vektoren aus dem Koordinatenursprung in Richtung der Messpunkte werden skaliert mit dem Pegel des entsprechenden Messpunktes. Die Vektorspitzen definieren die Eckpunkte (engl. vertices) einer Hüllfäche. Dann werden jeweils drei benachbarte Eckpunkte so ausgewählt, dass sie ein Element (face) der Außenfläche des Körpers beschreiben. Eine Punktmenge, die diese Voraussetzung erfüllt, ist eine konvexe Hülle. Sie ist die kleinstmögliche Punktmenge, die alle Punkte einer Ausgangsmenge umschließt (vgl. [Con20]). Aus der Gesamtheit aller Elemente dieser umhüllenden Fläche ergibt sich dann ein Ballonkörper dessen Ausmaße dem Schallpegel in der zugehörigen Richtung entsprechen. Diese Darstellung bietet den Vorteil einer dreidimensionalen Übersicht über die zu analysierenden Messdaten. Die konvexe Hülle ermöglicht eine geschlossene Darstellung verschiedenster Messraster, da sie immer in der Lage ist eine umschließende Punktmenge zu finden. Eine Demonstration der berechneten Flächenelemente für eine Kugel mit zufälligen Farben befindet sich in Abbildung 3. Die Phaseninformation kann dann zusätzich in die Farbwerte der Ballonoberfläche kodiert werden.

1.3.2 Kartendarstellung als Mollweide-Projektion

In der Kartographie ist, seit Bekanntwerden der Kugelgestalt der Erde, das Problem der zweidimensionalen Abbildung dieser allgegenwärtig. Unterteilt man eine Kugeloberflä-

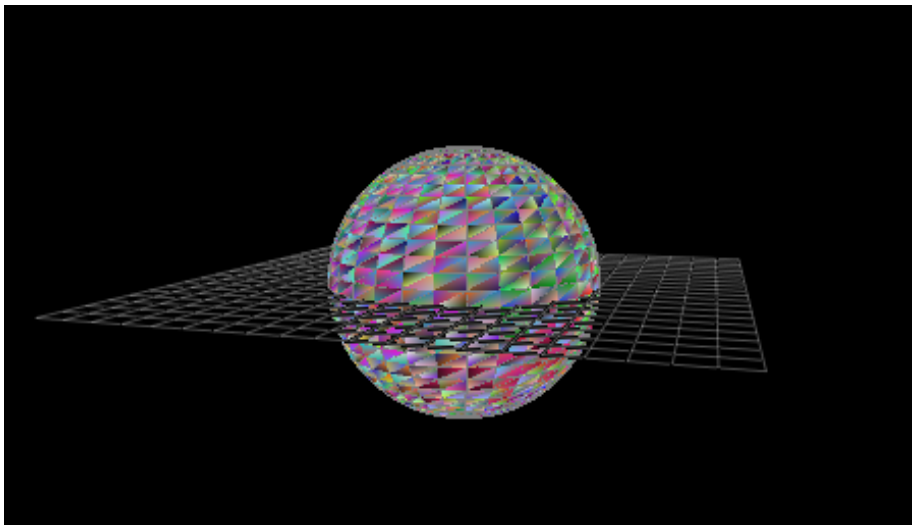


Abbildung 3: Ballondarstellung einer Kugel, mit Flächen in zufälligen Farben

che, zum Beispiel an bestimmten Winkeln, einerseits durch konzentrische und andererseits durch parallele Ebenen, entsteht ein umgebendes Netz aus Längen- und Breitenkreisen. Dieses Kugelnetz ist nicht zweidimensional abwickelbar, daher stellen Projektionen einen notwendigen Ansatz zur Visualisierung einer vollständigen Weltkarte dar. Eine Projektion ist dabei eine mathematische Transformation, die Punkte der Oberfläche eines dreidimensionalen Objektes einer zweidimensionalen Abbildung zuweist. Im Wesentlichen unterscheidet man die winkeltreuen von flächentreuen Projektionen durch die Vergleichbarkeit des jeweiligen Parameters innerhalb der Darstellung. Eine winkeltreue Projektion, wie beispielsweise die in der Seefahrt übliche Mercatorprojektion streckt die Darstellung um einen Faktor $|1/\cos\Theta|$, so dass eine rechteckiges Abbild des Kugelnetzes entsteht. So sind die eingezeichneten Kurse (und damit Winkel) relativ zum Messgitter (Längen-/ und Breitenkreise) überall identisch und damit vergleichbar. Flächenstücke an verschiedenen Positionen der Karte sind dementsprechend verzerrt. Zur Darstellung von Messpunkten, deren Größe und Relation zueinander wichtiger als deren exakte Lage und die Winkel zueinander sind, stellt aber die Fläche eine durchaus relevantere Größe dar. Die Winkelbeziehungen der Messrichtungen zueinander sind in anderen Darstellungsarten besser nachvollziehbar und werden hier vernachlässigt. Daher soll zur Weltkartenabbildung im Rahmen dieser Projektarbeit eine flächentreue Projektion, die Mollweideprojektion, verwendet werden. Diese ist nach Carl Brandan Mollweide benannt und projiziert die Kugeloberfläche auf eine elliptische Fläche. Eine Weltkarte in Mollweideprojektion ist in Abbildung 4 dargestellt.

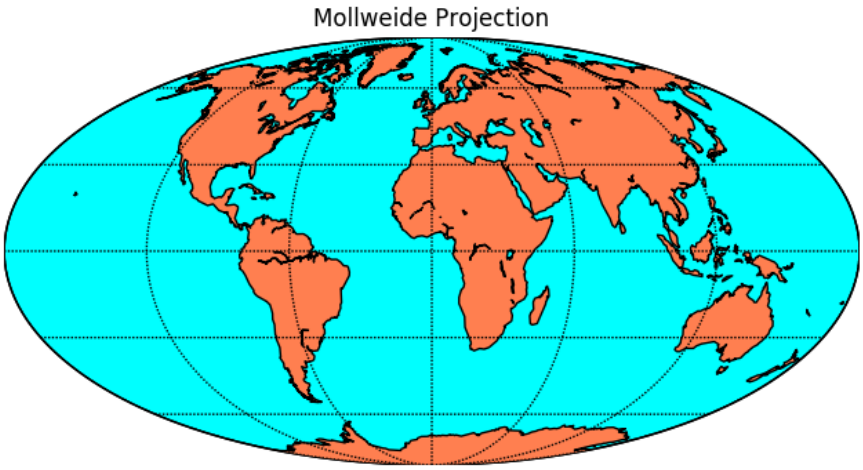


Abbildung 4: Mollweideprojektion, Darstellung aus [Whi11]

1.3.3 Polardarstellung durch Interpolation mittels Kugelflächenfunktionen

Die Polardarstellung bildet den gemessenen Pegel in einer Schnittebene als Auslenkung des Radius auf einem Polarkoordinatensystem ab. Existieren für die gewählte Schnittebene ausreichend viele Messpunkte, ist der Polarplot die einfachste Art der Darstellung. Es können jedoch nur im Messaufbau vorhandene Ebenen sinnvoll abgebildet werden. Um eine Darstellung auch für andere als die sich aus dem Aufbau ergebenden Schnittebenen zu ermöglichen, wird in diesem Visualisierungswerkzeug eine Interpolation der Messdaten anhand von Kugelflächenflunktionen, vergleichbar mit Ambisonics-Konzept, umgesetzt.

Dazu werden alle Messpunkte V in Kugelflächenfunktionen enkodiert (Y). Die entsprechende Kodierungsmatrix iY ist die Pseudoinverse der Kugeflächenrepräsentation Y. Für die Auswertungsstützstellen Q wird ebenfalls eine Repräsentation in Kugelflächenfunktionen Z zur Dekodierung gefunden. Aus der Multiplikation des Datenvektors d mit der Kodierungsmatrix iY und Dekodierungsmatrix Z ergeben sich schließlich die interpolierten Messdaten b in der ausgewerteten Ebene.

$$b = Z \cdot iY \cdot d \tag{1}$$

Der Absolutbetrag des interpolierten Datenvektors b ergibt den Radius für jeden dargestellten Messpunkt. Abbildung 5 zeigt ein Beispiel eines Polarplots.

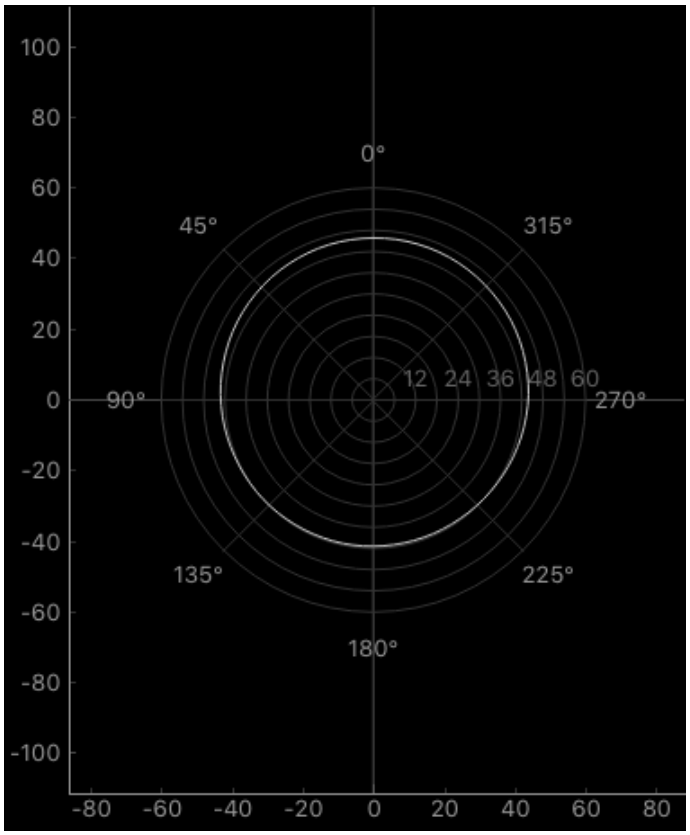


Abbildung 5: Polarplot, interpoliert in $4°$-Schritten.

2 Implementierung

Die Implementierung der Darstellungs- und Analysesoftware 3D Directivity GUI erfolgte in der Skriptprogrammiersprache python™, Version 3.7, verfügbar unter [VRDJ95]. Python wird unter open source Lizenzbedingungen entwickelt und bietet unter anderem die Möglichkeit grafische Benutzeroberflächen und eigenständige Anwendungen zu entwerfen. Die Distribution Anaconda [Ana16] ermöglicht durch die enthaltene Paketverwaltung einen anwenderfreundlichen Einstieg in die Programmierung mit python. Gerade die integrierte Entwicklungsumgebung Spyder erleichtert, durch ihren ähnlichen Aufbau, den Umstieg von der gleichsam etablierten Skriptsprache MATLAB auf python.

2.1 GUI-Programmierung in *PyQT5*

PyQt5, online unter [RCL19], ist eine python-Anbindung an das Qt-Framework (siehe [Qt 20]) und ermöglicht die Verwendung von in Qt vordefinierten Klassen und Modulen in py-

thon. Diese Klassen und Module stellen alle benötigten Oberflächen- und Bedienelemente dieser Anwendung zur Verfügung. Durch das Qt eigene Signal-/Slot-Konzept ist die schnelle Anbindung der Bedienelemente an den Programmkern gegeben. So können Benutzereingaben nicht nur Variablen verändern, sondern auch Funktionen des Programmkerns aufrufen.

Das optische Erscheinungsbild der Anwendung und deren Anbindung an den Programmkern kann durch das Zusatzprogramm QtDesigner entworfen werden. QtDesigner ist ein drag-and-drop-Werkzeug zur Erstellung von Anwendungsoberflächen in Qt. Dieses ist zwar nicht für die Funktion mit PyQt optimiert, deren Kompatibilität wird aber durch die PyQt5-Erweiterung *UI code generator, 5.7* ermöglicht. [3] Abbildung 6 zeigt beispielhaft den Entwurfsprozess für diese Anwendung. Durch die Definition von Layouts und Mindestgrößen der verwendeten Elemente im Anwendungsfenster, werden deren Anordnung und Größenverhältnisse festgelegt. Das ermöglicht die spätere Größenveränderung des angezeigten Fensters.

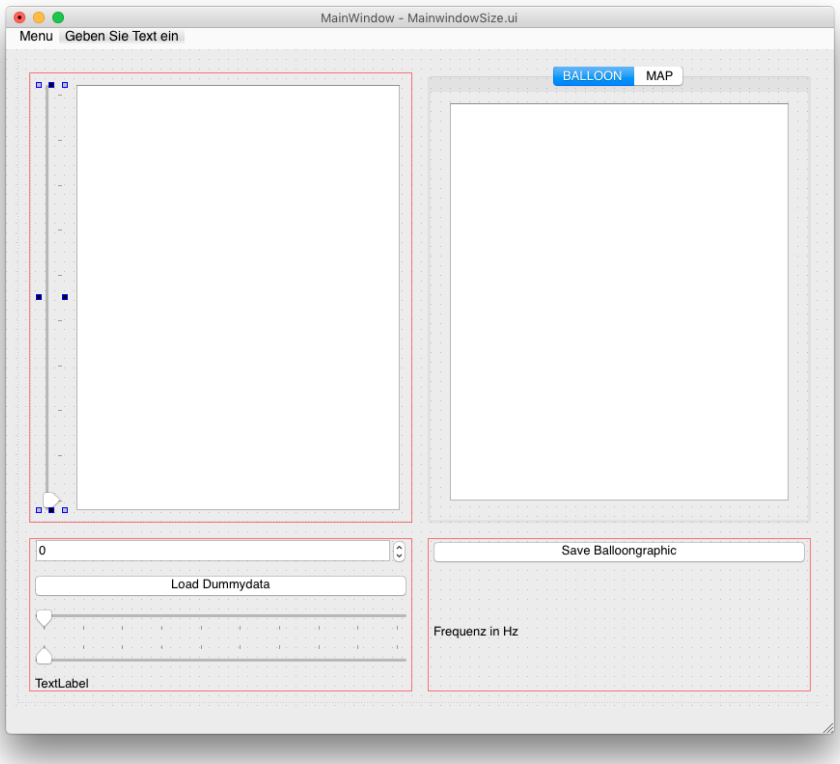


Abbildung 6: Entwurf der Oberfläche in QtDesigner

[3] https://www.riverbankcomputing.com/static/Docs/PyQt5/designer.html

Die mit PyQt5-*UI code generator* erzeugte Klasse *Ui_Fensterl*, stellt eine Erweiterung der Klasse *QtGui.QMainWindow* dar und erhält spezifisch zum Betriebssystem das Aussehen eines üblichen Programmfensters. Die in der Hauptschleife des Programms aufgerufene Instanz der Klasse *Ui_Fensterl* beinhaltet dann die Funktionen und Objekte der Benutzeroberfläche. In der Klassendefinition ist der Programmkern mit allen weiteren Funktionen und Prozeduren des Analysewerkzeuges angefügt.

2.2 Funktionen und Programmteile

Im Folgenden werden die Funktionen und die Aufteilung des Programmkernes näher erläutert.

2.2.1 Import und Verarbeitung der Daten in file_open

Die Funktion textbffile_open wird durch die Schaltfläche **Load data!** oder über das **Menu** ausgelöst. Die Funktion ist so aufgebaut, dass eine modulare Erweiterung, um beispielsweise zusätzliche Dateiformate zu unterstützen, möglich ist. Der Aufruf der **file_open**-Funktion öffnet ein Dialogfenster zur Dateiauswahl und bereitet deren Inhalt zur Weiterverarbeitung vor. Der Dateiname des geladenen Datensatzes wird im Fenster über der Schaltfläche angezeigt. Der Titel der Messung ist in der Statusleiste des Fensters (unten) zu sehen.

Datenimport Die unterstützten Dateiformate von 3D Directivity GUI sind die am IEM erhobenen Messdaten im .mat-Format und das für dreidimensionale Messdaten genormte AES69 SOFA-Format. Das SOFA-Datenformat basiert auf der *netCDF*-Konvention und kann deshalb durch die *netCDF4* python-Erweiterung (siehe [net12]) interpretiert werden. Konventionskonforme SOFA-Daten in sphärischen und kartesischen Koordinaten können eingelesen werden und das im Mittel weiter außen liegende Koordinatensystem, Quelle oder Empfänger, wird als Messgitter zur Richtungsauswertung verwendet.

Die .mat-Dateien müssen in Azimut- und Zenitwinkeln vorliegen und in einer *'hall'* genannten Matrix die Messdaten als mehrdimensionale Impulsantworten enthalten. Die Kompatibilität mit diesem Datentyp dient ausschließlich der Auswertung bereits vorhandener Messdaten dieser Art.

Datenverarbeitung Die eingelesenen Impulsantworten werden einer Fouriertransformation unterzogen und anschließend normalisiert und auf einen Wertebereich von $0\,\mathrm{dB}$ bis $60\,\mathrm{dB}$ begrenzt. Ein Dynamikbereich von $60\,\mathrm{dB}$ entspricht drei Dekaden und sollte für die meisten Anwendungsbereiche ausreichend sein. Als Bezugswert für die Normalisierung gilt der Maximalwert aller eingelesenen Pegeldaten.

2.2.2 Datenexport

Die erzeugten grafischen Darstellungen der eingelesenen Messdaten sollen als Grafikdateien exportiert werden. Mittels der Schaltfläche **Save graph** öffnet sich ein Dialogfenster zum Speichern der gerade aktiven Grafik in der Tab-Umgebung (Ballon oder Kartendarstellung). In der Eingabeaufforderung können Speicherort und Name der Grafikdatei gewählt werden. Die gespeicherte Grafik liegt im .png-Format vor und enthält den angezeigten Bildausschnitt.

Das Polardiagramm verfügt über eine interne Exportfunktion, die mittels Rechtsklick auf das Diagramm aktiviert werden kann. Gängige Grafikformate und die Bildgröße sind frei wählbar.

2.2.3 Ballondarstellung mittels balloonrender

Die Erzeugung der Ballondarstellung in **balloonrender** setzt sich aus den beiden Teilfunktionen **drawBalloonempty** und **ReDrawBalloon** zusammen. Diese werden durch den Prozessablauf ausgelöst. **drawBalloonempty** initialisiert das Grafikobjekt 'Balloon' und das zugehörige Koordinatensystem und fügt beide anschließend der Zeichenebene hinzu. Das Ballon-Grafikobjekt ist abgeleitet aus der Klasse GLMeshItem aus *pyqtgraph.opengl* (aus [Cam11]). Diese Klasse erwartet eine Matrix von Stützstellen (vertices) und eine Indexliste dieser, zur Definition der Oberflächen (faces). *Scipy.spatial* (aus [Sci08]) definiert den ConvexHull-Algorithmus zur Oberflächendefinition. **colorize** erzeugt die Farbe des Ballons durch eine interne Zuweisungstabelle. Die Farbe repräsentiert die Phaseninformation an der entsprechenden Stützstelle.

Das eingefügte Koordinatensystem in der XY-Ebene, verfügt über ein $6\,\mathrm{dB}$-Raster und ist eine Instanz der Klasse GlGridItem.

redrawBalloon aktualisiert die Daten der Ballongrafik und ist durch die nicht mehr notwendige Initialisierung der Grafik und des Koordinatensystems schneller als **drawBalloonempty**. Die Ballongrafik ist in eine Tab-Umgebung eingegliedert, so dass zwischen

ihr und der Weltkartendarstellung gewechselt werden kann. Das jeweils nicht angezeigte Element ist aus der Berechnung ausgenommen, auch dadurch sinkt die benötigte Rechenzeit.

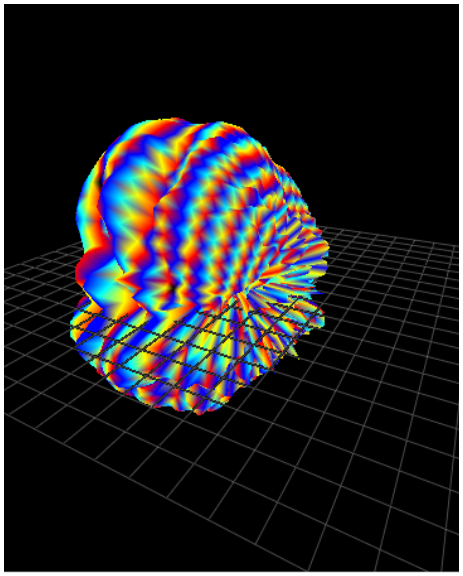


Abbildung 7: Balloondarstellung einer HRTF-Messung als Beispiel.

2.2.4 Kartenprojektion durch drawMap

Die Weltkartendarstellung bedient sich des gleichen Konzepts zur Reduktion der Rechenzeit wie die Ballondarstellung. In **drawMap** geschieht die Initialisierung der Zeichenebene und die Definition der Mollweide-Projektion mittels *pyproj* (entwickelt von [Whi06]). Die Grafikdarstellung wird durch *mpl_toolkits*s Basemap (aus der Standarbibliothek Matplotlib) erstellt und kann durch **redrawmap** schnell aktualisiert werden. Der darzustellende Pegel wird durch eine interne Zuweisungstabelle farblich kodiert. Ein Farbbalken stellt den Zusammenhang zwischen Farbe und Pegel dar. Abbildung 8 zeigt die Kartenprojektion eines HRTF-Beispieldatensatzes.

2.2.5 Polarplot und sphärische Interpolation in drawPolar

Auch für den Polarplot werden zwei verschiedene Funktionen zur Initialisierung und Aktualisierung verwendet. Die Initialisierung in **drawPolar** berechnet die Kodierungsmatrizen der Mess- und Auswertungspunkte (wie in Abschnitt 1.3.3), mittels *sph_harm* aus *Scipy.spatial* [Sci08]. Die ausgewertete Ebene ist parallel zur XY-Ebene und deren Ver-

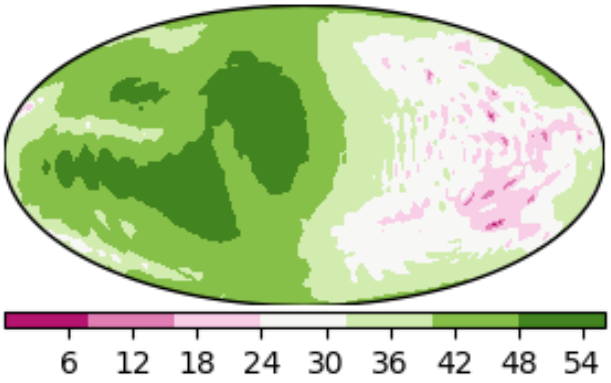


Abbildung 8: Kartendarstellung einer HRTF-Messung als Beispiel.

schiebung im Messgitter (Initialisierung: $\Theta = 0°$) ist durch einen Schieberegler an der linken Seite des Polarplots einstellbar. Eine Veränderung der Auswertungspunkte macht eine Neuberechnung der Dekodierungsmatrix notwendig. Die Berechnung der Pseudoinversen iY und die Auswertung der Kugelflächenfunktionen stellen sich dabei als vergleichsweise rechenintensiv heraus. Der Rechenaufwand steigt insbesondere mit der maximalen Ordnung, der zu bestimmenden Kugelflächenfunktion. Daher wurde dieser Parameter als Schieberegler auf der Bedienoberfläche zugänglich gemacht. Das ermöglicht eine Optimierung des Parameters zur Laufzeit.

Der abgebildete Dynamikbereich von $60\,\mathrm{dB}$ wird durch ein kreisförmiges Raster mit $6\,\mathrm{dB}$ Abständen unterteilt. Abbildung 9 zeigt die Polardarstellung einer Schnittebene durch einen HRTF-Datensatz als Beispiel.

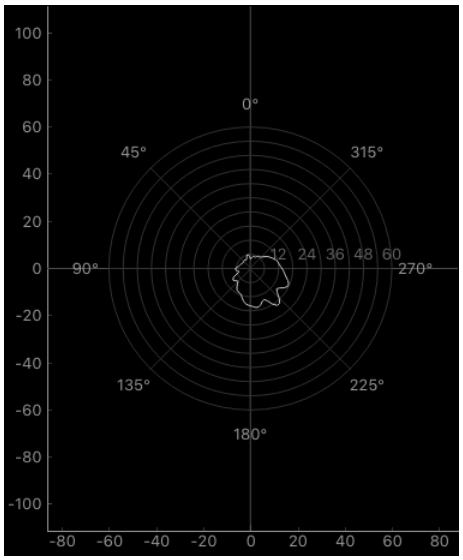


Abbildung 9: Polarplot eines HRTF-Datensatzes als Beispiel.

2.2.6 Bedienelemente

Neben den bereits beschriebenen Schaltflächen zum Laden eines Datensatzes und Speichern der Grafikdateien gibt es die folgenden weiteren Bedienelemente:

Menu Im Menü oben links sind die Funktionen zum Laden und Speichern von Daten ebenfalls erreichbar. Die eingetragenen Tastaturkürzel können zur Steuerung dieser Funktionen verwendet werden.

Frequenz-Schieberegler Direkt über dem Dateinamen befindet sich der horizontale Schieberegler zur Einstellung der zu analysierenden Frequenz. Die ausgewählte Frequenz ist als Textfeld rechts davon eingeblendet.

Interpolationsordnung-Schieberegler Der darüber liegende horizontale Schieberegler dient der Festlegung der Interpolationsordnung für das Polardiagramm. Standardmäßig ist dieser auf die Ordnung 18 festgelegt, um eine flüssige Bedienbarkeit von 3D Directivity GUI zu gewährleisten. Niedrigere Ordnungen führen zu ungenauer Interpolation, höhere Ordnungen neigen je nach Datensatz zur Instabilität der Anzeige.

Quellenauswahl-Nummernbox Die Nummernbox dient der Quellen- oder Empfängerauswahl im Messdatensatz. Im einfachen Beispiel eines HRTF-Datensatzes sind Mess-

daten für beide Ohren, also zwei verschiedene Empfänger und viele verschiedene Lautsprecher (Quellen) vorhanden. Die Lautsprecher und deren Richtungen werden als Messgitter ausgewertet. Die Auswahl des Empfängers geschieht durch die Nummernbox.

Schnittebene-Schieberegler Der vertikale Schieberegler links vom Polardiagramm verändert die ausgewertete Schnittebene der Polardarstellung. Im Allgemeinen ist dieser in der xy-Ebene ($\Theta = 0°$) initialisiert und hat einen maximalen Einstellbereich von $-90°$ bis $90°$ in $10°\circ$-Schritten.

2.3 Fehlerbetrachtung

Die erzielte Implementierung der 3D Directivity GUI macht eine ausführliche Fehlerbetrachtung notwendig. Die nachfolgenden Ausführungen erläutern bekannten Fehler dieser Anwendung und deren Zustandekommen.

Geschwindigkeit Obwohl auf der Geschwindigkeit und hohen Reaktionsschnelligkeit der Bedienelemente ein Hauptaugenmerk der Entwicklung lag, wurden diese nicht in jeder Hinsicht erreicht. Nicht nur das Öffnen eines neuen Datensatzes, einschließlich der notwendigen Datenverarbeitung dauert unter Entwicklungsbedingungen mehrere Sekunden, auch auch das Verschieben der Schnittebene des Polarplots verzögert den Programmablauf merklich. Nach näherer Untersuchung der einzelnen Funktionsgruppen konnte die *drawPolar*-Funktion als Hauptursache der Verzögerung gefunden werden. Ein Reduzieren der Interpolationsordnung innerhalb dieser Funktion verbessert die Reaktionsgeschwindigkeit der Anwendung. Ruckeln und Berechnungspausen treten dennoch gelegentlich auf. Die Interaktion des Ballonplots mit der Maus (Drehen, Zoom) funktioniert durch die direkte Grafikanbindung mittels *opengl* in *pyqtgraph* jedoch flüssig und schnell.

pyqtgraph Die Bibliothek *pyqtgraph* birgt, abseits ihrer Geschwindigkeit, einige Einschränkungen, die erst im Verlauf der Arbeit mit dieser offensichtlich werden. *Pyqtgraph* bietet keine integrierte Möglichkeit ein Polardiagramm zu erstellen, daher müssen die benötigten Anzeigeelemente (Raster, Beschriftungen) einzeln erzeugt werden. Ebenso existiert keine Funktion in *pyqtgraph* die es erlaubt, die Farbe einer gezeichneten Kurve punktweise zu bestimmen. Daran scheitert der Versuch die Phaseninformation in die Farbe des Plots zu kodieren. Eine weitere Einschränkung bedeutet das Fehlen einer Klasse für Farbbalken in der für die Ballondarstellung verwendeten Anzeigeumgebung (Gl-

ViewWidget). Die Farbauswahl der in *pyqtgraph* funktionierenden Zuweisungstabellen ist beschränkt. Die vergleichsweise kleine Nutzergruppe der Bibliothek erschwert die Onlinesuche von Problemlösungen. Die Dokumentation der Bibliothek ist häufig nicht ausreichend und macht zeitlich aufwendiges Ausprobieren notwendig. Ein Ausweichen auf eine andere Bibliothek zur Grafikdarstellung wäre möglich und *vispy* bietet sich als Erweiterung von *pyqtgraph* an. Die Umstellung selbst erfordert aber erheblichen Entwicklungsaufwand, der innerhalb dieses Projektes nicht geleistet werden kann. Zudem sind Einbußen in der Darstellungsgeschwindigkeit nicht auszuschließen.

Entwicklungsumgebung Spyder Kompatibilitätsprobleme zwischen *PyQt* und der Entwicklungsumgebung *Spyder* gestalteten den Programmierprozess in der Anfangszeit dieses Projektes aufwendig. Der in *Spyder* integrierte Debug-Modus unterstützte das für Benutzerinteraktion notwendige Schleifenkonstrukt nicht. So führte ein Aufruf der Debug-Routine zu einem Absturz der Anwendung. Dadurch war unter anderem nicht erkennbar in welchem Teilprogramm sich Fehler befinden und alle Teile mussten so getrennt voneinander entwickelt werden. Eine weitere Hürde in dieser Konstellation aus Bibliotheken und Entwicklungsumgebung stellten die sogenannten Namespaces dar. Ein Namespace stellt allgemein eine Verbindung zwischen Namen und Objekten her (vgl. [VRDJ95]), definiert aber damit auch abgeschlossene Räume innerhalb derer Objekte und Funktionen zugreifbar sein sollen. Dieses Konzept ermöglicht die mehrfache Instanziierung eines Objektes oder die parallele Ausführung von gleichen Funktion, da das jeweils andere Objekt/Funktion einen eigenen Namespace erhält. So können gleichnamige Variablen existieren, die voneinander völlig unabhängig sind. Im Zusammenhang mit der in *PyQt* verwendeten Objektstruktur einer Benutzeroberfläche ermöglichte *Spyder* aber ausschließlich den Zugriff auf den Namespace des Hauptfensters (QtWidgets.QApplication). Da aber alle Funktionen innerhalb des Objekts QtGui.QMainWindow definiert sind und zur Laufzeit kein Anhalten des Programmablaufes ohne Absturz des Hauptfensters möglich war, blieb nur die Verwendung von globalen Variablen, die im Namespace des Hauptfensters zugreifbar sind, als Lösung zur Kontrolle dieser. Dieses Problem konnte erst gegen Ende des Projektzeitraumes und mit *Spyder*, Version 4 behoben werden. Das machte den Variablenexplorer nutzbar und beschleunigte den Entwicklungsprozess ab diesem Zeitpunkt erheblich.

3 Zusammenfassung und Evaluation

Die erreichte Implementierung der 3D Directivity GUI erfüllt die grundgelegten Anforderungen.

Vorliegende SOFA-Messdaten können eingelesen und mittels drei verschiedener Darstellungen analysiert werden. Die Bedienelemente ermöglichen die Einstellung der zu analysierenden Frequenz und Quelle, sowie der Schnittebene der Polardarstellung. Eine Einstellung der Interpolationsordnung des Polarplots ist zu Testzwecken möglich. 3D Directivity GUI ermöglicht den Export aller dargestellten Diagramme. Im Folgenden sollen die Datensätze einer KEMAR-Kunstkopfanordnung und zweier IKO-Ikosaederlautsprecher beispielhaft verglichen werden. Die erzeugten Diagramme befinden sich nach Datensatz und Frequenz sortiert im Anhang A. Die im Anhang A.1 und A.2 gezeigten Messdaten entstammen der Richtwirkungsmessung der Ikosaederlautsprecher IKO1 und IKO3 aus [ZSZ18]. Der Datensatz der Außenohrübertragungsfunktionen (HRIR) der KEMAR Kunstkopfanordnung entstammt [Wie11]. Ein Vergleich der verschiedenen Frequenzen des Datensatzes IKO1, zeigt das grundsätzliche Frequenzverhalten des Ikosaederlautsprechers. Zu tiefen Frequenzen $f = 43\,\mathrm{Hz}$, siehe Abbildung 10, strahlt der Lautsprecher nahezu kugelförmig ab. Mit steigender Frequenz $f = 990\,\mathrm{Hz}$, vgl. Abbildung 11, zeigen sich erste Nebenkeulen. Für hohe Frequenzen $f = 9991\,\mathrm{Hz}$, siehe Abbildung 12, ist dann ein bereits erheblich reduzierter und Pegel und völlig gerichtete Abstrahlung erkennbar.

Der Vergleich mit IKO3 $f = 43\,\mathrm{Hz}$, siehe Abbildung 13, zeigt im Polardiagramm den geringeren Pegel und damit die geringere Tiefbassausbeute. Die Einschnürung zur Seite hin, deutet auf einen akustischen Kurzschluss hin. Durch die Normierung auf den Maximalpegel jedes Datensatzes ist keine absolute Vergleichbarkeit der Pegel gegeben. Bei der Mittenfrequenz $f = 990\,\mathrm{Hz}$, siehe Abbildung 14, zeigt IKO 3 etwas stärkere Bündelung als IKO1.

Der Vergleich mit dem HRTF-Datensatz offenbart weitaus geringere Bündelung (vgl. Abbilgung 16). Erst bei Frequenzen ab $1500\,\mathrm{Hz}$ zeigen sich Anzeichen einer Einschnürung. Bei $9750\,\mathrm{Hz}$ (vgl. Abbildung 17) entsteht ausgeprägte Bündelung. Durch den im Vergleich zu einem Ikosaederlautsprecher kleinen Kopf ist dieses Frequenzverhalten durchaus erwartbar.

Die Grafikdarstellung funktioniert zuverlässig und gibt einen schnellen Einblick in die zu analysierenden Richtwirkungsdaten. Der Bilderexport funktioniert schnell und zuverlässig. Der zeitaufwendige Programmierprozess, die in Abschnitt 2.3 aufgeführten Unwägbarkeiten und die zu Beginn fehlende Programmiererfahrung des Autors hinterlassen

jedoch einige Verbesserungsmöglichkeiten, die im Rahmen dieses Projektes nicht umgesetzt werden konnten. Die schwarze Hintergrundfarbe der Polar- und Ballondarstellung ist für gedruckte Veröffentlichungen nicht ideal. Ein Farbbalken zur Bestimmung der Phasenlage in der Ballondarstellung wäre nützlich, ist in *pyqtgraph* aber nicht vorgesehen. Ein kreisförmiges Raster wäre ebenso zweckmäßig, ist aber nicht implementiert. Eine farbliche Kodierung der Phase im Polardiagramm ist ebenfalls nicht umsetzbar. Beschriftete und logarithmische Schieberegler, zumindest für die Frequenzauswahl, wären nützlich, sind aber in *PyQt* nicht vorgesehen und nur aufwendig umsetzbar.

Die angefügte Bedienungsanleitung soll eine Verwendung von 3D Directivity GUI ohne ein vollständiges Studium dieser Arbeit ermöglichen.

4 Bedienungsanleitung

Diese Kurzanleitung erklärt die wichtigsten Bedienelemente des Analyse- und Darstellungswerkzeugs 3D Directivity GUI (vgl. 4.4)[4].

4.1 Importieren von Messdaten

Der Messdatensatz im .sofa-Format kann über die Schaltfläche **Load data** eingelesen werden. Nach erfolgtem Import der Daten ist der Dateiname im Fenster zu lesen und ein in der .sofa-Datei eingetragener Titel der Messung wird in der Statusleiste des Fensters angezeigt.

4.2 Interpretation der Anzeige

Alle Anzeigen sind begrenzt auf einen Dynamikumfang von $60\,\mathrm{dB}$ und auf den lautesten gemessenen Pegel normalisiert.

Polardiagramm Das Polardiagramm stellt eine Schnittebene durch das Koordinatensystem der Messung parallel zur xy-Ebene dar.

Ballondarstellung Die Ballondarstellung zeigt die Pegel als Ausdehnung in die zugehörige Raumrichtung an. Die Phaseninformation ist in die Farbe des Ballons kodiert. Das Raster entspricht $6\,\mathrm{dB/div}$.

Kartendarstellung Die Mollweideprojektion der Messdaten zeigt den Pegel in der entsprechenden Raumrichtung farblich kodiert an. Der Farbbalken gibt die Zuordnung zum Pegel (in dB) an. Die $0°$-Richtung liegt im Koordinatenzentrum in der Bildmitte.

4.3 Bedienelemente

Menu Im Menü sind die Funktionen zum Laden und Speichern von Daten erreichbar. Die eingetragenen Tastaturkürzel können zur Steuerung dieser Funktionen verwendet werden.

[4] `https://git.iem.at/s1131471/3DGuiPython_wegler`

Load data - Schaltfläche siehe 4.1

Save graph - Schaltfläche siehe 4.4

Frequenz-Schieberegler Direkt über dem Dateinamen befindet sich der horizontale Schieberegler zur Einstellung der zu analysierenden Frequenz. Die ausgewählte Frequenz ist als Textfeld rechts davon eingeblendet.

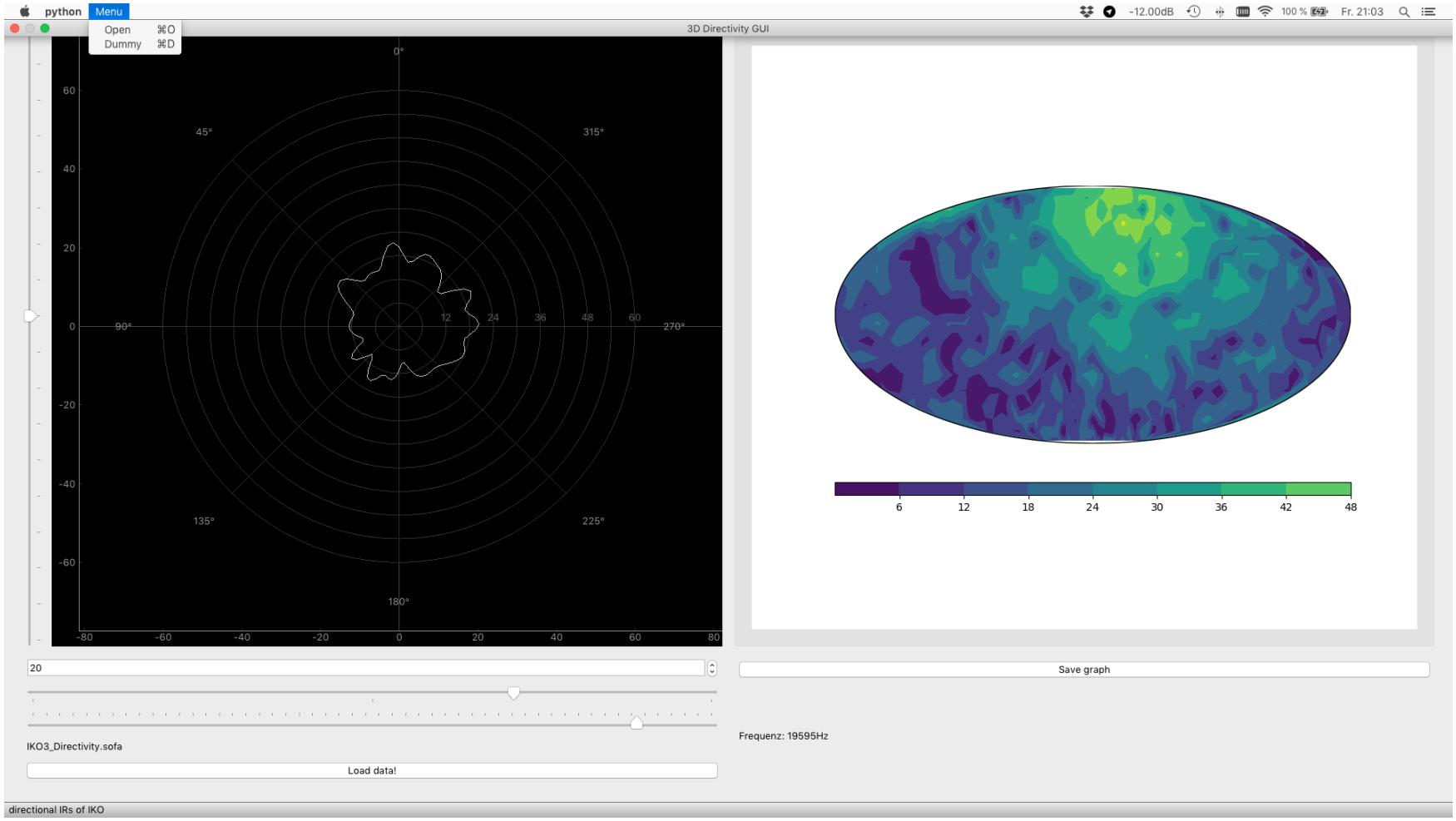
Interpolationsordnung-Schieberegler Der darüber liegende horizontale Schieberegler dient der Festlegung der Interpolationsordnung für das Polardiagramm. Standardmäßig ist dieser auf die Ordnung 18 festgelegt, um eine flüssige Bedienbarkeit von 3D Directivity GUI zu gewährleisten. Niedrigere Ordnungen führen zu ungenauer Interpolation, höhere Ordnungen neigen je nach Datensatz zur Instabilität der Anzeige.

Quellenauswahl-Nummernbox Die Nummernbox dient der Quellen- oder Empfängerauswahl im Messdatensatz. Im einfachen Beispiel eines HRTF-Datensatzes sind Messdaten für beide Ohren, also zwei verschiedene Empfänger und viele verschiedene Lautsprecher (Quellen) vorhanden. Der weiter außen liegende Koordinatensatz wird als Messgitter verwendet Die Auswahl des jeweils anderen Parameters (linkes oder rechtes Ohr bei HRTF) geschieht durch die Nummernbox.

Schnittebene-Schieberegler Der vertikale Schieberegler links vom Polardiagramm verändert die ausgewertete Schnittebene der Polardarstellung. Im Allgemeinen ist dieser in der xy-Ebene ($\Theta = 0°$) initialisiert und hat einen maximalen Einstellbereich von $-90°$ bis $90°$ in $10°$-Schritten im Koordinatensystem der Messdaten.

4.4 Export der angezeigten Grafik

Die Schaltfläche **Save graph** exportiert die gerade aktive Grafik (Ballon- oder Kartendarstellung) in eine .png-Datei. Ein Dialogfenster ermöglicht die Wahl des Dateinamen und Speicherortes. Exportiert wird die Grafik so wie sie angezeigt wird. Durch Rechtsklick auf das Polardiagramm ist ein Export desselben möglich. Das Dateiformat, sowie die Bildgröße sind wählbar.


python
Menu
Open ⌘O
Dummy ⌘D
-12.00dB
100 %
Fr. 21:03
3D Directivity GUI
0°
45°
315°
90°
270°
135°
225°
180°
60
40
20
0
-20
-40
-60
-80
12
24
36
48
60
6
12
18
24
30
36
42
48
20
Save graph
IKO3_Directivity.sofa
Load data!
Frequenz: 19595Hz
directional IRs of IKO

Literatur

[AES15] (2015) AES STANDARD AES69-2015: AES standard for file exchange - Spatial acoustic data file format. [Online]. Available: http://www.aes.org/publications/standards/search.cfm?docID=99

[Ana16] I. Anaconda. (2016) Anaconda navigator, 1.9.2. [Online]. Available: https://www.anaconda.com/

[Bra18] M. Brandner. (2018) Dirpat. DirPat Repository Elements: IEM Repository of Directivity Measurements of Sources and Receivers. [Online]. Available: https://phaidra.kug.ac.at/view/o:67857

[Bra19a] ——. (2019) Dirpat_viewer_2d_comparison. [Online]. Available: https://git.iem.at/brandner/DirPat_Viewer_2D_Comparison

[Bra19b] ——. (2019) Dirpat_viewer_3d. [Online]. Available: https://git.iem.at/brandner/DirPat_Viewer_3D

[Bra19c] ——. (2019) Polarpattern_and_spectrumanalyzer. [Online]. Available: https://git.iem.at/brandner/PolarPattern_and_SpectrumAnalyzer

[Cam11] L. Campagnola. (2011) pyqtgraph. [Online]. Available: http://www.pyqtgraph.org/documentation/index.html

[Con20] (2020) Convex hull. [Online]. Available: http://www.imn.htwk-leipzig.de/~medocpro/buecher/sedge1/k25t1.html

[net12] netCDF4. (2012) netcdf4 module, version 1.5.3. [Online]. Available: https://github.com/Unidata/netcdf4-python

[Qt 20] Qt Group (Nasdaq Helsinki: QTCOM). (2020) Pyqt5. [Online]. Available: https://www.qt.io

[RCL19] T. Q. C. Riverbank Computing Limited. (2019) Pyqt5. [Online]. Available: https://www.riverbankcomputing.com/static/Docs/PyQt5/

[Sci08] Scipy.spatial. (2008) Spatial algorithms and data structures (scipy.spatial). [Online]. Available: https://docs.scipy.org/doc/scipy/reference/spatial.html

[SOF20] (2020) SOFA (Spatially Oriented Format for Acoustics). [Online]. Available: https://www.sofaconventions.org/mediawiki/index.php?title=SOFA_(Spatially_Oriented_Format_for_Acoustics)&oldid=2151

[VRDJ95] G. Van Rossum and F. L. Drake Jr, *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995. [Online]. Available: https://www.python.org/

[Whi06] J. Whitaker. (2006) Python interface to proj (cartographic projections and coordinate transformations library). [Online]. Available: https://pypi.org/project/pyproj/

[Whi11] ——. (2011) Matplotlib basemap, mollweide projection. [Online]. Available: https://matplotlib.org/basemap/users/moll.html

[Wie11] H. Wierstorf. (2011) Hrirs. Anechoic HRIRs from the KEMAR manikin with different distances. [Online]. Available: https://gitlab.tubit.tu-berlin.de/twoears/data/tree/master/impulse_responses/qu_kemar_anechoic

[Zau18] M. Zaunschirm. (2018) balloon_holo. Balloon_holo is a further IEM driven attempt for convenient open data visualization of acoustic transceivers, such as icosahedral (IKO) and cubical shaped loudspeaker arrays capable of 3rd order and 1st order Ambisonic beamforming, respectively. [Online]. Available: https://git.iem.at/p2774/balloon_holo.git

[ZD11] F. Zotter and T. Deppisch. (2011) Zylia zm1. Directivity Measurement and Various Beamforming Filters of the Zylia ZM-1. [Online]. Available: https://phaidra.kug.ac.at/o:91938

[Zot19a] F. Zotter. (2019) Em32. Directivity Measurement and Various Beamforming Filters of the Eigenmike EM32. [Online]. Available: https://phaidra.kug.ac.at/o:69298

[Zot19b] ——. (2019) Sammlung: 170, 393, oko, dodekaeder... Directivity and Electro-Acoustic Measurements of the 170, 393, OKO, Dode, mixed-order Compact Spherical Loudspeaker Arrays. [Online]. Available: https://phaidra.kug.ac.at/view/o:91326

[ZSZ18] M. Zaunschirm, F. Schultz, and F. Zotter. (2018) Iko1...3. Directivity and Electro-Acoustic Measurements of the IKO. [Online]. Available: https://phaidra.kug.ac.at/o:67609

A Diagramme

A.1 IKO1

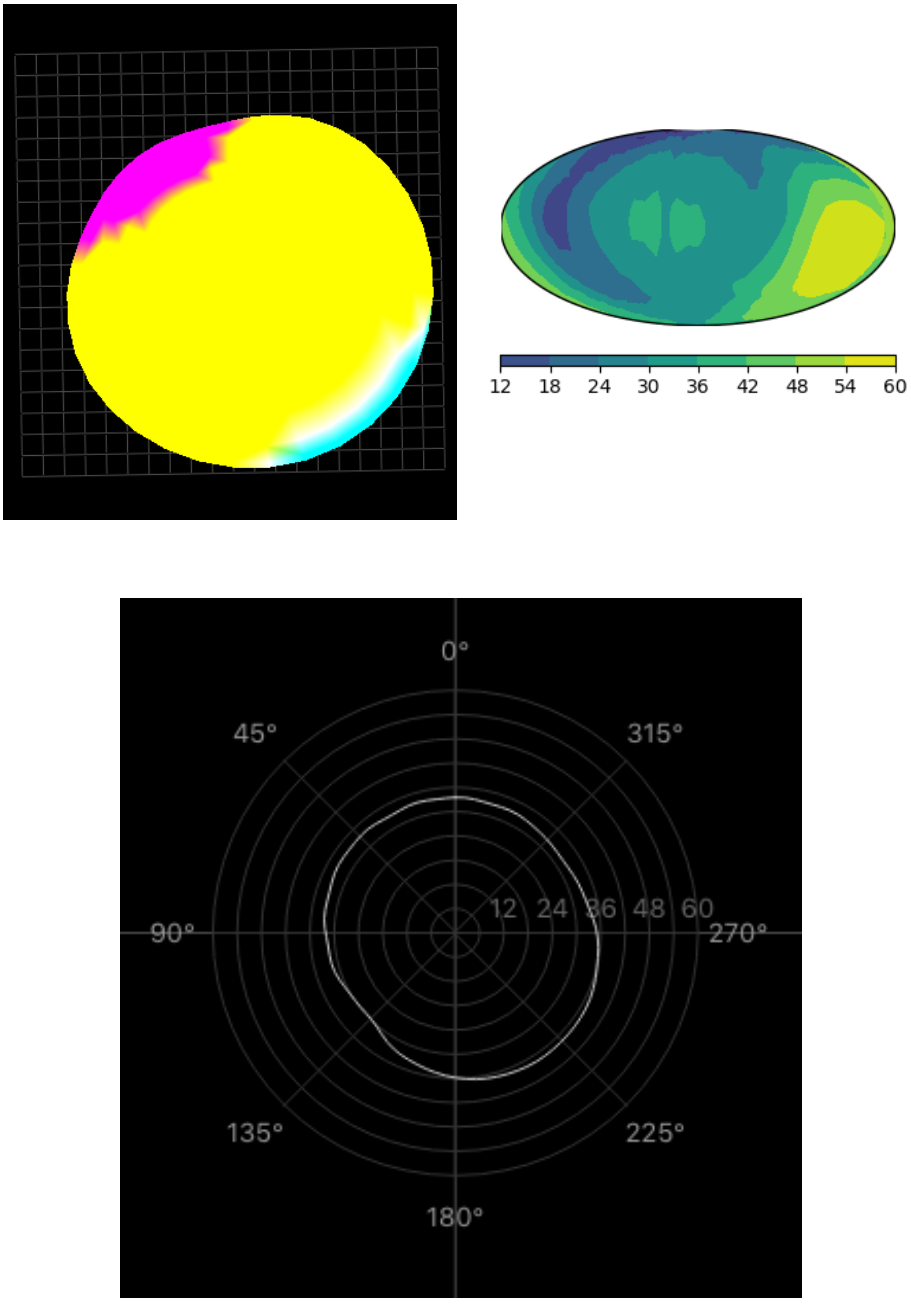


Abbildung 10: Diag: IKO1, 43Hz

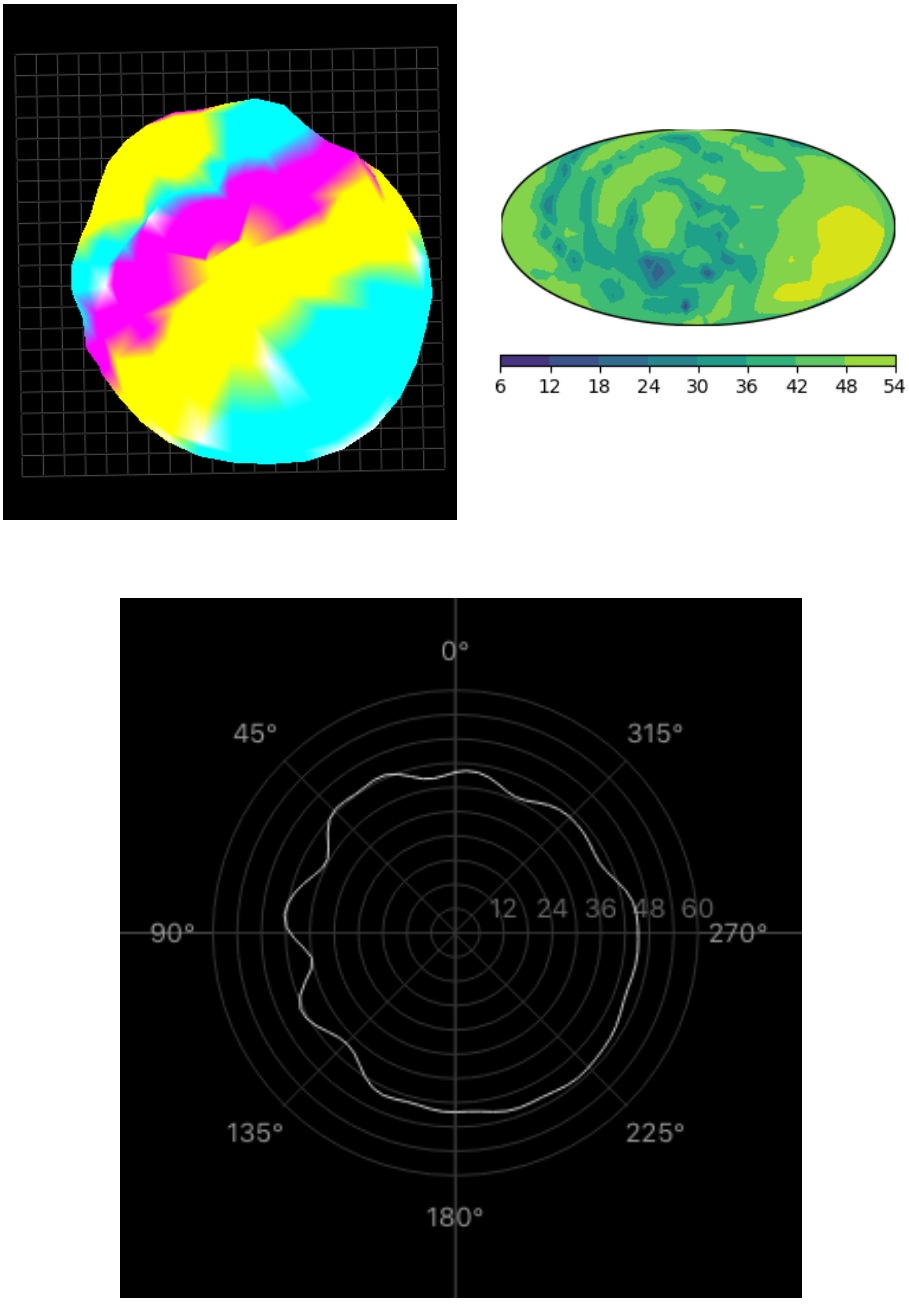


Abbildung 11: Diag: IKO1, 990 Hz

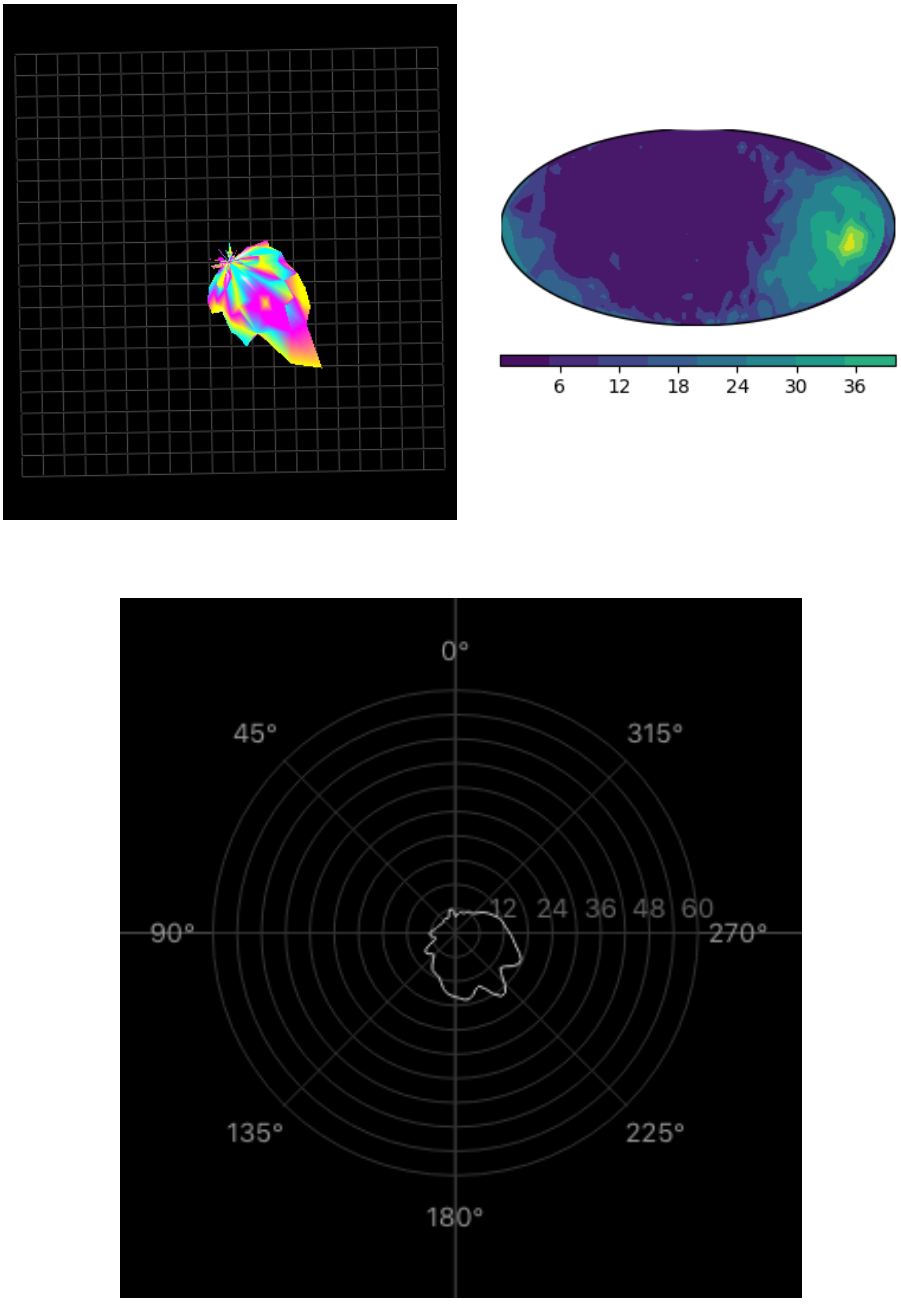


Abbildung 12: Diag: IKO1, 9991 Hz

A.2 IKO3

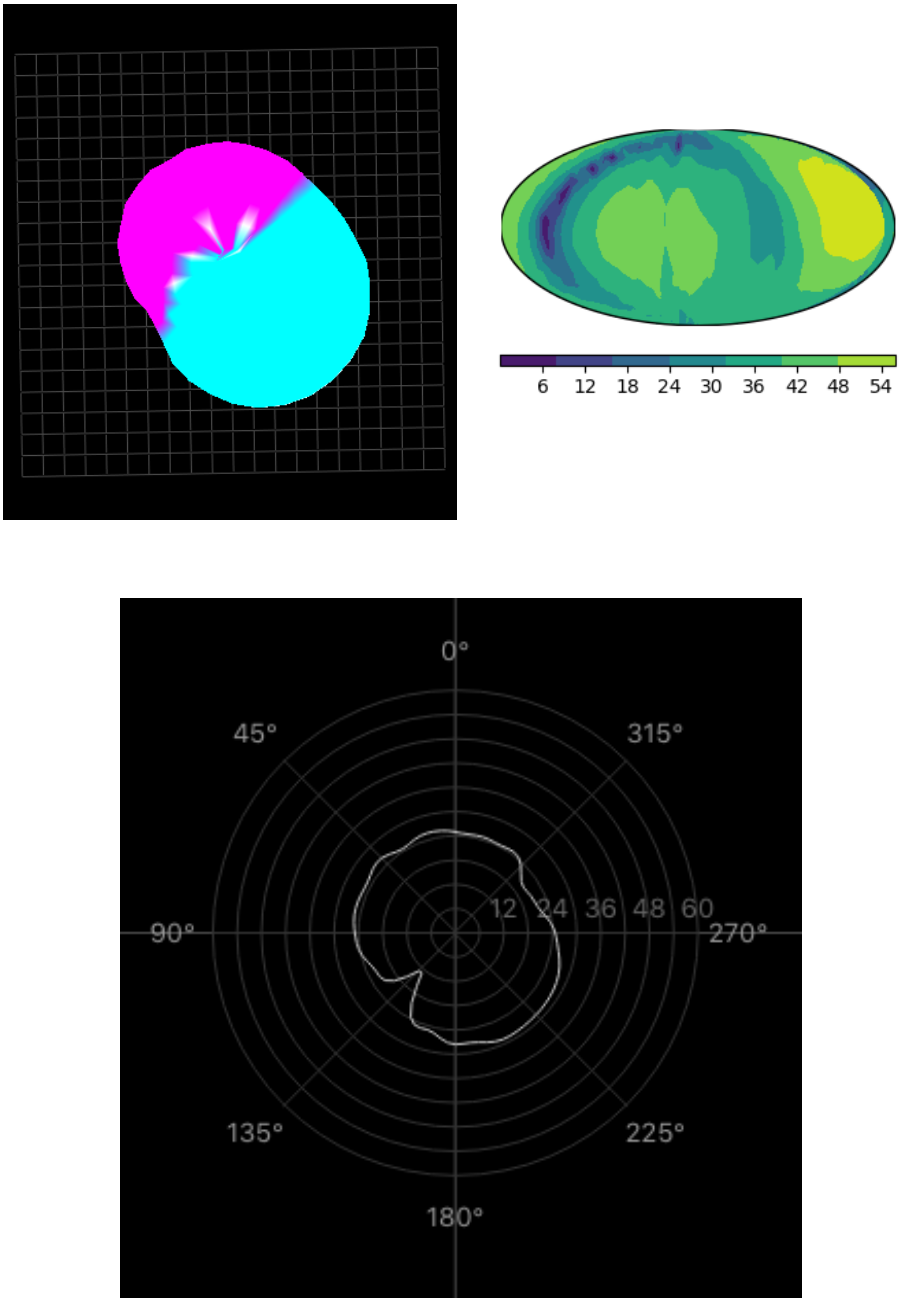


Abbildung 13: Diag: IKO3, 43Hz

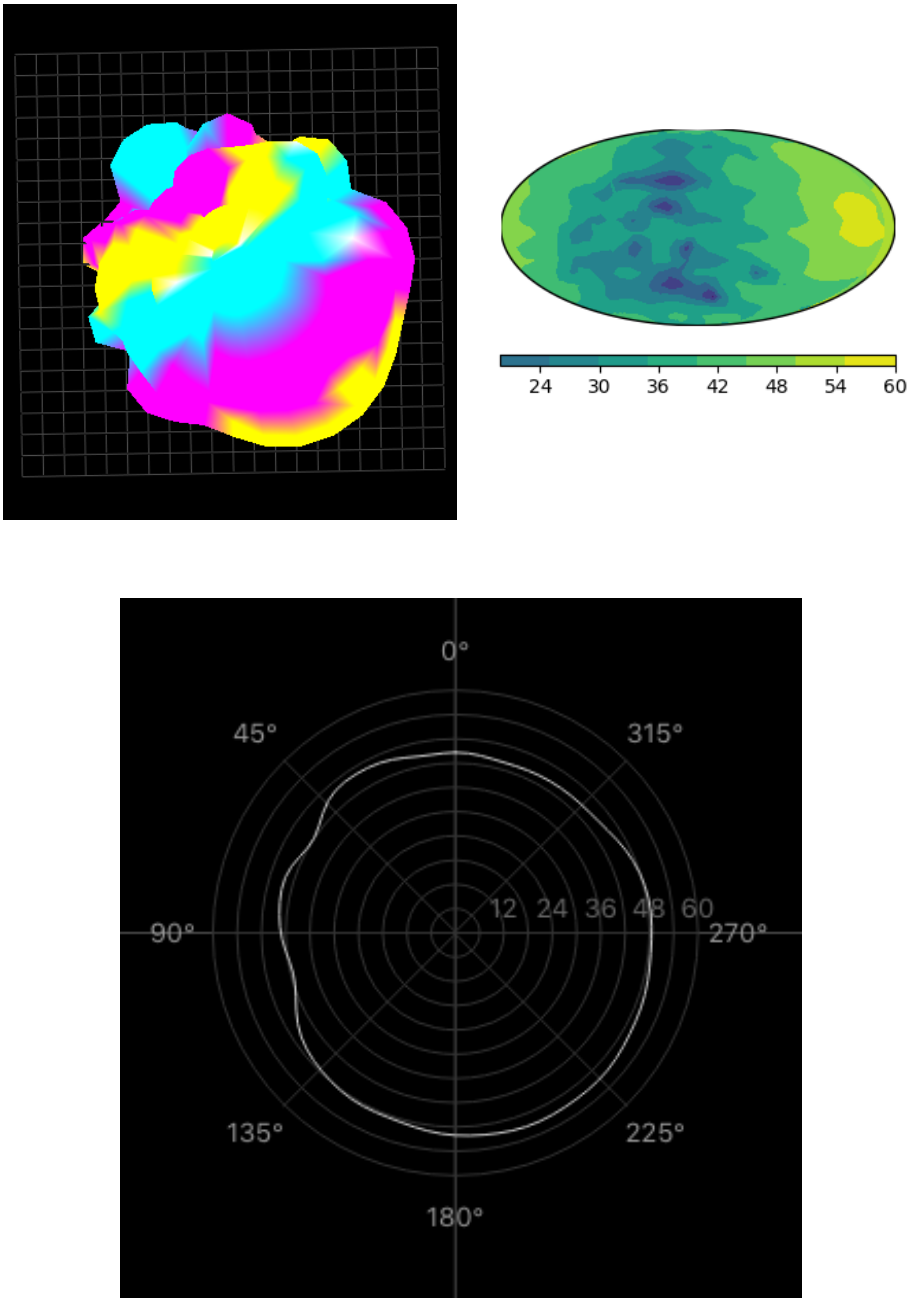


Abbildung 14: Diag: IKO3, 990 Hz

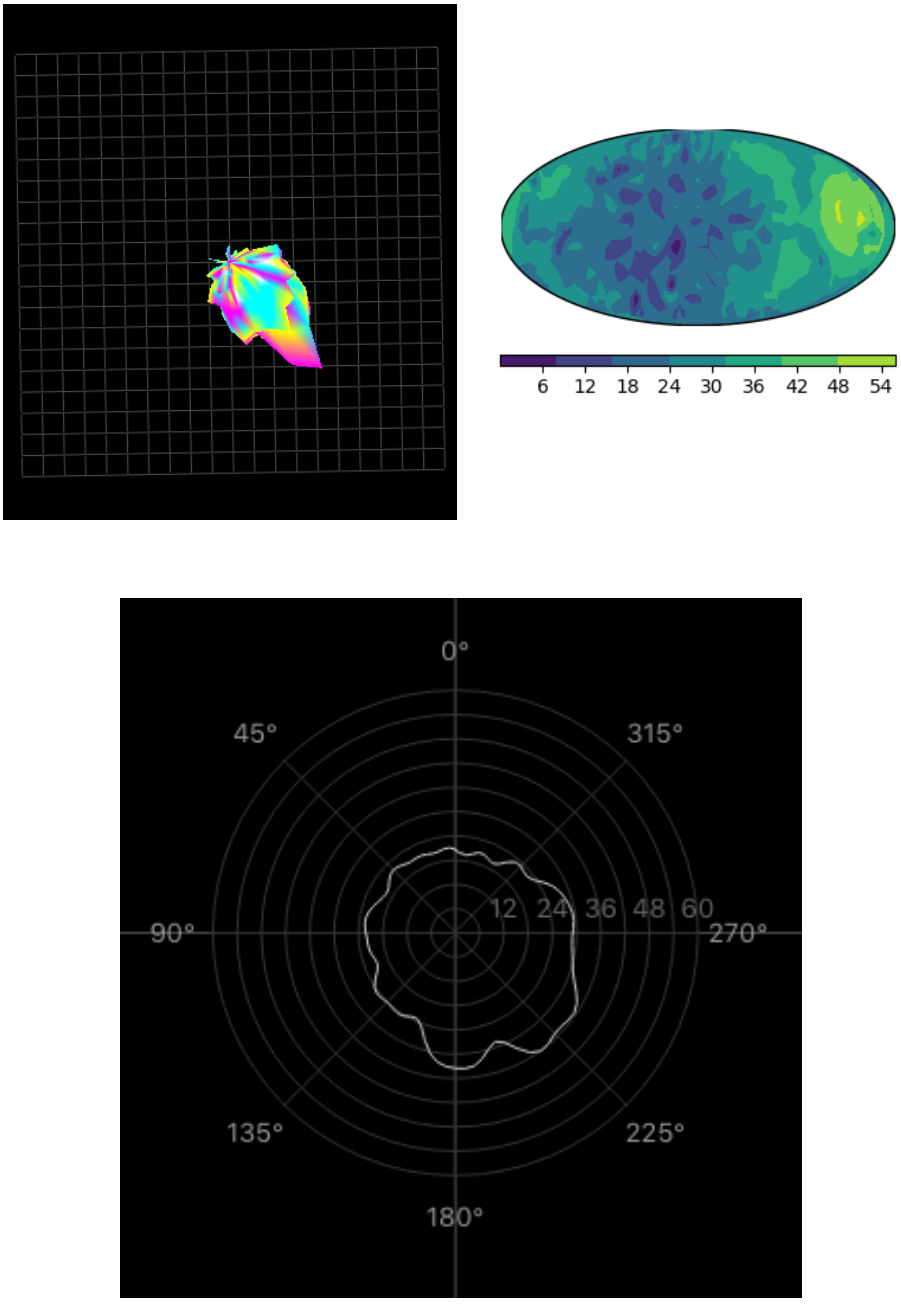


Abbildung 15: Diag: IKO3, 9991 Hz

A.3 HRIRs

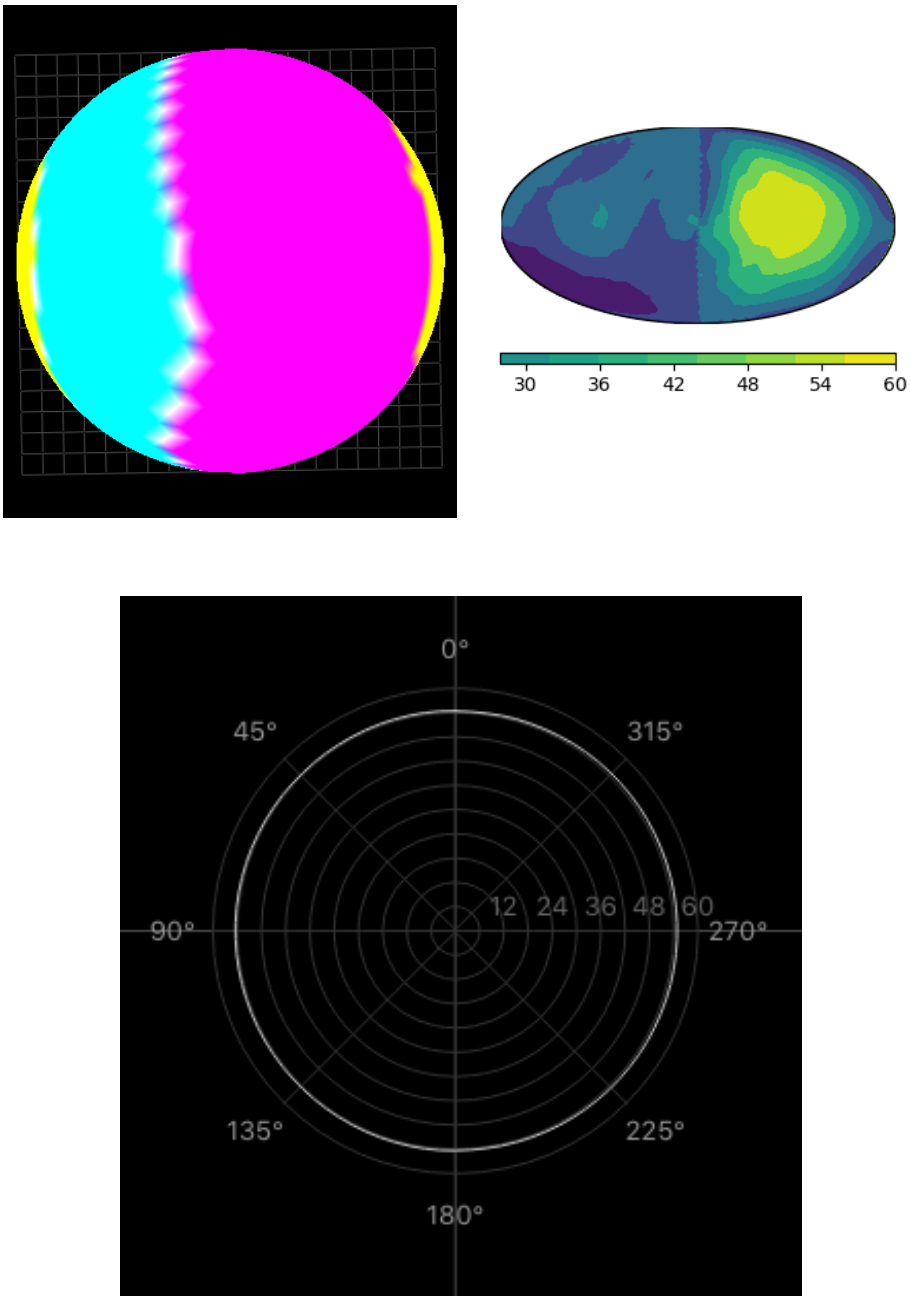


Abbildung 16: Diag: KEMAR Datensatz, HRIR L2702 , 1500Hz

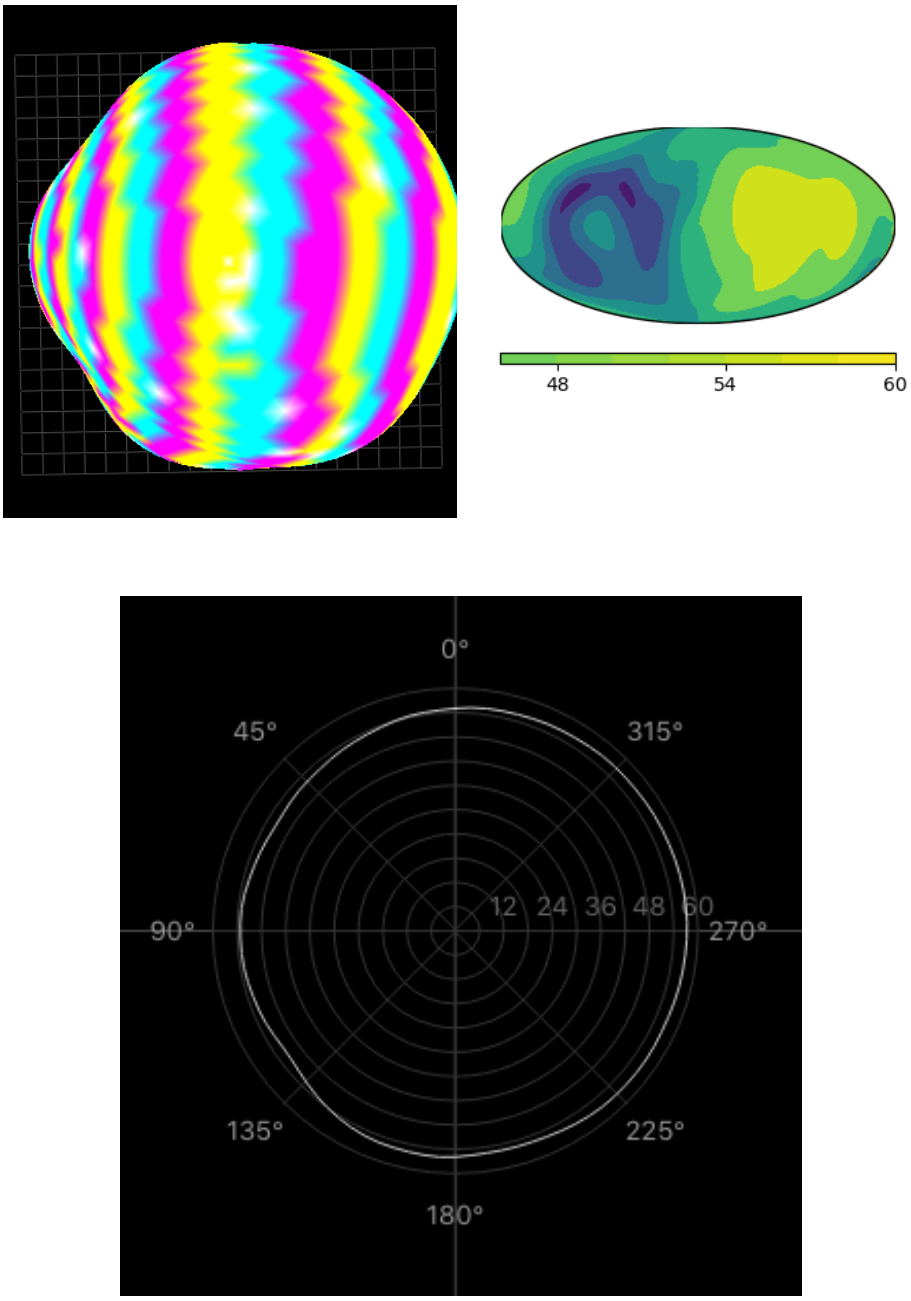


Abbildung 17: Diag: KEMAR Datensatz, HRIR L2702 , 9750Hz

B Programmcode

```
# 3D VisGUI
# Toningenieursprojekt Korbinian Wegler, 02.2020
# Funktionierende Conda-Environment: spydernew_3DGUI
# Python: 3.7.0
# OS: macOS 10.13
#
# Benutzeroberfläche erstellt mit QtDesigner:
# Form implementation generated from reading ui file '
                                    GUI_Widget_ohneTab.ui'
# Created by: PyQt5 UI code generator 5.7
# pyuic5 -x GUI_Widget_ohneTab.ui -o 3DVisGui_Window.py




from PyQt5 import QtCore, QtGui, QtWidgets
import numpy as np
import sys
import os
import pyqtgraph as pg
from pyqtgraph.opengl import GLViewWidget
from pyqtgraph import opengl as gl
import scipy.io as sio
from scipy.spatial import ConvexHull
import scipy.special as ssp
import netCDF4
import pyproj as pp
from mpl_toolkits.basemap import Basemap
#import cmocean
#import matplotlib.cm as cm
#from phasecolor import phasecolor
import matplotlib.pyplot as plt
import matplotlib.cm as cm
#from getcolorsfrommpled1 import cmapToColormap
#import getcolorsfrommpl
from matplotlib.figure import Figure
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as
                                     FigureCanvas
#from matplotlib.backends.backend_qt5agg import
                                    NavigationToolbar2QT as
                                    NavigationToolbar
from timeit import default_timer as timer
#from pgcbar import ColorBar
```

```
#from InfiniteLine import InfiniteLin

class MatplotlibWidget(QtWidgets.QWidget):
    def __init__(self, parent=None ,*args, **kwargs):
        super(MatplotlibWidget, self).__init__(parent)
        self.figure = Figure(*args,figsize=(2, 2), **kwargs)

        self.canvas = FigureCanvas(self.figure)
        self.layout = QtWidgets.QGridLayout()
        self.layout.addWidget(self.canvas)
        self.setLayout(self.layout)




class Ui_Fensterl(QtGui.QMainWindow):



    def setupUi(self, Fensterl):
        Fensterl.setObjectName("Fensterl")
        Fensterl.resize(900, 700) #858, 732
        self.centralwidget = QtWidgets.QWidget(Fensterl)
        self.centralwidget.setMinimumSize(QtCore.QSize(858, 689))
        self.centralwidget.setObjectName("centralwidget")
        self.gridLayout = QtWidgets.QGridLayout(self.centralwidget)
        self.gridLayout.setObjectName("gridLayout")
        self.frame = QtWidgets.QFrame(self.centralwidget)
        self.frame.setObjectName("frame")
        self.gridLayout_2 = QtWidgets.QGridLayout(self.frame)
        self.gridLayout_2.setObjectName("gridLayout_2")
        self.tabWidget = QtWidgets.QTabWidget(self.frame)
        self.tabWidget.setObjectName("tabWidget")
        self.BALLOON = QtWidgets.QWidget()
        self.BALLOON.setMinimumSize(QtCore.QSize(312, 306))
        self.BALLOON.setObjectName("BALLOON")
        self.gridLayout_5 = QtWidgets.QGridLayout(self.BALLOON)
        self.gridLayout_5.setContentsMargins(0, 0, 0, 0)
        self.gridLayout_5.setObjectName("gridLayout_5")
        self.graphicsView_4 = GLViewWidget(self.BALLOON)
        self.graphicsView_4.setObjectName("graphicsView_4")
        self.gridLayout_5.addWidget(self.graphicsView_4, 0, 0, 1, 1
```

```
                                        )
        self.tabWidget.addTab(self.BALLOON, "")

        self.gridLayout.addWidget(self.frame, 0, 0, 1, 1)

        self.MAP = QtWidgets.QWidget()


        self.graphicsView_3 = MatplotlibWidget(self)
        self.graphicsView_3.setObjectName("graphicsView_3")


        self.tabWidget.addTab(self.graphicsView_3, "MapTab") ##


        self.gridLayout_2.addWidget(self.tabWidget, 0, 1, 1, 1)
        self.verticalLayout = QtWidgets.QVBoxLayout()
        self.verticalLayout.setObjectName("verticalLayout")
        self.SpinBox_arraynr = QtWidgets.QSpinBox(self.frame)
        self.SpinBox_arraynr.setObjectName("spinBox")
        self.verticalLayout.addWidget(self.SpinBox_arraynr)
        self.Knopf = QtWidgets.QPushButton(self.frame)
        self.Knopf.setObjectName("Knopf")
        #self.Knopf.setText('Load dummy data!')
       # self.Knopf.clicked.connect(self.dummyload)
        self.verticalLayout.addWidget(self.Knopf)
        self.Slider_InterpolationOrder = QtWidgets.QSlider(self.
                                        frame)
        self.Slider_InterpolationOrder.setMinimumSize(QtCore.QSize(
                                        263, 22))
        self.Slider_InterpolationOrder.setOrientation(QtCore.Qt.
                                        Horizontal)
        self.Slider_InterpolationOrder.setTickPosition(QtWidgets.
                                        QSlider.TicksBelow)
        self.Slider_InterpolationOrder.setObjectName("
                                        Slider_InterpolationOrder
                                        ")
        self.verticalLayout.addWidget(self.
                                        Slider_InterpolationOrder
                                        )
        self.Slider_frequency = QtWidgets.QSlider(self.frame)
        self.Slider_frequency.setMinimumSize(QtCore.QSize(263, 0))
        self.Slider_frequency.setOrientation(QtCore.Qt.Horizontal)
        self.Slider_frequency.setTickPosition(QtWidgets.QSlider.
```

```
                                        TicksAbove)
        self.Slider_frequency.setObjectName("Slider_frequency")
        self.verticalLayout.addWidget(self.Slider_frequency)
        self.label = QtWidgets.QLabel(self.frame)
        self.label.setObjectName("label")
        self.verticalLayout.addWidget(self.label)
        self.gridLayout_2.addLayout(self.verticalLayout, 1, 0, 1, 1
                                        )
        self.verticalLayout_2 = QtWidgets.QVBoxLayout()
        self.verticalLayout_2.setObjectName("verticalLayout_2")
        self.pushButton = QtWidgets.QPushButton(self.frame)
        self.pushButton.setObjectName("pushButton")
        self.verticalLayout_2.addWidget(self.pushButton)
        self.label_frequency = QtWidgets.QLabel(self.frame)
        self.label_frequency.setObjectName("label_frequency")
        self.verticalLayout_2.addWidget(self.label_frequency)
        self.gridLayout_2.addLayout(self.verticalLayout_2, 1, 1, 1,
                                        1)
        self.horizontalLayout = QtWidgets.QHBoxLayout()
        self.horizontalLayout.setObjectName("horizontalLayout")
        self.POLARPLOT = QtWidgets.QWidget(self.frame)
        self.POLARPLOT.setObjectName("POLARPLOT")
        self.gridLayout_3 = QtWidgets.QGridLayout(self.POLARPLOT)
        self.gridLayout_3.setContentsMargins(0, 0, 0, 0)
        self.gridLayout_3.setObjectName("gridLayout_3")
        self.verticalSlider = QtWidgets.QSlider(self.POLARPLOT)
        self.verticalSlider.setOrientation(QtCore.Qt.Vertical)
        self.verticalSlider.setTickPosition(QtWidgets.QSlider.
                                        TicksBelow)
        self.verticalSlider.setObjectName("verticalSlider")
        self.gridLayout_3.addWidget(self.verticalSlider, 0, 0, 1, 1
                                        )
        self.graphicsView_Polar = pg.PlotWidget(self.POLARPLOT)
        self.graphicsView_Polar.setObjectName("pgPlotWidget")
        self.gridLayout_3.addWidget(self.graphicsView_Polar, 0, 1,
                                        1, 1)
        self.horizontalLayout.addWidget(self.POLARPLOT)
        self.gridLayout_2.addLayout(self.horizontalLayout, 0, 0, 1,
                                        1)
        self.gridLayout.addWidget(self.frame, 0, 0, 1, 1)
        Fensterl.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(Fensterl)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 858, 22))
        self.menubar.setObjectName("menubar")
```

```
        self.menuMenu = QtWidgets.QMenu(self.menubar)
        self.menuMenu.setObjectName("menuMenu")
        Fensterl.setMenuBar(self.menubar)
        self.statusbar = QtWidgets.QStatusBar(Fensterl)
        self.statusbar.setObjectName("statusbar")
        Fensterl.setStatusBar(self.statusbar)
        self.actionOpen = QtWidgets.QAction(Fensterl)
        self.actionOpen.setObjectName("actionOpen")
        self.actionQuit = QtWidgets.QAction(Fensterl)
        self.actionQuit.setObjectName("actionQuit")
        self.menuMenu.addAction(self.actionOpen)
        self.menuMenu.addAction(self.actionQuit)
        self.menubar.addAction(self.menuMenu.menuAction())


        self.Knopf.setObjectName("Knopf")
        self.Knopf.setText('Load data!')
        self.Knopf.clicked.connect(self.file_open)
        self.verticalLayout.addWidget(self.Knopf)



        self.tabWidget.currentChanged.connect(self.drawMap)
        self.statusbar = QtWidgets.QStatusBar(Fensterl)
        self.statusbar.setObjectName("statusbar")
        Fensterl.setStatusBar(self.statusbar)

# Menubar
        self.menubar = QtGui.QMenuBar(Fensterl)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 858, 22))
        self.menubar.setObjectName("menubar")
        self.menuMenu = QtGui.QMenu(self.menubar)
        self.menuMenu.setObjectName("menuMenu")
        self.setMenuBar(self.menubar)


        self.actionOpen = QtGui.QAction("&Open File", Fensterl)
        self.actionOpen.setShortcut("Ctrl+o")
        self.actionOpen.setStatusTip('Open File')
        self.actionOpen.triggered.connect(self.file_open)


        self.actionQuit = QtGui.QAction("&End", Fensterl)
        self.actionQuit.setShortcut("Ctrl+q")
```

```
        self.actionQuit.setStatusTip('Quit')
        self.actionQuit.triggered.connect(self.close_application)

        self.actionDummy = QtGui.QAction("&Dummy", Fensterl)
        self.actionDummy.setShortcut("Ctrl+d")
        self.actionDummy.setStatusTip('Dummy Data')
        self.actionDummy.triggered.connect(self.dummyload)


        self.menuMenu.addAction(self.actionOpen)
        self.menuMenu.addAction(self.actionQuit)
        self.menuMenu.addAction(self.actionDummy)
        self.menubar.addAction(self.menuMenu.menuAction())


## Some init
        self.input = np.zeros(0)
        self.retranslateUi(Fensterl)
        self.tabWidget.setCurrentIndex(0)
        self.filename = '_'
        self.tabWidget.currentChanged.connect(self.
                                              frequency_changed)

        self.verticalSlider.setMaximum(180)
        self.verticalSlider.setValue(90)

 #        self.colmap = cmapToColormap(getcolorsfrommpl.test_cm)
#        self.colmap = cmapToColormap()
        self.pushButton.clicked.connect(self.export)



    def file_open(self):
        # Öffnet Daten und überprüft sie auf deren Kompatibilität
        # Auslesen und Normierung der Messdaten
        # Aufbereitung der geometrischen Anordnung
        self.name = QtGui.QFileDialog.getOpenFileName(self, 'Open
                                              File')[0]
        filename, fileext  =  os.path.splitext(self.name)

        if fileext.lower().endswith('.mat'):
            inp = sio.loadmat(self.name)
            ## Umformatieren
            inpmat = inp['hall']
```

```
            s1, s2 = inpmat.shape[:2]
            self.inpmat = inpmat
            inpmatreord = np.resize(inpmat,(s1,s2,inpmat.shape[2]*
                                            inpmat.shape[3]))
            inpmatreord = np.swapaxes(inpmatreord,0,2)
            self.inpmatreord = inpmatreord

            self.meas_grid = np.fliplr(inp['mic_zenith_azimuth'][:]
                                            )

            self.meas_grid = np.c_[self.meas_grid,np.ones(self.
                                            meas_grid.shape[0])]
            self.meas_grid[:,0] = self.meas_grid[:,0]
            self.meas_grid[:,1] = self.meas_grid[:,1]-90
            self.input = np.fft.fft(inpmatreord,1024)[:,:,1:int(
                                            inpmatreord.shape[2]/
                                            2)]
            self.samplerate = int(inp['fs'])

            self.title = inp['prefix'][0]
            print('!!!Filetype: .mat!!!')

        elif fileext.lower().endswith(('.Sofa', '.SOFA', '.sofa','.
                                        SoFA')):

            file = netCDF4.Dataset(self.name)


        ### self.input is FFT of Inputmatrix: arraygrid x
                                        measurementgrid x
                                        frequency
            inp = file.variables['Data.IR'][:,:,:]
            self.input = np.fft.fft(inp,1024)[:,:,1:int(inp.shape[2
                                            ]/2)]
            self.samplerate = file.variables['Data.SamplingRate'][0
                                            ]
#
#            Ensure grids being spherical:
            if file.variables['ReceiverPosition'].Type == '
                                            spherical':
                self.receiver_grid = file.variables['
                                                ReceiverPosition'
                                                ][:]
                if (np.max(self.receiver_grid[:,1]) >= 91 and np.
```

```
                                        min(self.
                                        receiver_grid[:,1
                                        ])> -1):
                self.receiver_grid[:,1] = self.receiver_grid[:,
                                            1] - 90
            else:
                self.receiver_grid[:,1] = self.receiver_grid[:,
                                            1]

        else:

            recpos = np.transpose(np.squeeze(np.ma.copy(file.
                                        variables['
                                        ReceiverPosition'
                                        ][:])))
            self.receiver_grid = self.cart2sph(recpos[0,:],
                                        recpos[1,:],
                                        recpos[2,:])


        if file.variables['SourcePosition'].Type == 'spherical'
                                            :
            self.source_grid = file.variables['SourcePosition']
                                        [:]
            if (np.max(self.source_grid[:,1]) and np.min(self.
                                        source_grid[:,1])
                                         > -1):
                self.source_grid[:,1] = self.source_grid[:,1] -
                                            90
            else:
                self.source_grid[:,1] = self.source_grid[:,1]

        else:
            srcpos = np.transpose(np.squeeze(np.ma.copy(file.
                                        variables['
                                        SourcePosition'][
                                        :])))
            self.source_grid = self.cart2sph(srcpos[0,:],srcpos
                                        [1,:],srcpos[2,:]
                                        )


        # Weiter aussen liegender Punkt ist Messpunkt
```

```
            if np.mean(self.receiver_grid,0)[2]> np.mean(self.
                                                source_grid,0)[2]:
                self.meas_grid = np.squeeze(self.receiver_grid)
                self.input = np.transpose(self.input,(1,0,2))

            else:
                self.meas_grid = np.squeeze(self.source_grid)


            # if file.variables['SourcePosition'].Type == '
                                                cartesian':
            #     self.meas_grid = self.cart2sph(self.meas_grid)
            # else:
            #     self.meas_grid = self.meas_grid
#             elif not self.checkBox.isChecked():
            self.title = file.Title

            #print('notchecked')


        else:
            print('File format not supported')


        self.frequs = np.fft.fftfreq(inp.shape[2],1/(self.
                                            samplerate))[1:int(inp.
                                            shape[2]/2)]

        # Eingang normiert auf Maximalwert und ausgegeben mit 60dB
                                            Dynamik
        self.inpnorm = 20 * np.log10(abs(self.input)/np.amax(abs(
                                            self.input))) # 20 *

        # Anpassung Wertebereich 0 - dBmax = 60dB
        self.inpnorm = self.inpnorm + 60  #
        self.inpnorm =   np.where(self.inpnorm < 0 , 0 , self.
                                            inpnorm)


        self.statusbar.showMessage(self.title)

        self.filename = os.path.basename(self.name)
        self.label.setText(QtCore.QCoreApplication.translate("
                                            Fensterl", self.filename)
```

```
                                        )



        # Translate Maxs/Mins
        self.Slider_frequency.setMaximum(self.input.shape[-1]-1)
        self.SpinBox_arraynr.setMaximum(self.input.shape[1])
        self.SpinBox_arraynr.setMinimum(1)
        self.Slider_InterpolationOrder.setMaximum(int(np.sqrt(self.
                                            input.shape[0])))

        if (int(np.sqrt(self.inpnorm.shape[0]))<18):
            self.Slider_InterpolationOrder.setValue(int(np.sqrt(
                                                self.inpnorm.shape[0]
                                                ))-1)
        else:
            self.Slider_InterpolationOrder.setValue(18)

    #  self.Slider_InterpolationOrder.setValue(int(np.sqrt(self.
                                        input.shape[0]))-1)

        # Connect Faders to functions

        self.Slider_frequency.valueChanged.connect(self.
                                            frequency_changed)
        self.SpinBox_arraynr.valueChanged.connect(self.
                                            ReDrawBalloon)
        self.SpinBox_arraynr.valueChanged.connect(self.
                                            frequency_changed)

        self.Slider_InterpolationOrder.valueChanged.connect(self.
                                            drawPolar)

        self.Slider_InterpolationOrder.valueChanged.connect(self.
                                            redrawPolar)

        self.verticalSlider.setMaximum((np.amax(self.meas_grid[:,1]
                                            )+90)/10)
        self.verticalSlider.setMinimum((np.amin(self.meas_grid[:,1]
                                            )+90)/10)
        self.verticalSlider.setValue(int((np.amax(self.meas_grid[:,
                                            1])+np.amin(self.
                                            meas_grid[:,1])+90)/10))
```

```
        self.verticalSlider.valueChanged.connect(self.drawPolar)
        self.verticalSlider.valueChanged.connect(self.redrawPolar)
        self.verticalSlider.setTickInterval(1)


        # Funktionenaufruf
        if self.BALLOON.isVisible():
            self.balloonrender(self, Fensterl.graphicsView_4)
        if self.graphicsView_3.isVisible():

            self.drawMap()

        self.drawPolar()
        self.frequency_changed()


    def close_application(self):

        QtCore.QCoreApplication.instance().quit()



    def dummyload(self):

        print('Beginn dummyload', timer())
        self.name = os.path.realpath('HRIR_L2702.sofa')

        file = netCDF4.Dataset(self.name)

        inp = file.variables['Data.IR'][:,:,:]
        self.input = np.fft.fft(inp,1024)[:,:,1:int(inp.shape[2]/2)
                                          ]
        # Eingan normiert auf Maximalwert und ausgegeben mit 60dB
                                          Dynamik
        self.inpnorm = 20* np.log10(abs(self.input)/np.amax(abs(
                                          self.input)))
        # Anpassung Wertebereich 0 - dBmax = 60dB
        self.inpnorm = self.inpnorm + 60  #
        self.inpnorm =   np.where(self.inpnorm < 0 , 0 , self.
                                          inpnorm)

        self.input = np.fft.fft(inp)[:,:,1:int(inp.shape[2]/2)]

        self.samplerate = file.variables['Data.SamplingRate'][0]
```

```
        self.meas_grid = file.variables['SourcePosition'][:]

        self.title = file.Title
        self.filename = os.path.basename(self.name)
        self.statusbar.showMessage(self.title)
        self.label.setText(QtCore.QCoreApplication.translate("
                                        Fensterl", self.filename)
                                        )
        self.frequs = np.fft.fftfreq(inp.shape[2],1/(self.
                                        samplerate))[1:int(inp.
                                        shape[2]/2)]



        self.Slider_frequency.setMaximum(self.input.shape[-1]-1)

        self.SpinBox_arraynr.setMaximum(self.input.shape[1])
        self.SpinBox_arraynr.setMinimum(1)
        self.Slider_InterpolationOrder.setMaximum(int(np.sqrt(self.
                                        inpnorm.shape[0])))
        if (int(np.sqrt(self.inpnorm.shape[0]))<25):
            self.Slider_InterpolationOrder.setValue(int(np.sqrt(
                                            self.inpnorm.shape[0]
                                            ))-1)
        else:
            self.Slider_InterpolationOrder.setValue(25)
        self.Slider_frequency.valueChanged.connect(self.
                                        frequency_changed)
        self.SpinBox_arraynr.valueChanged.connect(self.
                                        ReDrawBalloon)
        self.SpinBox_arraynr.valueChanged.connect(self.
                                        frequency_changed)
        self.SpinBox_arraynr.setMaximum(self.input.shape[1])

        self.Slider_InterpolationOrder.valueChanged.connect(self.
                                        drawPolar)

        self.verticalSlider.setMaximum((np.amax(self.meas_grid[:,1]
                                        )+90)/10)
        self.verticalSlider.setMinimum((np.amin(self.meas_grid[:,1]
                                        )+90)/10)
        self.verticalSlider.setTickInterval(1)
        self.verticalSlider.setValue(int((np.amax(self.meas_grid[:,
```

```
                                        1])+np.amin(self.
                                        meas_grid[:,1])+90)/10))
        self.verticalSlider.valueChanged.connect(self.drawPolar)



        self.balloonrender(self, Fensterl.graphicsView_4)
        self.drawPolar()
        self.drawMap()
        self.frequency_changed()
        print('Ende dummyload', timer())

#data generation
# calculation # Input format: frequency x Arraygridnr x
                                    measurementgridrow x
                                    measurementgridcol



    def drawMap(self):

        start_drawmap = timer()


        input_mat = self.inpnorm[:,self.arraynbr,self.frequency]
        r_grid = self.meas_grid

        self.mollgrid = pp.Proj(proj='moll',ellps='sphere',a=1,b=1)

        self.x,self.y = self.mollgrid(r_grid[:,0],r_grid[:,1]) #
        self.fig =   self.graphicsView_3.figure
        self.fig.clear()
        self.ax = self.fig.add_subplot(111)

       # x,y  = meshgrid(self.x,self.y)

        self.m = Basemap( projection = 'moll',lon_0=0  ,resolution=
                                        'c', ax=self.ax, rsphere
                                        =1,llcrnrlon=-180,
                                        llcrnrlat=-90, urcrnrlon=
                                        180 , urcrnrlat=90)

        self.cmap = plt.get_cmap('viridis') #'PiYG'
       # self.colormesh = self.m.pcolormesh(self.x, self.y,
```

```
                                        input_mat, vmin = 0, vmax
                                        = 60, cmap=Fensterl.cmap)
        self.cf = self.m.contourf(x=self.x+self.m.urcrnrx/2, y=-
                                         self.y+self.m.urcrnry/2,
                                         extend= 'neither', data=
                                         input_mat, cmap=self.cmap
                                         , tri = True) #plt.cm.jet
        ##self.pcme = plt.pcolormesh(cmap=self.cmap, vmin=None,
                                         vmax=None, shading='flat
                                         ', antialiased=False,
                                         data=input_mat)

        self.cf.set_clim(0,60)
        self.cb = self.m.colorbar(self.cf  ,fig=self.fig, ax=self.
                                         ax, location='bottom',
                                         pad ='15%',shrink = 2 )
       # self.cb = self.cf.colorBar
#         self.cb.set_ticks([8,16,24,32,40,48,56])
        self.cb.set_ticks([6,12,18,24,30,36,42,48,54,60])
        self.fig.canvas.draw()

        drawmap_time = timer() - start_drawmap
        print('drawmap_time: ', drawmap_time)


    def redrawMap(self):
        start_redrawmap = timer()

        input_mat = self.inpnorm[:,self.arraynbr,self.frequency]
        r_grid = self.meas_grid

#
        self.x,self.y = self.mollgrid(-r_grid[:,0],-r_grid[:,1])



        self.cf = self.m.contourf(x=self.x+self.m.urcrnrx/2, y=-
                                         self.y+self.m.urcrnry/2,
                                         data=input_mat, extend= '
                                         neither', cmap=self.cmap,
                                          tri = True, vmin=0, vmax
                                          = 60) #plt.cm.jet
        self.cf.changed()
        self.cf.set_clim(0,60)
```

```
        self.cb.update_normal(self.cf)
#Fensterl.cb.on_mappable_changed()

        self.fig.canvas.draw()

        redrawmap_time = timer() - start_redrawmap
        print('redrawmap_time: ', redrawmap_time)



    def drawPolar(self):
        start_drawpolar = timer()

        ### Variable Init
        self.frequency = self.Slider_frequency.value()
        self.arraynbr = self.SpinBox_arraynr.value()-1
        self.Slider_InterpolationOrder.setMinimum(1)

        order = self.Slider_InterpolationOrder.value()

        layer = self.verticalSlider.value()*10/180*np.pi # -90...90
                                            /180*pi = -pi/2... pi/2


        self.d = self.inpnorm[:,self.arraynbr,self.frequency] ##
                                                    Datenvektor d (N x
                                            1 complex) N= 584
        self.Y = np.zeros(shape = [self.d.shape[0],order+1,order+1]
                                           , dtype=complex)
        self.V = self.meas_grid[:,:2]*np.pi/180 # Richtungsvektoren
                                            V (N x 2): Theta und Phi
                                            für jeden Messpunkt.
        self.V[:,0] = self.V[:,0]#+np.pi
        self.V[:,1] = self.V[:,1]+np.pi/2


        ## maximum order

####    Spherical Harmonics für jeden Richtungsvektor (Messpunkt)
        timersphdegord_start = timer()
        for degree in range(order):

            for order_i in range(0,degree+1):
                self.Y[:,degree,order_i] = np.squeeze(ssp.sph_harm(
```

```
                                                    order_i,degree,
                                                    self.V[:,0],self.
                                                    V[:,1]))
        print('timersphdegord Y : ', timer()-timersphdegord_start)

        linalgtime_start = timer()
        self.Y = np.reshape(self.Y,[self.Y.shape[0],-1])
###     Berechnung der Abmisonics Darstellung jedes einzelnen
                                    Datenpunktes
###     Y = sphericalHarmonics(order,V) ... Y ist N x (order+1)^2
                                    Matrix  Ambi-Darstellung der
                                    Datenpunkte
        self.iY = np.linalg.pinv(self.Y)    #iY = pinv(Y) ...
                                            pseudo-inverse von Y [ (
                                            order+1)^2 x N ]
#        self.y = np.dot(self.iY/np.amax(self.iY) ,self.d)  #y = iY
                                     * d ... SH Koeffizienten Vektor
                                    [ (order+1)^2 x 1 ]
        self.y = np.dot(self.iY ,self.d)  #y = iY * d ... SH
                                            Koeffizienten Vektor [ (
                                            order+1)^2 x 1 ]
        #  y ...  spherical harmonics Datenvektor
        print('drawpolar linalgtime: ', timer() - linalgtime_start
                                            )

###     Berechnung der Auswertungspunkte in Polardarstellung
        self.Q = np.zeros(shape = [91,2]) #Richtungsvektoren Q (L x
                                            2): Theta und Phi für
                                            gewünschte Richtungen L=
                                            180

        self.Q[:,0]= np.arange(0,2.01*np.pi,np.pi/45)

        self.Q[:,1] = layer


        print('drawPolar: Maximaler Grad und aktueller Grad ', int(
                                            np.sqrt(Fensterl.V.shape[
                                            0])), order,  'layer ',
                                            layer)

###     Ambidarstellung der Auswertungspukte
        self.Z = np.zeros(shape = [self.Q.shape[0],order+1,order+1]
                                            , dtype=complex)
```

```
        # Z = sphericalHarmonics(order,Q) ... [ L x (order+1)^2  ]
                                          Ambi-Darstellung der
                                          Auswertungsrichtung
        timersphdegordZ_start = timer()
        for degree in range(order+1):
#             for order_i in range(-degree,degree+1,1):
            for order_i in range(0,degree+1):
                self.Z[:,degree,order_i] = ssp.sph_harm(order_i,
                                                        degree,self.Q[:,0
                                                        ],self.Q[:,1])
        print('timersphdegord Y : ', timer()-timersphdegordZ_start)
        self.Z = np.reshape(self.Z,[self.Z.shape[0],-1])

###     Berechnung: Betrag für jeden interpolierten Punkt. Jedes
                                    Grad...
        self.b = np.dot(self.Z,self.y)
        #b = L * y  (interpolierter Datenvektor L x 1 complex:
                                          360x1)
        rstarttime = timer()
        plot = self.graphicsView_Polar
        plot.clear()
        plot.setAspectLocked()
       # xachs = pg.AxisItem(orientation='bottom')
       # plot.removeItem(self.pol)  ## geht!

        # Add polar grid lines
        plot.addLine(x=0, pen=0.3)
        plot.addLine(y=0, pen=0.3)
        for agl in range(0, 360, 45):
            grline = pg.QtGui.QGraphicsLineItem(0, 0, 0, 60, )
            grline.rotate(agl)
            grline.setPen(pg.mkPen(0.2))
            plot.addItem(grline)

            text = 'text_' + str(agl)
            text = pg.TextItem((str(agl) + '°'), color=(150, 150,
                                              150), anchor=(0.5, 0.
                                              5))

            plot.addItem(text)
            text.setPos(-70*np.sin(agl*np.pi/180), 70*np.cos(agl*np
                                              .pi/180) )

       # plot.addItem(xachs)
```

```
        idx= 0
        for r in range(6, 61, 6):
            circle = pg.QtGui.QGraphicsEllipseItem(-r, -r, r*2, r*2
                                                   )
            circle.setPen(pg.mkPen(0.2))
            plot.addItem(circle, name = r)

            if ( idx % 2 ):
                text = 'text_' + str(r)
                text = pg.TextItem(str(r), color=(100, 100, 100),
                                                  anchor=(0.5, 1.0)
                                                  )
                plot.addItem(text)
                text.setPos(r, 0)
            idx +=1

        # make polar data

        theta = self.Q[:,0]
        self.radius = abs(self.b)
        self.radius[-1]=self.radius[0]

        # Transform to cartesian and plot
        x = -self.radius * np.sin(theta)
        y = self.radius * np.cos(theta)
        self.pol = plot.plot(x, y)
        print('plotpolar_plottime: ', timer() -  rstarttime)

        drawpolar_time = timer() - start_drawpolar
        print('drawpolar_time: ', drawpolar_time)
        #Polardarstellung: Theta = Q(:,1), Radius = abs(b),
                                          Füllfarbe = angle(b)

    def redrawPolar(self):
        start_redrawpolar = timer()
#         Datenvektor d (N x 1 complex) N= 584
#
#         Richtungsvektoren V (N x 2): Theta und Phi für jeden
                                  Messpunkt.
#
#         Y = sphericalHarmonics(order,V) ... Y ist N x (order+1)^2
                                  Matrix
#
#         iY = pinv(Y) ... pseudo-inverse von Y [ (order+1)^2 x N ]
```

```
#
#         y = iY * d ... SH Koeffizienten Vektor [ (order+1)^2 x 1 ]
#
#                              L = 180
#         Richtungsvektoren Q (L x 2): Theta und Phi für gewünschte
                                    Richtungen
#            (z.B Theta = 0,..2*pi; Phi = 0 für Schnitt am Horizont)
#
#         Z = sphericalHarmonics(order,Q) ... [ L x (order+1)^2  ]
#
#         b = L * y  (interpolierter Datenvektor L x 1 complex)
#
#         Polardarstellung: Theta = Q(:,1), Radius = abs(b),
                                    Füllfarbe = angle(b)
#
#
#         Die Ordnung hängt von der Anzahl und Verteilung der
                                    Messpunkte ab, sinnvoll ist order
                                     <= sqrt(N)-1
        self.frequency = self.Slider_frequency.value()
        self.arraynbr = self.SpinBox_arraynr.value()-1

###     Neuer Datenvektor roh
        self.d = self.inpnorm[:,self.arraynbr,self.frequency]
#        self.d = np.ones(self.d.shape)
###     Neuer Datenvektor in Ambidarstellung
#
#        self.y = np.dot(self.iY/np.amax(self.iY) ,self.d)
        self.y = np.dot(self.iY ,self.d)

#
                                    ============================================


#
#          # Debug   erzeugt Ambi 1. Ordnung Keulen
#          self.y = np.zeros(self.y.shape)
#          self.y[15] = 1
#
# #         self.y = np.ones(self.y.shape)
#          if self.y.size >= 16:
#              self.y[16] = 1
# #         self.y = np.identity(self.y.shape)
#
#
```

```
                                        ========================================

###     Interpolierter Datenvektor b

        self.b = np.dot(self.Z,self.y)
#


        plot = self.graphicsView_Polar

#         plot.setAspectLocked()
        plot.removeItem(self.pol)  ## entfernt vorherigen Plot

#
#         # make polar data
        theta = self.Q[:,0]
        self.radius = abs(self.b)
        self.polarphase = np.angle(self.b)
        self.polarcolor = self.colorize_polar(self.polarphase)
        # Transform to cartesian and plot
        x = -self.radius * np.sin(theta)
        y = self.radius * np.cos(theta)
       # pen = pg.mkPen(self.polarcolor)
        self.pol = plot.plot(x, y) #,pen

        redrawpolar_time = timer() - start_redrawpolar
        print('redrawpolar_time: ', redrawpolar_time)



    def drawBalloonempty(self, Fensterl):
#         print('drawBalloonempty durchgeführt')
        start_drawballoon = timer()
        rows=30
        cols=30
        radius =1
        verts = np.empty((rows+1, cols, 3), dtype=float)
        offset = False

        ## compute vertexes
        phi = (np.arange(rows+1) * np.pi / rows).reshape(rows+1, 1)
        s = radius * np.sin(phi)
        verts[...,2] = radius * np.cos(phi)
```

```
        th = ((np.arange(cols) * 2 * np.pi / cols).reshape(1, cols)
                                            )
        if offset:
            th = th + ((np.pi / cols) * np.arange(rows+1).reshape(
                                                rows+1,1))  ## rotate
                                                 each row by 1/2
                                                column
        verts[...,0] = s * np.cos(th)
        verts[...,1] = s * np.sin(th)
        verts = verts.reshape((rows+1)*cols, 3)[cols-1:-(cols-1)]
                                            ## remove redundant
                                            vertexes from top and
                                            bottom

        ## compute faces
        faces = np.empty((rows*cols*2, 3), dtype=np.uint)
        rowtemplate1 = ((np.arange(cols).reshape(cols, 1) + np.
                                            array([[0, 1, 0]])) %
                                            cols) + np.array([[0, 0,
                                            cols]])
        rowtemplate2 = ((np.arange(cols).reshape(cols, 1) + np.
                                            array([[0, 1, 1]])) %
                                            cols) + np.array([[cols,
                                            0, cols]])
        for row in range(rows):
            start = row * cols * 2
            faces[start:start+cols] = rowtemplate1 + row * cols
            faces[start+cols:start+(cols*2)] = rowtemplate2 + row *
                                                cols
        faces = faces[cols:-cols]  ## cut off zero-area triangles
                                            at top and bottom

        ## adjust for redundant vertexes that were removed from top
                                             and bottom
        vmin = cols-1
        faces[faces<vmin] = vmin
        faces -= vmin
        vmax = verts.shape[0]-1
        faces[faces>vmax] = vmax

        # Colors are specified per-vertex
        colors = np.random.random(size=(verts.shape[0], 3, 4))
        #
```

```
        self.Balloon = gl.GLMeshItem(vertexes=verts, faces = faces,
                                     smooth=True,
                           drawEdges=False, vertexColors=colors,
                                                          glOptions
                                                          ='
                                                          opaque
                                                          ')#
                                                          vertex
                                                           ,
                                                          shader
                                                          ='
                                                          balloon
                                                          '
        self.Balloon.translate(0, 0, 0)
        return self.Balloon
        drawballoon_time = timer() - start_drawballoon
        print('drawballoon_time: ', drawballoon_time)

    def ReDrawBalloon(self):
#         print('Redrawballoon angefangen')
        # Selection of input data: frequency x arraygridnr x
                                     measurementaz x
                                     measurementelev.
        #measgrid = Fensterl.meas_grid
        start_redrawballon = timer()
        r_grid = self.meas_grid
        self.frequency = self.Slider_frequency.value()
        self.arraynbr = self.SpinBox_arraynr.value()-1


        [x,y,z] = self.sph2cart(np.squeeze(r_grid[:,0]),np.squeeze(
                                     r_grid[:,1]),np.squeeze(
                                     r_grid[:,2]))


        verts = np.empty(shape = [np.shape(r_grid)[0],3])
#         print(verts.shape)

        verts[...,0] = np.squeeze(x)
        verts[...,1] = np.squeeze(y)
        verts[...,2] = np.squeeze(z)
        self.pos = verts
```

```
#         ## compute hull and faces

        hull = ConvexHull(verts)

        self.verts_new = np.empty(shape = np.shape(self.input[:])+(
                                            3,))

        #
        for ii in range(3):
#             print(ii)
            for i  in range(verts.shape[0]):
                self.verts_new[i,self.arraynbr,:,ii] = verts[i,ii]*
                                                    self.inpnorm[i,
                                                    self.arraynbr,:]#
                                                    *0.05



#         print('verts_new erzeugt')
        self.verts =  self.verts_new[:, self.arraynbr, self.
                                            frequency,:]
        self.phase = np.angle(self.input[:, self.arraynbr, self.
                                            frequency])


        self.face = hull.simplices

        print(self.verts.shape)


        # Colors are specified per-vertex
        self.colorize(self.phase)
        # #
        # colorBar = pg.GradientLegend(size=(50, 200), offset=(15,
                                            -25))
        # colorBar.setGradient(self.cm.getGradient())
        # #labels = dict([("%0.2f" % (v * (2*np.pi), v) for v in np
                                            .linspace(0, 1, 4)])
        # #colorBar.setLabels(labels)
        # self.graphicsView_4.addItem(colorBar)

        self.Balloon.setMeshData(vertexes=self.verts, faces = self.
                                            face, smooth=True,
                            drawEdges=False, vertexColors=self.
```

```
                                                                colors
                                                                ,
                                                                glOptions
                                                                =’
                                                                opaque
                                                                ’) #
                                                                vertex
                                                                 ,
                                                                shader
                                                                =’
                                                                balloon
                                                                ’

        self.Balloon.translate(0, 0, 0)
        self.Balloon.meshDataChanged()
#        print(’Redrawballoon beendet’)
        redrawballoon_time = timer() - start_redrawballon
        print(’redrawballoon_time: ’, redrawballoon_time)
        return self.Balloon



    def balloonrender(self,Fensterl,graphicslot):
        w = graphicslot
        print(’balloonrender durchgeführt’)

        if Fensterl.name == None :

            w.setCameraPosition(distance = 120, elevation = 90,
                                                azimuth = 180)
            g = gl.GLGridItem(size = QtGui.QVector3D(120,120,1))
            g.setSpacing(spacing = QtGui.QVector3D(6,6,1))
            g.scale(1,1,1)
            w.addItem(g)
            self.Balloon =   Fensterl.drawBalloonempty(Fensterl)
#            print(w)
            w.addItem(self.Balloon)



        else:
           self.Balloon = Fensterl.ReDrawBalloon()
           w.addItem(self.Balloon)
```

```
def frequency_changed(self):


    self.arraynbr = self.SpinBox_arraynr.value()-1
    self.frequency = self.Slider_frequency.value()

    self.verts = (self.verts_new[:,self.arraynbr,self.frequency
                                        ,:])
    self.phase = np.angle(self.input[:, self.arraynbr, self.
                                        frequency])

    self.colorize(self.phase)
    self.redrawPolar()

    if self.BALLOON.isVisible():
        self.Balloon.setMeshData(vertexes=self.verts,
                                            vertexColors=self.
                                            colors, faces = self.
                                            face, glOptions='
                                            opaque')
        self.Balloon.translate(0, 0, 0)
        self.Balloon.meshDataChanged()

    if self.graphicsView_3.isVisible():
        self.redrawMap()

    self.label_frequency.setText('Frequenz: '+ str(int(self.
                                        frequs[self.frequency]))+
                                        'Hz')



def sph2cart(self,az, el, r):

    rcos_theta =  np.cos(el*np.pi/180) #r *
    x = rcos_theta * np.cos(az*np.pi/180)
    y = rcos_theta * np.sin(az*np.pi/180) #-
    z = np.sin(el*np.pi/180) #r *
    return x, y, z

def cart2sph(self, xyz):
```

```
        #takes list xyz (single coord)
        x       = xyz[0]
        y       = xyz[1]
        z       = xyz[2]
        r       =  np.sqrt(x*x + y*y + z*z)
        theta   =  np.arccos(z/r)*180/ np.pi #to degrees
        phi     =  np.arctan2(x,y)*180/ np.pi #-
        return [theta,phi,r]




    def colorize(self,phase):

#         color = np.array(phasecolor())
#         color = color.astype(np.ubyte)
#         pos = np.arange(-np.pi, np.pi, 2*np.pi/256)
#
#
#         pos1 = np.array([-np.pi, -np.pi*3/4, -np.pi/2, -np.pi/4, 0
                                  , np.pi/4,\
#                          np.pi/2, np.pi/4 , np.pi])
#         color = np.array([[0,255,255,255], [0, 0, 255, 255], [0,0,
                                0,255], [255, 0, 0, 255],[255,255
                                ,0,255]
#                               , [255, 0, 0, 255], [0,0,0,255] , [0,
                                  0, 255, 255], [0,255,255,255]
                                ], dtype=np.ubyte)

#         pos = np.array([-np.pi, 0,  np.pi])
#         color = np.array([[0,255,0,255], [255,255,0,255], [255,0,0
                                ,0]], dtype=np.ubyte)

#         pos = np.r_[-np.pi, -0.5*np.pi, 0.5*np.pi, np.pi]
#         color = np.array([[0, 0, 1, 0.7], [0, 1, 0, 0.2], [0, 0, 0
                                , 0.8], [1, 0, 0, 1.0]])

#          pos = np.arange(256)
#          color = pg.getLookupTable(start=-np.pi, stop=np.pi, nPts=
                                256, alpha=None, mode='byte')
#         lut=cmocean.cm.phase
```

```
        # pos = np.array([-np.pi, -np.pi*7/8, -np.pi*6/8,-np.pi*5/8
                                        , -np.pi/2, -np.pi*2/8, 0
                                        , np.pi*2/8,\
        #                 np.pi/2, np.pi*5/8 , np.pi*6/8 ,np.pi*7/8
                                         , np.pi])
        # color = np.array([[0,255,255,255],       [0,128,255,255],
                                           [0,128,128,255],
        #                   [0,0,196,255] ,        [0, 0, 255, 255],
        #                   [0, 0, 128, 255] ,     [0,0,0,255],
                                              [128,0,0,255] ,
        #                   [255, 0, 0, 255],      [255, 64, 0, 255
                                        ],  [255, 128, 0, 255],
        #                   [255, 196, 0, 255],    [255,255,0,255]
        #                        ], dtype=np.ubyte)

        #pyqtgraph.GradientEditorItem    ColorMap
        pos1 = np.arange(-np.pi, np.pi, np.pi/129)


        colormap = cm.get_cmap('hsv')#  "nipy_spectral" 'viridis'

        colormap._init()
        lut = (colormap._lut * 255).view(np.ndarray)
        cmap = pg.ColorMap(pos1, lut)
        self.cm = cmap
        #pos, rgba_colors = zip(cmapToColormap(getattr(mcm,
                                        colormap_name)), n_ticks)
        #cmap = pg.ColorMap(pos, color)



        self.colors=cmap.map(phase)
        self.colors.astype('uint8')

    def colorize_polar(self,polarphase):

        pos = np.array([-np.pi, -np.pi*7/8, -np.pi*6/8,-np.pi*5/8,
                                        -np.pi/2, -np.pi*2/8, 0,
                                        np.pi*2/8,\
                        np.pi/2, np.pi*5/8 , np.pi*6/8 ,np.pi*7/8 ,
                                                        np.pi])
        color = np.array([[0,255,255,255],       [0,128,255,255],
                                           [0,128,128,255],
                          [0,0,196,255] ,        [0, 0, 255, 255],
```

```
                        [0, 0, 128, 255] ,    [0,0,0,255],

                                                    [128,0,
                                                    0,255]
                                                    ,
                        [255, 0, 0, 255],     [255, 64, 0, 255],
                                                     [255,
                                                    128, 0,
                                                     255],
                        [255, 196, 0, 255],   [255,255,0,255]
                             ], dtype=np.ubyte)


        cmap = pg.ColorMap(pos, color)
        self.polarcolor = cmap.map(polarphase)
        self.polarcolor.astype('uint8')

    def export(self):
        filename, fileext  =  os.path.splitext(self.name)

        if self.graphicsView_3.isVisible():
            port = self.graphicsView_3
            savestr = filename +'_map_' + str(int(self.frequs[self.
                                            frequency]))+'Hz.png'
            savename = QtGui.QFileDialog.getSaveFileName(self, '
                                            Save as',savestr)# ,
                                            selectedFilter='*.xml
                                            '
            svn = os.fsencode(savename[0])
            with open(svn, "wb") as f:
                ret =  self.fig.savefig(f)

        elif self.BALLOON.isVisible():
            port = self.graphicsView_4
            savestr = filename +'_bal_' + str(int(self.frequs[self.
                                            frequency]))+'Hz.png'
            savename = QtGui.QFileDialog.getSaveFileName(self, '
                                            Save as',savestr)# ,
                                            selectedFilter='*.xml
                                            '
            ret = port.grabFrameBuffer().save(savename[0])

        return ret
```

```
    def retranslateUi(self, Fensterl):
        _translate = QtCore.QCoreApplication.translate
        Fensterl.setWindowTitle(_translate("Fensterl", "3D
                                           Directivity GUI"))
        self.label.setText(_translate("Fensterl", "No data loaded
                                           yet!"))
        self.tabWidget.setTabText(self.tabWidget.indexOf(self.
                                           BALLOON), _translate("
                                           Fensterl", "Balloon
                                           diagramm"))
        self.tabWidget.setTabText(self.tabWidget.indexOf(self.
                                           graphicsView_3),
                                           _translate("Fensterl", "
                                           Map"))
        self.menuMenu.setTitle(_translate("Fensterl", "Menu"))
        self.actionOpen.setText(_translate("Fensterl", "Open"))
        self.actionQuit.setText(_translate("Fensterl", "Quit"))
        self.name = None
        self.pushButton.setText(_translate("Fensterl", "Save graph"
                                           ))
        self.label_frequency.setText(_translate("Fensterl", "
                                           Frequenz:"))
        start = timer()
        print('Anfang:', start)



# main loop

if __name__ == "__main__":

    app = 0

    if not QtWidgets.QApplication.instance():

        app = QtWidgets.QApplication(sys.argv)

    else:
```

```
        app = QtWidgets.QApplication.instance()


    app.aboutToQuit.connect(app.deleteLater)
    Fensterl = Ui_Fensterl()
    Fensterl.setupUi(Fensterl)

#  Plotting and rendering

    slot4 = Fensterl.graphicsView_4
    slot3 = Fensterl.graphicsView_3
    Fensterl.balloonrender(Fensterl,slot4)

# Execute

    Fensterl.show()
    Fensterl.raise_()
    app.exec_()
```