POLITECNICO DI MILANO
Scuola di Ingegneria dell'Informazione

POLO TERRITORIALE DI COMO

Master of Science in Computer Engineering

An Ambisonics-based VST plug-in for 3D music production.

Supervisor:
Prof. Augusto Sarti

Assistant Supervisor:
Dott. Salvo Daniele Valente

Master Graduation Thesis by:
Daniele Magliozzi
Student Id. number 739404

Academic Year 2010-2011

POLITECNICO DI MILANO

Scuola di Ingegneria dell'Informazione

POLO TERRITORIALE DI COMO

Corso di Laurea Specialistica in Ingegneria Informatica

Plug-in VST per la produzione musicale 3D basato sulla tecnologia Ambisonics.

Relatore:
Prof. Augusto Sarti

Correlatore:
Dott. Salvo Daniele Valente

Tesi di laurea di:
Daniele Magliozzi
Matr. 739404

Anno Accademico 2010-2011

To me

Abstract

The importance of sound in virtual reality and multimedia systems has brought to the definition of today's *3DA* (*Tridimentional Audio*) techniques allowing the creation of an immersive virtual sound scene. This is possible virtually placing audio sources everywhere in the space around a listening point and reproducing the sound-filed they generate by means of suitable DSP methodologies and a system of two or more loudspeakers.

The latter configuration defines multichannel reproduction techniques, among which, *Ambisonics Surround Sound* exploits the concept of spherical harmonics sound 'sampling'.

The intent of this thesis has been to develop a software tool for music production able to manage more source signals to virtually place them everywhere in a (3D) space surrounding a listening point employing Ambisonics technique. The developed tool, called *AmbiSound-Spazializer* belong to the plugin software category, i.e. an expantion of already existing software that play the role of host.

Sommario

L'aspetto sonoro, nella simulazione di realtà virtuali e nei sistemi multimediali, ricopre un ruolo fondamentale. Ciò ha portato alla definizione delle moderne tecniche di renderizzazione dell'audio tridimensionale (*3DA*), attraverso le quali è possibile creare scene sonore realistiche posizionando sorgenti virtuali nello spazio circostante il punto di ascolto. Il campo sonoro da esse generato è riprodotto utilizzando appropriate tecnologie di DSP e sistemi di riproduzione formati da due o più altoparlanti.

Questi ultimi rientrano nella categoria dei sistemi multicanale e impiegano specifiche tecniche di riproduzione e registrazione, tra cui *Ambisonics Surround Sound* che sfrutta il concetto campionamento spaziale del campo suono in armoniche sferiche.

Scopo di questa tesi è stato lo sviluppo di uno strumento software per la produzione musicale che impiega la tecnica di spazializzazione Ambisonics per gestire il posizionamento di più sorgenti sonore in uno spazio tridimensionale.

Il software, denominato *AmbiSound-Spazializer* appartiene alla classe dei plugins. Questi ultimi espandono le funzionalità di software, detti 'host', che li ospitano e ne gestiscono il flusso dati.

Acknowledgements

I would like to thank :

my father for financing me; my mother and Emanuela for their support; Maestro Giovanni Cospito, Electronic Music teacher of 'Conservatorio G. Verdi di Milano', for teaching me understanding sound; Prof. Augusto Sarti for making me discover the "cool side" of engineering; Dott. Salvo Daniele Valente for guiding and aiding me during the whole project work ; Carolina Arias and Luca Terzi for the photos taken at the sound laboratory 'Laboratorio di Elaborazione e produzione dei segnali Audio e Musicali' of 'Politecnico di Milano' in Como; all the people that have contributed to the software testing phase.

Contents

Abstract **I**

Sommario **III**

Acknowledgements **V**

1 Introduction **1**
1.1 Context overview: tridimentional audio (3DA) 1
1.2 Work brief description 3
1.3 Thesis structure 5

2 State of the Art **7**
2.1 State of the Art of Ambisonics realizations 7
2.2 Hardware Releases 10
2.3 Software Releases 12
2.3.1 Standalone software tools 12
2.3.2 Ambisonics VST plugins 13

3 Research problem formulation **17**
3.1 Traditional audio rendering techniques 17
3.1.1 Stereophonic Sound 17
3.1.2 Quadraphonic Sound 18
3.1.3 Dolby Surround and Home Theater 19
3.1.4 Dolby Digital, 5.1 and 7.1 20
3.2 DAW employment in studio mixing 22
3.3 Project aims 25

4 Techniques and Technologies **27**
4.1 Ambisonics Surround Sound 27
4.1.1 Acoustic waves remarks 29

4.1.2 Ambisonics theory 32
4.1.3 Near field effects 42
4.2 Virtual Studio Technology 51
4.2.1 VST plugins overview 51
4.2.2 Introduction to VST *Software Development Kit* 53
4.3 Reaper DAW 54
4.3.1 Reaper main features and layouts 55
4.3.2 Additional features compared with other DAWs 56
4.3.3 Reaper relevant features for *AmbiSound-Spatializer* project 59

5 System Architecture 67
5.1 *AmbiSound-Spatializer*: top level view 68
5.2 Plugin architecture overview 69
5.3 Plugin process architecture 72

6 Testing and evaluation 87
6.1 AmbiSound-Spatializer testing 87
6.1.1 System overview 87
6.1.2 Tests realization 87
6.1.3 Test results 91
6.2 Project evaluation 92

7 Next research directions and conclusions 95

A Listing 99

Bibliography 133

Chapter 1

Introduction

1.1 Context overview: tridimentional audio (3DA)

Sound is becoming everyday more important in multimedia systems and virtual reality. In fact, throughout tridimentional audio (3DA) or 'auralization' techniques, it is possible to create a convincing and efficient model of a tridimentional sound event. 3DA aims to offer to the listener a more suggestive and realistic fruition for a given acoustic information. There are many applications for these technologies: video games, video conference systems, electrical musical instruments, entertainment (digital cinema, etc.), Virtual Display systems, multichannel audio systems (films on DVD and Virtual Home Teather (VHT)), etc.

The 3DA term defines a set of methodologies and techniques that allow the creation of a virtual "sound scene" or rather everything involved in the sound scene: the listening environment and the listener, integrated and modelled using DSP technologies.

3DA techniques allows, especially, to virtually place one or more sound sources everywhere in the space around a listening point. Hence, by mean of the suitable DSP methodologies, with 3DA is possible to reproduce every kind of sound field: static or dynamic, natural or artificial ones by using two or more loudspeakers.

The reproduction methods can be divided into three principal categories:

- multichannel reproduction;
- binaural reproduction with two loudspeakers;

- binaural reproduction with headphones.

In the last two techniques only two physical sources are used to diffuse sound, hence to recreate the tridimentional sound space it is necessary to rely upon a listener model (human ears, head and body - trunk). Through this model, created by means of numerical filtering techniques, it is possible to 'trick' the ear and place the virtual source everywhere around the listener.

Obviously in multichannel systems, where sources are (ideally) physically distributed in each point of the surrounding space, the listener model is no more necessary. Thus, from this point of view, using these systems is easier to reach good results in sound field reproduction by little efforts for each loudspeaker. On the other hand the whole systems are bigger than binaural ones, hence they are more expensive, difficult to manage and require more space to be setted. Moreover comparing binaural reproduction with headphones and multichannel reproduction system it's remarkable that the first doesn't suffer of 'sweet spot' problems despite the second one where the listener is bounded into a restricted area (that can be enlarged in through appropriate design specifications) inside the loudspeakers arrangement. For these reasons multichannel systems have been mainly employed for sound reproduction in big spaces till now (cinemas, theatres, concert halls, etc.).

A further classification can be done in multichannel reproduction systems between:

- systems that can be conceived as binaural systems expansions,

  as *linear arrays of loudspeaker* and *Vector Based Amplitude Panning* (*VBAP*), that relies on a limited number of array and take advantage on psychoacoustics laws;

- systems that aims to reproduce the acoustic field with a particular loudspeakers disposition,

  as *Holophonic systems* and *Ambisonics systems* that generally relies on a large number of loudspeakers placed in a specific manner (for example in circle or on a sphere surface) and employ the concept of sound 'sampling' spherical harmonics and space domain.

The intent of this thesis has been to develop a software tool able to manage more source signals to virtually place them everywhere in a

(3D) space surrounding a listening point employing Ambisonics technique. The tool is an expantion of already existing software employed in signal treatment, studio and home recording, sound editing, post-processing and production. It takes the name of *plugin*: a set of software components that adds specific abilities to a larger software application often called ‘host’ application.

The whole software, named **AmbiSound-Spatializer**, has been developed in C++ employing the *Virtual Studio Technology* (*VST*) proposed by the well know German software house *Steinberg*. Compared to existing VST plugins and other software developed during last years, the work has been manly focused on the following purposes of innovation:

- extension of adopted Ambisonics techniques to *Higher Order Ambisonics* (*HOA*), till 3$^{\text{rd}}$ order, in VST plugins;
- extension of the reproduction system supported by the VST plugin from a 2D speakers array to a 3D disposition, hence possibility to virtually position sources in a tridimentional space;
- many input feeds supported by the same plugin (max. 32 inputs) to diminish the CPU resources usage granting a good real-time feedback, allowing the plugin to be also employed for live executions;
- implementation of *Near Field Compensation* (*NFC*) filters, giving the possibility to use the plugin relying on small sized reproduction systems (with loudspeakers at least 1 meter distant from the listener) adopted in little recording studios.

The obtained VST plugin can be considered as a starting platform for further developments of more Higher Order Ambisonics plugins, relying on more complex reproduction systems and to be connected with plugins for the emulations of virtual environments to give the listener also the perception of the interaction with the location where the sound is placed in.

1.2 Work brief description

The developed software finds its theoretical roots mainly in Jérôme Daniel’s doctoral thesis at Paris 6 University [2]. Some chapters of this document are dedicated to the description and definition of Ambisonics theory and techniques [2, c. 2, 3 and a. A] and, in particular, Daniel clearly defines the actions of encoding and decoding of Ambisonics signals, from a mathematical point of view, for both 2D and 3D reproduction systems extending these definitions to a generic encoding order M. Both encoding and decoding operations are written in a compact form using matrices and vectors to represent discrete source signals and spherical harmonics sampling of input and output signals. In addition Daniel deepens how the loudspeakers arrangement affects the goodness of the reproduction system and how it is possible to reach more simple encoding/decoding operation forms on special arrangement conditions defining form, loudspeaker number and disposition for the latter.

All these arguments have been the starting point for the work faced in this thesis directed toward the realization of a software tool able to encode signals in HOA (till the 3° order) and decode them in a tridimentional reproduction system of regular form.[1]

Jérôme Daniel wrote also many articles exposing his research work on HOA further investigations during the past years. One of these ([6]) deals with near field aliasing introduced by realistic sound fields that generates spherical waves instead of plane waves, which the Ambisonics theory is based on. The study of a compensation technique for this type of disturb has brought Daniel to the definition of *Near Field Compensation* filters to be applied on Ambisonics signals and also to the description of a rigorous method to be followed to design the digital version of such filters.

The above mentioned method has been employed for the development of the process part that realize the NFC filtering often neglected in previous realizations of Ambisonics software tools. Ambisonics technique is based on far-field assumption so this improvement has given the possibility for the plugin to be used also in small recording studios contexts without any lost of efficiency for the applied Ambisonics spatialization technique.

As already said the source code implements a non-standalone software that has to be recalled inside a host application to run its processes. The type of host applications used for sound manipulation, processing, production, post-production and play-back take the name of *Digital Audio Workstations (DAW)*. *AmbiSound-Spatializer* can man-

[1] What 'regular form' reproduction system is intend for will be explained in Chapter 4.

age up to 32 input audio tracks and 36 output audio streams. The latter signals are used to feed the reproduction system: up to 36 loudspeakers arranged in hemispherical position.

The host application also provide a graphic user interface (GUI) for those variables defined inside the code as adjustable parameters. *AmbiSound-Spatializer* has been designed to give the user the possibility of position each input source acting on azimuth, elevation and distance controls. Moreover one can also set the distance of the reproduction system from the origin (listening point) and enable or disable near field compensation.

The development work has been focused also on the research of a computational strategy to diminish the CPU usage that grows faster than linearly increasing the number of sources to be processed. A decrease of 30% of the CPU usage has been reached for one source and a linear growth when increasing the number of inputs.

Due to this improvement the plugin doesn't require the employment of an high-performance machine to run it and it can be used also in live reproductions.

1.3 Thesis structure

This document is structured as follows:

Chapter 2 exposes the state of art inherent in Ambisonics technique, in tridimentional rendering technologies and researches by listing and briefly describing all bibliographical sources used to acquire the proper knowledge about this topic and useful for the project developing. Moreover many of the actual software projects regarding Ambisonics are shown especially focusing on the history of VST plugin realized for Ambisonics and their features.

Chapter 3 shows the whole panorama of today's traditional spatialization technologies by briefly describing each one and introducing the possibility of a 3D mixing and of Ambisonics technique employment. Then it gives an overview on today's DAWs and how they are used to realize modern mixing methodologies. The last part of the chapter reports the reasons for the implementation technology chosen, telling what is VST justifying the employment of this standard for the plugin development deepening the relations with other DAWs supported plugin technology.

In *Chapter 4* the various technologies involved in the project realization are explained in more detail dividing all content into three principal sections:

- *Section 1* deepens *Ambisonics Surround Sound, HOA* and *Field Compensation* theory and fully describes how the problem of spherical harmonic sampling, aliasing compensation and reproduction with a loudspeaker system is solved by a mathematical point of view.
- *Section 2* expands the knowledge of *VST technology* explaining how a VST plugin is structured from a logical point of view and what are the actions carried out by the main plugin source code parts.
- *Section 3* illustrates the DAW (*Reaper*) chosen for the project justifying the specific choice.

Chapter 5 shows the system architecture expanding the various modules, in particular focusing on two modules: the encoding/decoding and the NFC filtering blocks.

The last chapter, *Chapter 6*, takes conclusions from the work done on this project telling about the research future perspectives in this area, making some evaluations on what as been developed and giving hints on what shall be the adjustments and addictions to be released by the software next versions.

Then follow three appendixes:

Appendix A presents the whole source code developed to implement *AmbiSound-Spatializer* fully commented.

Appendix B illustrates a user manual aiming to clarify to a possible user how to set the DAW ambient to correctly use AmbiSound, to handle all plugin parameters to obtain the reproduction of a desired virtual sound environment and how it is possible to interact with the latter using the DAW capabilities (for example to implement automations).

Appendix C gives and idea of what can be a possible application of this software describing as a specific context of usage the audio editing for cinema sound effects realization, showing an example of a 3D sound scene render done by using AmbiSound plugin.

Chapter 2

State of the Art

2.1 State of the Art of Ambisonics realizations

Ambisonics (not to be confused with ambiophonics) is a series of recording and replay techniques using multichannel mixing technology that can be used live or in the studio. By encoding and decoding sound information on an arbitrary number of channels, a 2-dimensional ("planar", or horizontal-only) or 3-dimensional ("periphonic", or full-sphere) sound field can be presented. Ambisonics was invented by *Michael Gerzon* of the Mathematical Institute, Oxford, who – with Professor Peter Fellgett of the University of Reading, David Brown, John Wright and John Hayes of the now defunct IMF Electronics, and building on the work of other researchers – developed the theoretical and practical aspects of the system in the early 1970s. [19]

In the basic version, known as first-order Ambisonics, sound information is encoded into four channels: W, X, Y and Z. This is called Ambisonic B-format. The W channel is the non-directional mono component of the signal, corresponding to the output of an omnidirectional microphone. The X, Y and Z channels are the directional components in three dimensions. They correspond to the outputs of three figure-of-eight microphones, facing forward, to the left, and upward respectively. The B-format signals are based on a spherical harmonic decomposition of the sound field and correspond to the sound pressure (W), and the three components of the pressure gradient (X, Y, and Z) at a point in space. Together, these approximate the sound field on a sphere around the microphone; formally the first-order truncation of the multipole expansion. This first-order truncation is only an approximation of the overall sound field. Loudspeakers signals are derived by using a linear combination of these four channels, where each signal is dependent on the actual position of the speaker in relation to the center of an imaginary sphere the surface of which passes through all available speakers. [19]

Up to now many progresses have been made in Ambisonics research that have brought to the definition of some standards encoding and decoding formats based on modern reproduction systems and existing transducers designed for signal recording and directional sampling.

Today's traducers employed in Ambisonics are able to sample directional components of order zero (pressure microphones) and first order (gradient pressure microphones). Higher orders correspond to an acoustic sound field sampling done through microphones with higher directivity and require a larger channels number (at least 9). [1]

Once derived from transducers, the signals can be encoded in many ways. In fact different format have been developed:

- A-format. Used with directional microphones placed in a single point and symmetrically oriented.
- B-format. Developed for studio equipments (most common) and when the available microphones have directional features similar to those in figure 2.1.
- C-format for transmission.
- D-format for decoding and reproduction.
- UHJ-format employed to make mono and stereo reproduction systems compatible with multichannel encoding.

[1]

A particularly active area of current research is the development of *"higher orders" of Ambisonics.* For about eight years , the Ambisonics extension to higher spatial resolution systems has been the object of increasingly numerous studies which promising features are becoming practicable. [4]

The main results obtained during these years can be summed up in new encoding and decoding equations and the extension of the supported reproduction system to a tridimentional one exploiting Ambisonics encoding higher than first order.

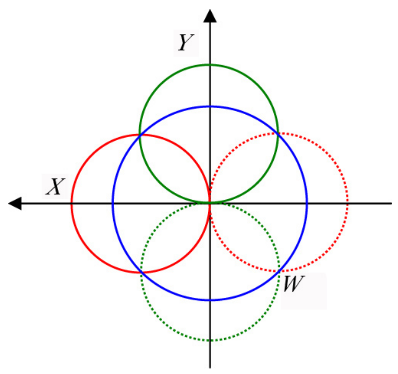


Figure 2.1: B-format 2D Polar diagram (z axis omitted).

Using more channels than the original first-order B-Format allows to capture significantly more spatial information. At present,“real” recording techniques using HOA are in their infancy, it is, however, straightforward to compose synthetic recordings. Benefits include greater localisation accuracy and better performance in large-scale replay environments such as performance spaces. The higher orders correspond to further terms of the multipole expansion of a function on the sphere in terms of spherical harmonics. In absence of obstacles, sound in a space over time can be described as the pressure at a plane or over a sphere – and thus if one reproduces this function, one can reproduce the sound of a microphone at any point in the space pointing in any direction. [19]

Encoding/decoding strategies developed up to now for HOA can be divided into the following two categories.

The one described by Jérôme Daniel in [2] is an extension of Gerzon’s first order formalism and uses the 3D pressure wave equation solution derived in [7, ch. 10] to define the signal spherical harmonic decomposition and an algorithm to compute encoding coefficients recursively (as illustrated in [6]). Daniel in [2] has defined the above mentioned coefficients as functions of position parameters naming them *“Ambisonics Encoding Functions”* and computing them for both Cartesian and spherical coordinate systems using different notations. Then he has also introduced the corresponding *Furse-Malham* weights, already defined by Dave Malham in [13], as normalization terms for each encoding function, used in practical systems.[1] Others important results

[1]Both encoding and decoding theory for HOA treated by Daniel in [2] and [4] will be deepen in detail in chapter 4 first section .

discussed in [2] are the simplified encoding\decoding forms for reproduction systems regular arrangements (spheres, hemispheres, regular polyhedra, etc.) and the algebraic relation with the pan-pot 3D spatialization technique.

A different approach is introduced in [10] by Rabenstein and Spors. Starting from the solution of wave equation derived in [7, ch. 1] extended for 3D pressure waves and moved in frequency domain they have defined the “angular modes” wave decomposition done by *“Fourier series coefficients”* or *“angular coefficients”*. Hence Ambisonics signals have been described as the inverse Fourier Transform of the latter.

However this strategy doesn’t seem to have found wide employment on 3D Ambisonics applications, thus the way followed during the project development stars mainly from Daniel research work.

Another interesting topic developed in recent studies by Jérôme Daniel modelling of real sound-fields. Previous literature only rarely addressed the modelling of spherical waves, radiated by finite distance sources. Nevertheless, correct encoding and reconstruction of realistic sound fields require it, and couldn’t satisfy themselves with the usual plane wave approximation. [4] This has brought Daniel to a complete definition of near field compensating filters theory showing in [6] how to obtain the impulse responses in digital domain and giving also some computed coefficients up to 6^{th} order.

2.2 Hardware Releases

Many Ambisonic recordings have been made using a special microphone called *Sound-Field Microphone (SFM)*. This kind of microphone can be reconfigured electronically or via software to provide different stereo and 3D polar response either during or after recording.

Other releases are Ambisonic pan-pots that, with differing degrees of sophistication, provide the fundamental additional studio tool required to create an Ambisonic mix, by making it possible to localise individual, conventionally-recorded multi-track or multi-microphones sources around a 360° stage analogous to the way conventional stereo pan-pots localise sounds across a front stage. However, unlike stereo pan-pots, which traditionally vary only the level between two channels, Ambisonic panning provides additional cues which eliminate conventional localisation accuracy problems. This is especially pertinent to surround, where our ability to localise level-only panned sources is severely limited to the sides and rear. [19]

By the early 1980s, studio hardware were realized for the creation of multi track-sourced, Ambisonically-mixed content, including the ability to incorporate SFM-derived sources (for example for room ambience) into a multichannel mix. This was thanks primarily to the efforts of Dr Geoffrey Barton (now of Trifield Productions) and the pro-audio manufacturers Audio & Design Recording, UK (now Audio & Design Reading Ltd). Barton designed a suite of outboard rack-mounted studio units that became known as the Ambisonic Mastering System. These units were patched into a conventional mixing console and allowed conventional multi-track recordings to be mixed Ambisonically. The system consisted of four units:

- Pan-Rotate Unit – This enabled eight mono signals to be panned in B-format, including 360° "angle" control and a "radius vector" control allowing the source to be brought in towards the centre, plus a control to rotate an external or internal B-format signal.

- B-Format Converter – This connected to four groups and an aux send and allowed existing console pan-pots to pan across a B-Format quadrant.

- UHJ Transcoder – This both encoded B-Format into 2-channel UHJ and in addition allowed a stereo front stage and a stereo rear stage (both with adjustable widths) to be transcoded direct to 2-channel UHJ.

- Ambisonic Decoder – this accepted both horizontal (WXY) B-format and 2-channel UHJ and decoded it to four speaker feeds with configurable array geometry.

Versions of these units were subsequently made available in the late 1990s by Cepiar Ltd along with some other Ambisonics hardware. It is not known if they are still currently available. [19]

The lack of availability of 4-track mastering equipment led to a tendency (now regretted by some of the people involved) to mix directly to 2-channel UHJ rather than recording B-format and then converting it to UHJ for release. The fact that you could mix direct to 2-channel UHJ with nothing more than the transcoder made this even more tempting. As a result there is a lack of legacy Ambisonically-mixed B-format recordings that could be released today in more advanced formats (such as G-Format). However, the remastering – and in some cases release – of original 2-channel UHJ recordings in G-Format has proved to be surprisingly effective, yielding results at least as good as the original studio playbacks, thanks primarily to the significantly higher quality of current decoding systems (such as file-based software decoders) compared to those available when the recordings were made. [19]

2.3 Software Releases

The advent of digital audio workstations has led to the development of both encoding and decoding tools for Ambisonic production. Many of these have been developed under the auspices of the University of York [12]. Most of them have been created using the VST plugin standard developed by Steinberg and used widely in a number of commercial and other software-based audio production systems, notably Steinberg's Nuendo. With the lack of necessity to interface to a conventional console, the encoding tools have primarily taken the form of B-Format pan-pots and associated controls. Decoder plugins are available for monitoring.

2.3.1 Standalone software tools

Many software implementing 3D audio rendering techniques and Ambisonics has been realized up to know. The following lines briefly describe the most used and mentioned ones.

A& G Soluzioni Digitali (Italy) born in 1995 patent in 1998 the *3D Enhanced Surround Technology* based on concept exposed in the book "Creating Soundscapes" (2007). It has developed under this technology an hardware platform *X-spat boX2* and a control software *See'nSound* used for multichannel signals editing and reproduction. The choice of a dedicate hardware could be very binding and expensive (an X-pat player costs about 990 Euro much more then a common eight channel audio board) on the other side the compatibility with formats stereo, quad and 5.1 is noteworthy and Ambisonics can be rencoded in each standard using the same plugin.

Another tool has been realized by Thomas Chen (California) [18]. B-Format Mixer is a standalone software released by that emulates an eight mono or stereo channels mixer with traditional features (as equalization, compression, etc.) and permits sources spatialization through Ambisonics Surround Sound technique but with a static approach for sources number and encoding order setting.

A set of Max\MSP (www.cycling74.com) externals tool has been released that implement Ambisonics Surround Sound by Graham Wakefield [22]. The suite include Max patches for encoding, rotating and decoding up to 3rd order for two or three dimensional speaker arrays. It encodes up to 16 sources to distinct azimuth and elevation orientations, balances the components of an Ambisonic encoded sound-field per order, using a set of pre-defined or user-defined weights and decodes an Ambisonics encoded sound field to a user-defined speaker array of up to 16 channels.

A similar set of tools [11] has been released by Jan C. Schacher and Philippe Kocher including patches for encoding and decoding Ambisonics signals. To give the user an intuitive access to positions of the source-sounds the GUI-object "ambimonitor" was developed. It integrates seamlessly with the encoder, transmitting the correctly formatted indexed lists containing the position information for each point. The graphic display is used to visualize positions on a two-dimensional surface or with half (or full) sphere three-dimensional display using two views (top and front). Sources are displayed as dots and are labelled either with symbolic names or their indices.

2.3.2 Ambisonics VST plugins

Daniel Courville (France) has developed *Ambisonic Studio's B2X* and OBO-RO [17]. The former is an Ambisonics audio production complete and free-ware VST/AU suite compatible with the main AU workstations and VST hosts as Logic, Cubase and Nuendo. The encoding\decoding order reaches two for the periphonic reproduction and five for the planar one but requires one plugin instance for each source to be spatialized. OBORO permits only an encoding\decoding of order three for planar reproduction.

Dave Malham and Ambrose Field from the University of York Department of Music have implemented (2000-2001) several VST plugin during last years available for download in the web page [14], described as follows:

BPan. An Ambisonic encoder that comes in four basic forms, any or all of which can be installed as VST plugins in a compatible host such as Steinberg's Nuendo, Audiomulch, Plogue Bidule, Tracktion and so on. There are two variants without a custom Graphical User Interface (GUI) of their own, relying on the host's own interface and two variants with their own GUI. The GUI-less and the GUI-based forms are each configured as either ordinary send effect plugins or as spatializer plugin. The reason for all these different variants is that each one has its own advantages for different ways of working. The GUI-less versions offer immediate access to all the parameters, using the host's own interface (usually sliders) . This gives much more flexibility (at the cost of ease of control) than the GUI-based versions which, on the other hand, present the more frequently used parameters in an easier to use form with the other parameters available only through setting automation." [14]

The B format signals generated by these plugins are routed to the standard Nuendo output channels as follows, again, assuming that you have a four (or more) channel sound card. [14]

B-dec. A 64-bit double precision Ambisonics decoder optimised for sound quality and large listening spaces. The decoding method reflects this means by not processing the signal inappropriately: there are no shelf filters in this design. Instead, the decoding method is a clean signal path, in-phase decode. [14] Up to eight output layouts are available. B-dec is designed to be used as part of the master effects section. It will install itself as a 'surround effect' in Nuendo. Make sure you have sufficient hardware output channels available to run the process, and that you have set-up and activated in "Master Outputs" sufficient output channels. B-dec does not worry about the speaker layout specified in Steinberg's box, as this does not currently use the height (Z) information." [14]

B-Proc. Allows you to rotate, tilt and tumble a complete first order B-Format sound-field. The order that these operations occur in is important, since it affects the final orientation of the sound-field, as all operations are with respect to the central listening position, not the sound-field – so, if you rotate by 30 degrees to the left, sounds that were 30 degrees to the right are now at the front, if you then tumble the field 20 degrees down, the sounds which were, after the rotate, due front but 20 degrees up, move 20 degrees down to the horizontal and then a tilt by 40 degrees to the left makes any sounds that were (after the rotate and tumble) due left and horizontal, go to due left but 40 degrees down. This will produce a different result from any other sequence of the operations. [14]

B-Zoom. Allows you to zoom in on, or away from, a point on a complete first order B-Format sound-field. It has three controls, a pair of direction controls, Azimuth and Elevation, which allow you to set the point to which you will zoom (Azimuth in degrees anticlockwise from due front, elevation in degree from horizontal (90 degrees is straight up)) and a zoom control. This varies from no effect at the centre point, to zoomed fully towards the zoom point at +1 and zoomed fully towards the opposite direction at -1.0. The effect of zooming in is to make sounds in that direction louder (and those in the opposite direction quieter) and also to spread the angles between them (and reduce the angular spread of sounds in the opposite direction). [14]

B-Plane Mirror. Allows you to distort a B-Format sound-field. It has three controls, a pair of direction controls, Azimuth and Elevation, which allow you to define the axis along which the mirroring takes place (Azimuth in degrees anticlockwise from due front, elevation in degree from horizontal (90 degrees is straight up)) and a mirror control. This varies from collapsing the field onto the plane perpendicular to the axis when it is at the centre point, to normal field presentation when fully towards the 'Normal' end point at +1 and mirrored fully towards the opposite direction at -1.0, the 'Mirror' end point. [14]

B-Mic. Allows you to generate a virtual coincident stereo pair of microphones from a B-Format sound-field, either recorded or synthesized. The plugin can also be used as the basis of a fully programmable decoder matrix. You can do this by using multiple copies of the plugin, each generating the feeds for two (usually diametrically opposed) speakers. " [14]

One big issue for all these plugins is that the user has to control his

B-format signal is in W, X, Y, Z (1,2,3,4) format. Currently Nuendo (and also other DAWs) does not offer a multichannel file input option so any B Format recordings have to be provided as two stereo files feeding separate input channels. Thus Nuendo has to be configured in such a way that these two channels feed the Decoder on these busses, then BPan connects itself correctly automatically. [14]

Chapter 3

Research problem formulation

3.1 Traditional audio rendering techniques

To understand the application of a 3D mixing tool in audio production studios is useful to illustrate today's most used spatialization techniques and their employment contexts. The following paragraphs retrace the development history of the above mentioned rendering techniques during last years.

3.1.1 Stereophonic Sound

The term *Stereophonic Sound*, commonly called *Stereo*, refers to any method of sound reproduction in which an attempt is made to create an illusion of directionality and audible perspective. This is usually achieved by using two or more independent audio channels through a configuration of two or more loudspeakers in such a way as to create the impression of sound heard from various directions, as in natural hearing. Thus the term "stereophonic" applies to so-called "quadraphonic" and "surround-sound" systems as well as the more common 2-channel, 2-speaker systems. Traditionally it is referred has a synonymous of a reproduction system with two loudspeaker and it is often contrasted with monophonic, or "mono" sound, where audio is in the form of one channel, often centred in the sound field. Stereo sound is now common in entertainment systems such as broadcast radio and TV, recorded music and the cinema.

Stereophonic sound attempts to create an illusion of location for

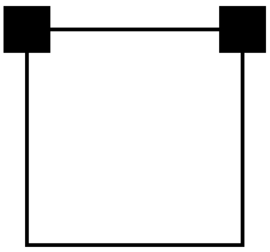


Figure 3.1: Label for 2.0 sound (stereo).

various sound sources (voices, instruments, etc.) within the original recording. The recording engineer's goal is usually to create a stereo "image" with localization information. When a stereophonic recording is heard through loudspeaker systems (rather than headphones), each ear, of course, hears sound from both speakers. The audio engineer may, and often does, use more than two microphones (sometimes many more) and may mix them down to two tracks in ways that exaggerate the separation of the instruments, in order to compensate for the mixture that occurs when listening via speakers. In addiction, especially in synthetic music editing, mixing and live reproduction, some techniques defined in psychoacoustics are used to obtain the illusion of placing sound sources between the two loudspeakers. This techniques are based on the effects perceived by human hear when applying small differences to signal amplitudes (*interaural amplitude difference* or *IAD*) or reproduction time delays (*interaural time difference* or *ITD*).

Reproduction systems with a stereophonic arrangement for commercial use have been very common till early 1990s but now they are disappearing, being replaced by modern surround sound formats.

3.1.2 Quadraphonic Sound

Quadraphonic (or*Quadrophonic* and sometimes *Quadrasonic*) sound – the most-widely-used early term for what is now called *4.0 surround sound* – uses four channels in which speakers are positioned at the four corners of the listening space, reproducing signals that are (wholly or in part) independent of one another. Quadraphonic audio was the earliest consumer offering in surround sound. It was a commercial failure due to many technical problems and format incompatibilities. Quadraphonic audio formats were more expensive to produce than standard two-channel stereo. Playback required additional speakers and specially designed decoders and amplifiers. The rise of home theatre products in the late 1980s and early 1990s brought multi-channel audio recording back to into popularity, although in new digitally based formats. Thousands of quadraphonic recordings were made during the 1970s, and some of these recordings have been reissued in modern surround sound formats such as DTS, Dolby Digital, DVD-Audio and Super Audio CD.

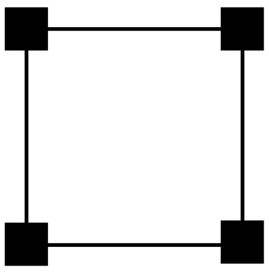


Figure 3.2: Label for 4.0 sound (quadraphonic).

3.1.3 Dolby Surround and Home Theater

Dolby Surround was the earliest consumer version of Dolby's multichannel analogue film sound decoding format Dolby Stereo introduced to the public in 1982 during the time home video recording formats (such as Betamax and VHS) were earlier introducing Stereo and Hi-Fi capability. Dolby Surround is the earliest domestic version of theatrical Dolby Stereo and has contributed to the birth of *Home Theater* conception, that tries to reproduce a real "cinema experience" (or theatre) at a private home by using a particular disposition of more than two loudspeaker giving the listener the sensation of being immersed in the middle of the sound scene. The term Dolby Surround also applies to the encoding of material in this sound format.

When a Dolby Stereo/Dolby Surround soundtrack is produced, four channels of audio information – left, center, right, and mono surround – are matrix-encoded onto two audio tracks. The stereo information is then carried on stereo sources such as videotapes, laser-discs and television broadcasts from which the surround information can be decoded by a processor to recreate the original four-channel surround sound. Without the decoder, the information still plays in standard stereo or mono.

Dolby Surround Matrix	**Left**	**Right**	**Center**	**Surround**
Left Total (Lt)	1	0	$\frac{\sqrt{2}}{2}$	$j\frac{\sqrt{2}}{2}$
Right Total (Rt)	0	1	$\frac{\sqrt{2}}{2}$	$-j\frac{\sqrt{2}}{2}$

Table 3.1: Dolby Surround matrix. Note that j represents a 90° phase shift.

As the technology of a Dolby Surround decoder is virtually the same as decoding the monaural surround soundtrack, many Dolby Stereo encoded films could be transferred with little change to the stereo soundtrack, lowering the costs of re-recording the audio of a film to video. In fact, most L/R/S Dolby Surround decoders included a modified Dolby B decoder as part of their design. The Dolby Surround decoding technology was updated during the mid-1980s and re-named Dolby Pro Logic in 1987. The terms Dolby Stereo, Dolby Surround and LtRt are used to describe soundtracks that are matrix-encoded using this technique.

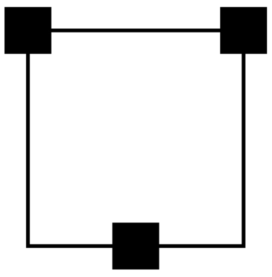


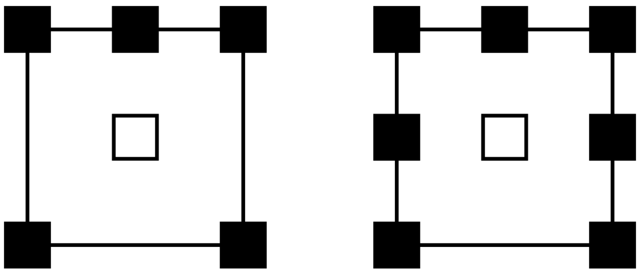
Figure 3.3: Label for 2.1 (Dolby surround).

3.1.4 Dolby Digital, 5.1 and 7.1

Digital audio introduction by Sony and Philips in 1982 has brought radical changes in multichannel audio world thanks to the appearance of laser-disc combined with contemporary data compression techniques. In fact, these concepts led to the realization of a new support able to store a huge amount of different nature information in digital format: the *DVD*. Hence, it was possible to place audio, video and many other information all in the same support and to use the available memory to store more audio channels.

Up to now the most used compression technologies are *Dolby Digital* and *DTS*, that employs a less destructive compression algorithm. Dolby Digital contains up to six discrete channels of sound (*5.1 Surround Sound* reproduction system). The most elaborate mode in common usage involves five channels for normal-range speakers (20 Hz – 20,000 Hz) (right front, center, left front, rear right, rear left) and one channel (20 Hz – 120 Hz allotted audio) for the sub-woofer driven low-frequency effects. Mono and stereo modes are also supported.

Figure 3.4: Formats allowed by digital supports



(a) Label for 5.1 (b) Label for 7.1

Another common standard derived from this technology is the *7.1 Surround Sound*, name for eight channel surround audio systems. It adds two additional rear speakers to 5.1 audio configuration using the standard front, center, and LFE (bass) speaker configuration, but in addition, includes two speakers positioned to the side and two to the rear. With such a sound configuration, almost every angle of sound can, theoretically, be captured for a completely immersive experience.

The just described rendering techniques rely on 2D reproduction systems and exploit mainly psychoacoustics concepts to obtain a better listener involvement into the sound scene that consist on controlling the amplitude and the delay of reproduced signals. Systems supporting these multichannel audio formats increase their handling complexity proportionally with their cardinality. Thus 3D reproduction systems, able to improve listener sound experience, require the employment of different rendering techniques to better exploit the higher number of needed loudspeakers and to reduce system handling complexity.

Ambisonics spatialization technique has been chosen to implement a software tool for 3D studio mixing as one of today's most known technologies, able to offer good sound rendering results by even employing low number of loudspeakers in reproduction system (starting from 9 one can already render satisfying tridimentional sound scenes).

3.2 DAW employment in studio mixing

The term *digital audio workstation (DAW)* was born to define an electronic system designed solely or primarily for recording, editing and playing back digital audio. DAWs were originally tape-less, microprocessor-based systems such as the Synclavier and Fairlight CMI. Modern DAWs are software running on computers with audio interface hardware.

A computer-based DAW has four basic components:

- a computer;
- an ADC-DAC (also called a sound card, audio interface, etc.);
- a digital audio editor software;
- and at least one input device for adding or modifying musical note data (this could be as simple as a mouse, and as sophisticated as a MIDI controller keyboard, or an automated fader board for mixing track volumes, etc.).

The computer acts as a host for the sound card and software and provides processing power for audio editing. The sound card (if used) or external audio interface typically converts analogue audio signals into digital form, and for playback converting digital to analogue audio; it may also assist in further processing the audio. The software controls all related hardware components and provides a user interface to allow for recording, editing, and playback. Most computer-based DAWs have extensive MIDI recording, editing, and playback capabilities, and some even have minor video-related features.

As software systems, DAWs could be designed with any user interface, but generally they are based on a multi-track tape recorder metaphor, making it easier for recording engineers and musicians already familiar with using tape recorders to become familiar with the new systems. Therefore, computer-based DAWs tend to have a *standard layout* (or *mixer layout*) which includes transport controls (play, rewind, record, etc.), track controls and/or a mixer, and a *waveform display* (or *edit layout*). In single-track DAWs, only one (mono or stereo form) sound is displayed at a time. Multi-track DAWs support operations on multiple tracks at once. Like a mixing console, each track typically has controls that allow the user to adjust the overall volume and stereo balance (pan) of the sound on each track. In a traditional recording studio additional processing is physically plugged in to the audio signal path, a DAW however can also route in software or uses software plugins to process the sound on a track.

DAWs are capable of many of the same functions as a traditional tape-based studio set-up, and in recent years have almost completely replaced them. Modern advanced recording studios may have multiple types of DAWs in them and it is not uncommon for a sound engineer and/or musician to travel with a portable laptop-based DAW, although interoperability between different DAWs is poor.

Commonly DAWs feature some form of automation, often performed through "envelopes". Envelopes are procedural line segment-based or curve-based interactive graphs. The lines and curves of the automation graph are joined by or comprised between adjustable points. By creating and adjusting multiple points along a waveform or control events, the user can specify parameters of the output over time (e.g., volume or pan). Automation data may also be directly derived from human gestures recorded by a control surface or controller. MIDI is a common data protocol used for transferring such gestures to the DAW.

It follows that today DAWs are widely involved in sound productions that make use of the audio rendering formats described in previous section. In fact, the above mentioned layouts subdivision allows first to edit each sound track using the edit layout with all its specific features (audio cut, past, copy, invert, envelope drawing, automation, etc.) then it gives the possibility to mix up the edited tracks by simple controlling amplitudes, equalizations, effects and filters superposition, routing, etc. All these functionality fully satisfy the requirements of a stereo, 5.1 or 7.1 mixing console but they are not sufficient to implement a 3D mixing where the user is supposed to have the possibility to control sources position adjusting dynamic parameters as azimuth, elevation and distance. However the mixer layout, especially the routing section, is helpful also for this purpose. Therefore, what is needed is a third layout to be placed between the edit and the mixer one. Such a

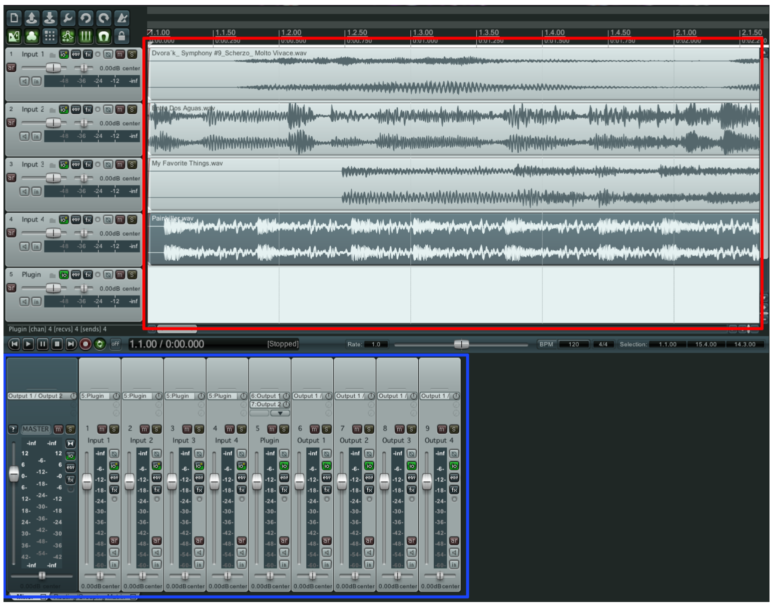


Figure 3.5: Reaper DAW Edit layout (in red) and Mixer layout (in blue).

layout should be called *3D spatialization layout* and must provide the acceptance of k edited input tracks, a processing stage that computes n output signals to be send to the mixer layout and a user interface that allows to place virtual sources everywhere in space.

The plugin notion is very suitable for this aim, in fact it is defined as an internal process that has to be recalled by the DAW and can receive and send signals controlling the adjustable process parameters interacting with a GUI (already implemented by the host as default).

There exist several plugin standards, the most diffused are:

- *VST* (standard created by Steinberg Media Technologies GmbH);
- *Direct-X* or *DX* (available for Microsoft Windows operative system);
- *Audio Units* or *AU* (available for Apple MacOSX);
- *RTAS* (for Pro Tools systems);
- *TDM* (used by Pro Tools too but employing external added DSP computational power instead of the CPU one)

Many plugin are available for more than one above listed standards to allow their usage on different platforms and DAWs and their commercial diffusion, but there are also some DAWs, such as Reaper or Audacity, that support different plugin standards. The plugin programming algorithm remains the same with different ‘plugs’, thus adaptors have been implemented to use VST plugins on DX hosts and vice-versa ,often called *‘wrappers’*.

The choice of the technology adopted to develop AmbiSound plugin has necessarily fall on the Steinberg’s VST standard, for sure the most diffuse one and the only supported both by Windows and MacOSX platforms. VST plugins can run on multi-platform hosts like Reaper, Audacity (both free-ware), Cubase and Nuendo and doesn’t requires a dedicate external hardware like TDM ones. Unfortunately this standard is not supported by two other DAWs often used in professional sound productions: Logic (MacOSX) and Pro Tools.

The VST standard version employed is the 2.x, chosen instead of the most recent 3.x to guarantee Reaper AmbiSound support – VST 3.x standard isn’t supported by the latter yet. In fact, the developed plugin has been designed to primarily run on Reaper host and to be completely configurable and best exploited inside this DAW.[1]

3.3 Project aims

Today’s Ambisonics software tools, already discussed in chapter 2, have many disadvantages and aspects to be improved.

Some of them requires both dedicated hardware (external DSP, soundboard, etc.) and software (as RTAS and TDM plugins that can be hosted only by Pro Tools), causing huge costs only for system configuration. In most cases they allow a limited reconfigurability of input sources and encoding/decoding order. The number of accepted input sources is often limited to only one signal except for those designed with Max/MSP that reach up to 16 inputs. The orders of encoding/decoding Ambisonics signals applied and correctly tested till today reach the third spherical harmonic.

Moreover today’s diffused Ambisonics tools are supposed to be employed inside big recording/mixing studios or reproduction environments and don’t take into consideration the artefacts that can be produced when using them in small environmental contexts.

[1]The reasons for choosing Reaper to be the proper host to run AmbiSound plugin will be analysed in detail on Sec. 4.3.

Looking specifically at VST plugins available for Ambisonics mixing also the low layout reconfigurability and platform supportability are limitations. Many of the Malham VST suite plugins [14] aren't supported by MacOSX yet and output layout can be configured by choosing a specific configuration from a limited list offering layout more suitable for 2D rendering.

Hence the purposes of the Ambisonics plugin developed in this project have been the following:

- increase the number of input signals processable by the plugin avoiding an excessive usage of the CPU and realizing something that can be considered a 3D mixing layout to be professionally used in DAWs.

- enlarge the reconfigurability of the reproduction system introducing more degrees of freedom in setting up the loudspeakers arrangement such as the user can decide the azimuth and elevation for each one and the distance from the listening point of the overall system.

- realize a 3D mixing tool that takes into account the possibility of being employed in little sound production studios using the unexploited near field compensation theory.

- develop a cross-platform Ambisonics plugin that can run also on free-ware DAWs (as Reaper) using a widely diffused technology (as VST) that doesn't necessitate dedicate hardware or software to be correctly configured allowing the possibility of large diffusion for this software tool also in small productions and even amateur ones.

Chapter 4

Techniques and Technologies

This chapter will analyse in detail the fundamental aspects on which the whole project is based. Starting form the Ambisonic Surround Sound theory developed from 1970s (4.1) up to now (4.1.2), proceeding by Virtual Studio Technology illustration (4.2), defining VST SDK features and the technology architecture. The last section will introduce Reaper digital audio workstation deepening the host important features clarifying the differences with the ones that permit a better use of AmbiSound plugin.

4.1 Ambisonics Surround Sound

Ambisonics theory was developed originally by Gerzon, Barton and Fellgett in 1970s [3]. Its aim was to develop a multichannel sound recording-reproduction technique capable to recreate a total listener immersion sensation into a reproduced sound environment.

In Ambisonic systems, sound signals together with their directional components are encoded into spherical harmonics vectorial form. Similarly to Holophonic systems (for more details see [1, ch. 10]) applying a decoding matrix to the just mentioned acquired components one can obtain the feeding signals for the loudspeaker placed around the listening area. The main advantages of an Ambisonics system are:

1. a simple recording/encoding system, using more microphones (at least four for 3D fields reconstruction) placed in the same point (or coincident) at the centre of the acoustic scene;

2. the decoding/reproduction system (formed by a loudspeakers array) is totally independent from the encoding stage. The signal

can be decoded generating a stereo output, an output supported by standard formats as Dolby Surround, 5.1 or 7.1 or even by multichannel (4, 6, 8, . . . loudspeakers) systems.

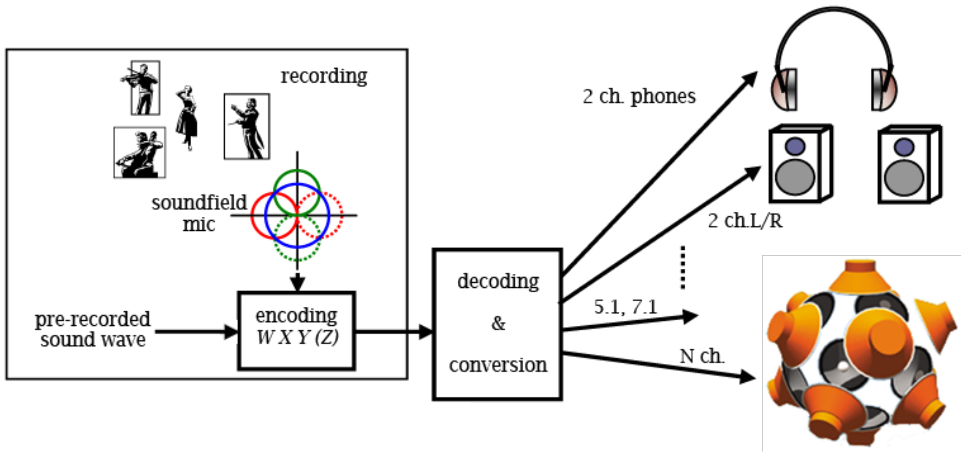


Figure 4.1: Ambisonics system basic scheme.

Some disadvantages, derived from the assumptions made in Ambisonics technique formulation, are that:

1. sources must irradiate only plane waves (they have to be – physically or virtually – placed far from the recording or listening point);

2. reproduced sound field must irradiate plane waves, thus loudspeakers must be placed conveniently far from the listening point.

The last problem can't be neglected especially in small sized reproduction environment (such as a car passenger compartment), instead, the former isn't too restrictive since any sound field configuration can be reconstructed as linear combination of plane waves.

In practice, using Ambisonics method, the acoustic field directional features are reconstructed by summing up the spherical harmonic components of the field itself, illustrated up to the 3rd order in figure 4.2. Each component should be acquired with a proper transducer having the same directional features (the 0th order component with an omnidirectional, the 1st order components with 'figure eight' microphones and so on). Alternatively an artificial sound-field can be obtained applying the Ambisonics encoding operation to mono signals.

Signals obtained from the transducers can be both re-encoded or decoded in many different formats that has been developed up to now. The most significant one, especially in this project, is the *B-format*, developed for studio equipments and employed to obtain the so call *Ambisonics signals* (the actual Ambisonics encoding): the set of signal spherical harmonics with directional features like those shown in figure 4.2.

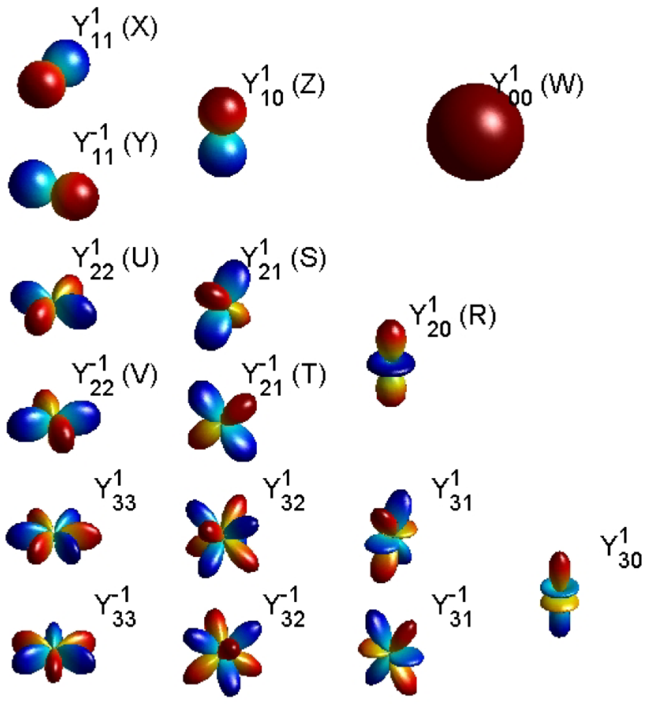


Figure 4.2: Acoustic field spherical harmonics.

4.1.1 Acoustic waves remarks

The spherical wave equation written for acoustic pressure p has the following form [2]:

$$\left[\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right)+\frac{1}{r^2}\frac{1}{\sin\vartheta}\frac{\partial}{\partial\vartheta}\left(\sin\vartheta\frac{\partial}{\partial\vartheta}\right)+\frac{1}{r^2\sin^2\vartheta}\frac{\partial^2}{\partial\varphi^2}+k^2\right]p=0. \tag{4.1}$$

Figure 4.3 shows the relation between the spherical coordinate system

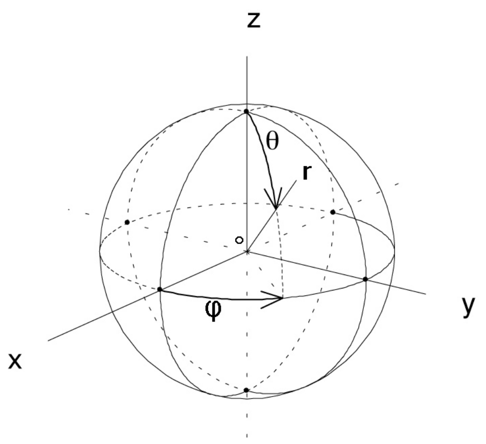


Figure 4.3: Spherical coordinate system.

to the rectangular one. The functional relationships are:

$$\begin{aligned} x &= r \sin\vartheta \cos\varphi \\ y &= r \sin\vartheta \sin\varphi \\ z &= r \cos\vartheta. \end{aligned} \tag{4.2}$$

The separation of variable method leads to a solution in terms of:

- spherical *Bessel* and *Neumann* functions of order m: $j_m(kr)$ and $n_m(kr)$ respectively, or alternatively in terms of convergent and divergent *Hankel* functions: $h_m^+ = h_m = j_m + in_m$ and $h_m^- = h_m^* = j_m - in_m$ respectively;
- polar associated *Legendre* polynomials functions $P_{mn}(\cos\vartheta) = P_{mn}(z)$ (from the relation $z = \cos\vartheta$) with $0 \le n \le m$;
- latitude functions $\cos n\varphi$ and $\sin n\varphi$;
- temporal functions $e^{i\omega t}$ and $e^{-i\omega t}$.

Assuming the product solution [7]

$$p(r,\vartheta,\varphi) = R(r)\Theta(\vartheta)\Phi(\varphi)T(t),$$

three different equations that leads to the solution can be obtained (written in compact form with the notation used in [7]):

$$p = \left\{ \begin{array}{c} h_m^+(kr) \\ h_m^-(kr) \end{array} \right\} \left\{ \begin{array}{c} \cos n\varphi \\ \sin n\varphi \end{array} \right\} P_{mn}(\cos\vartheta) \left\{ \begin{array}{c} e^{i\omega t} \\ e^{-i\omega t} \end{array} \right\}. \tag{4.3}$$

From here on the term $h^-_m(kr)$ will be neglected, considering only the progressive wave, together with the temporal functions, neglected for the presentation transparency, obtaining:

$$p = h^+_m(kr) \left\{ \begin{array}{c} \cos n\varphi \\ \sin n\varphi \end{array} \right\} P_{mn}(\cos\vartheta). \tag{4.4}$$

The '*spherical harmonics*' of degree m and order n can be defined, with $0 \le n \le m$, as the product [2]

$$Y^\sigma_{mn}(\vartheta,\varphi) = P_{mn}(\cos\vartheta) \times \begin{cases} \cos n\varphi & \text{if } \ \sigma = 1, \\ \sin n\varphi & \text{if } \ \sigma = -1 \quad \text{and} \quad n \ge 1. \end{cases} \tag{4.5}$$

To use these functions as basis for an expantion of the wave equation solution they must form an orthogonal or orthonormal basis. It's proved that orthogonal spherical harmonics can be obtained applying the Schmidt semi-normalization to associated Legendre polynomials (see [2]) that become:

$$\widetilde{P}_{mn}(\cos\vartheta) = \sqrt{\varepsilon_n \frac{(m-n)!}{(m+n)!}} P_{mn}(\cos\vartheta), \tag{4.6}$$

where $\varepsilon_n = 1$ if $n = 0$ and $\varepsilon_n = 2$ for $n \ge 1$. The definition of semi-normalized spherical harmonics follows as:

$$Y^{\sigma SN}_{mn}(\vartheta,\varphi) = \widetilde{P}_{mn}(\cos\vartheta) \times \begin{cases} \cos n\varphi & \text{if } \ \sigma = 1, \\ \sin n\varphi & \text{if } \ \sigma = -1 \quad \text{and} \quad n \ge 1. \end{cases} \tag{4.7}$$

Having fixed the order m of decomposition, they are $2m+1$ and form an orthogonal basis such in scalar product sense, i.e.:

$$\langle Y^{\sigma SN}_{mn}, Y^{\sigma' SN}_{m'n'} \rangle_{4\pi} = \frac{1}{2m+1} \delta_{mm'} \delta_{nn'} \delta_{\sigma\sigma'},$$

where δ is the Kronecker's delta and the scalar product between the spherical functions $F(\vartheta,\varphi)$ and $G(\vartheta,\varphi)$ is defined as:

$$\langle F, G \rangle_{4\pi} = \frac{1}{4\pi} \int_0^{2\pi} \int_0^{\pi} F(\vartheta,\varphi) G(\vartheta,\varphi) \sin\vartheta d\vartheta d\varphi.$$

Then, an orthogonal basis can be obtained from the semi-normalized spherical harmonics $Y^{\sigma SN}_{mn}$ multiplying them by $\sqrt{2m+1}$:

$$\Rightarrow Y^{\sigma N}_{mn}(\vartheta,\varphi) = \widetilde{Y}^\sigma_{mn}(\vartheta,\varphi) = Y^{\sigma SN}_{mn}\sqrt{2m+1}, \tag{4.8}$$

$$\Rightarrow \langle \widetilde{Y}^\sigma_{mn}, \widetilde{Y}^{\sigma'}_{m'n'} \rangle_{4\pi} = \delta_{mm'} \delta_{nn'} \delta_{\sigma\sigma'}.$$

The *Fourier-Bessel expantion* of the wave equation solution (4.4) can be written as:

$$p(r, \vartheta, \varphi) = \sum_{m,n,\sigma} A^{\sigma}_{mn} \widetilde{Y}^{\sigma}_{mn}(\vartheta, \varphi) i^m j_m(kr), \tag{4.9}$$

where A^{σ}_{mn}, called '*expantion coefficients*', can be computed for a radiant source, inside a sphere of radius R and centred in the origin, using the following expression:

$$\langle p_R, \widetilde{Y}^{\sigma}_{mn} \rangle_{4\pi} = i^m j_m(kr) A^{\sigma}_{mn}, \qquad \text{with} \quad p_R = (r = R, \vartheta, \varphi). \tag{4.10}$$

4.1.2 Ambisonics theory

Problem formulation: waves Fourier-Bessel expansion

Ambisonics technique is based on two fundamentals assumptions here transcribed for convenience:

- sources can only produce plan wave front. This is a non-restrictive condition, as any sound-field can be modelled as the linear combination of planar wave-fronts;
- in order to render the sound-field as a sum planar wave fronts, loudspeakers must be placed sufficiently far from the listener.[1]

For the first assumption it will be necessary to start Ambisonics treatment writing the Fourier-Bessel expantion for a tridimentional progressive and plane sound wave.

Before writing the acoustic pressure expression for such a sound wave it is better to introduce a different convention for spherical coordinates system representations. Figure 4.4 shows the new system that takes the listener head as the reference point for the system origin defining all the coordinate axes directions in relation with the head possible movements. Basing on this convention the (4.2) become:

$$\begin{aligned} x &= r \cos\vartheta \cos\varphi \\ y &= r \cos\vartheta \sin\varphi \\ z &= r \sin\vartheta. \end{aligned} \tag{4.11}$$

[1]Later on it will be explained how to 'straight' the non-planar wave front produced by a loudspeaker system near the listener.

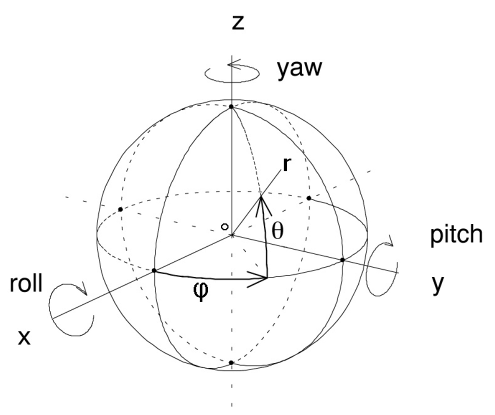


Figure 4.4: 'Listener related' spherical coordinate system.

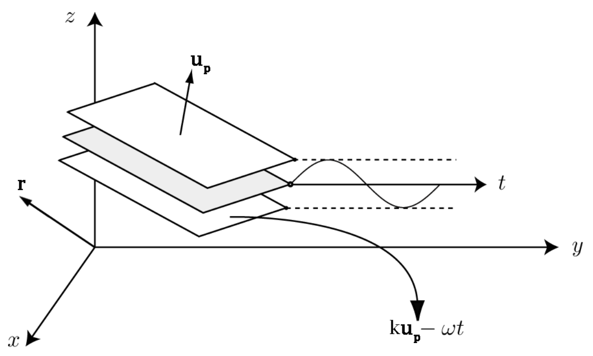


Figure 4.5: 3D Plane wave that propagates along the direction $\mathbf{u_p}$ *and received in* $\mathbf{r}$.

Now consider a sound-field produced by a tridimentional plane wave that propagates along the direction $\mathbf{u_p}$ (unitary vector) from the source S as shown in figure 4.5. Using the convention defined in figure 4.4, the expression of acoustic pressure p detected on a listening point, defined by $\mathbf{r} = r\mathbf{u_r}$, can be written as

$$p(r,\vartheta,\varphi) = P_r e^{i(k\mathbf{r}^T \cdot \mathbf{u_p} - \omega t)} = P_r e^{i(kr\mathbf{u_r}^T \cdot \mathbf{u_p} - \omega t)}, \tag{4.12}$$

$$\text{where} \quad \mathbf{u_r} = \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} = \begin{bmatrix} \cos\vartheta_r \cos\varphi_r \\ \cos\vartheta_r \sin\varphi_r \\ \sin\vartheta_r \end{bmatrix}, \qquad \mathbf{u_p} = \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} = \begin{bmatrix} \cos\vartheta_p \cos\varphi_p \\ \cos\vartheta_p \sin\varphi_p \\ \sin\vartheta_p \end{bmatrix} \tag{4.13}$$

and $k = \pm\omega/c$ is the wave number. Thus one can write p using the Fourier-Bessel expantion obtaining [2]:

$$p(r,\vartheta,\varphi) = P_r \sum_{m=0}^{\infty} (2m+1) i^m j_m(kr) e^{j\omega t} P_m(\mathbf{u_r}\cdot\mathbf{u_p}), \tag{4.14}$$

where $$P_m(\mathbf{u_r}\cdot\mathbf{u_p}) = \sum_{n=0}^{m} \varepsilon_n \frac{(m-n)!}{(m+n)!} P_{mn}(\sin\vartheta_r) P_{mn}(\sin\vartheta_p)\cos[n(\varphi_r-\varphi_p)].$$

Using the definition of normalized spherical harmonics $\widetilde{Y}^{\sigma}_{mn}(\vartheta,\varphi)$ given in (4.8), (4.14) becomes:

$$p(r,\vartheta,\varphi) = \sum_{m,n,\sigma} \widetilde{P}^{\sigma}_r \widetilde{Y}^{\sigma}_{mn}(\vartheta_r,\varphi_r) i^m j_m(kr) e^{j\omega t}, \tag{4.15}$$

$$\text{where} \quad P^{\sigma}_r = P_r \widetilde{Y}^{\sigma}_{mn}(\vartheta_p,\varphi_p).$$

Matching conditions definition

The reconstruction of the above mentioned sound-field can be done using an array of loudspeakers (see figure 4.6, where the z-axis has been omitted to simplify the picture).

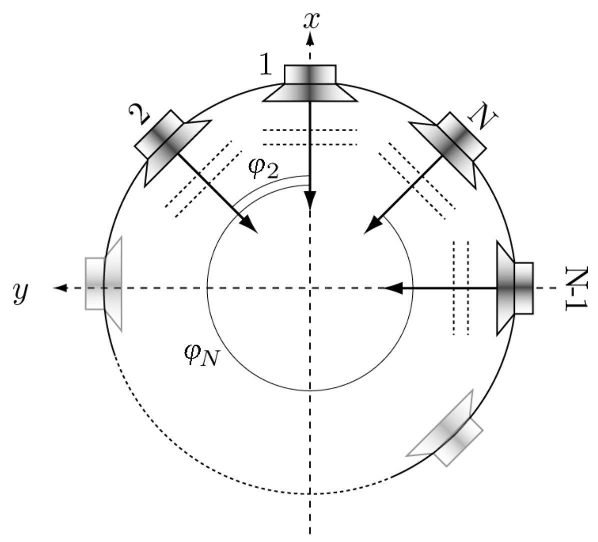


Figure 4.6: Circular array rig.

In a N loudspeakers layout only an approximation of the expantion (4.15) can be reached. The approximation quality increases as N goes to infinity. Every loudspeaker in figure 4.6 (considering also the polar component) emits a 3D plane wave if sufficiently far from the listening point, ideally placed in the system origin:

$$p_i(r,\vartheta,\varphi) = P_{ri}e^{i(kr\mathbf{u_r}\cdot\mathbf{u_{pi}}-\omega t)}, \tag{4.16}$$

where $\mathbf{u_{pi}}$ is the propagation direction of the wave irradiated by the *i-th* loudspeaker. Written using the Fourier-Bessel expantion, the latter becomes:

$$\begin{aligned} p_i(r,\vartheta,\varphi) &= \sum_{m,n,\sigma} \widetilde{P}^{\sigma}_{ri}\widetilde{Y}^{\sigma}_{mn}(\vartheta_r,\varphi_r)i^m j_m(kr)e^{j\omega t}, \\ \text{where} \quad \widetilde{P}^{\sigma}_{ri} &= P_{ri}\widetilde{Y}^{\sigma}_{mn}(\vartheta_{pi},\varphi_{pi}), \\ \Rightarrow \widetilde{p} = \sum_{i=1}^{N} p_i(r,\vartheta,\varphi) &= \sum_{i=1}^{N}\sum_{m,n,\sigma} \widetilde{P}^{\sigma}_{ri}\widetilde{Y}^{\sigma}_{mn}(\vartheta_r,\varphi_r)i^m j_m(kr)e^{j\omega t}, \end{aligned} \tag{4.17}$$

where $\widetilde{p}$ notation is used to indicate the reconstructed (approximated) version of $p(r,\vartheta,\varphi)$.

The aim of Ambisonics rendering stage is to reproduce the sound-field radiated by S by the cooperation of plane waves $p_i(r,\vartheta,\varphi)$ emitted by each loudspeaker in the rig, keeping as much as possible of the perceived position information from the sound-field. This means that in the listening positioning the following relation must be satisfied:

$$p(r,\vartheta,\varphi) \cong \widetilde{p}(r,\vartheta,\varphi), \tag{4.18}$$

where the symbol $\cong$ assumes the meaning of "as equal as possible". Using the Fourier-Bessel expantion for both members of (4.18) a set of equations can be derived, called *Matching Conditions*, that have to be satisfied in order to have the sound-field reconstruction. The Fourier-Bessel expantion of (4.18) is:

$$\sum_{m,n,\sigma} \widetilde{P}^{\sigma}_{r}\widetilde{Y}^{\sigma}_{mn}(\vartheta_r,\varphi_r)i^m j_m(kr)e^{j\omega t} \cong \sum_{i=1}^{N}\sum_{m,n,\sigma} \widetilde{P}^{\sigma}_{ri}\widetilde{Y}^{\sigma}_{mn}(\vartheta_r,\varphi_r)i^m j_m(kr)e^{j\omega t}. \tag{4.19}$$

Comparing (4.19) right and left sides term by term, one can state that

(4.18) and (4.19) are satisfied when

$$P_r^\sigma = \sum_{i=1}^{N} \widetilde{P}_{ri}^\sigma,$$

$$\Rightarrow P_r \widetilde{Y}_{mn}^\sigma(\vartheta_p, \varphi_p) = \sum_{i=1}^{N} P_{ri} \widetilde{Y}_{mn}^\sigma(\vartheta_{pi}, \varphi_{pi}). \tag{4.20}$$

Recalling the spherical harmonics definition, the expression (4.20) Can be decomposed into a set of three equations:

$$\begin{cases} P_r = \sum_{i=1}^{N} P_{ri} & \text{for} \quad n = 0 \\ P_r \widetilde{P}_{mn}(\sin\vartheta_p)\cos n\varphi_p = \sum_{i=1}^{N} P_{ri} \widetilde{P}_{mn}(\sin\vartheta_{pi})\cos n\varphi_{pi} & \text{for} \quad n \geq 1 \text{ and } \sigma = 1 \\ P_r \widetilde{P}_{mn}(\sin\vartheta_p)\sin n\varphi_p = \sum_{i=1}^{N} P_{ri} \widetilde{P}_{mn}(\sin\vartheta_{pi})\sin n\varphi_{pi} & \text{for} \quad n \geq 1 \text{ and } \sigma = -1 \end{cases} \tag{4.21}$$

called *Matching Conditions*, that are the core of Ambisonics Surround Sound technology. The index m is the order of reconstruction while $i = 1, \ldots, N$ is the loudspeaker index. As N tends to infinity, the N^{th}-order approximation tends to an exact matching and the *sweet-spot*[2] expands.

From the matching conditions it's noted that all is needed to reconstruct a sound-field p with N loudspeakers is its P_r and directional $P_r Y_{mn}^\sigma(\vartheta_p, \varphi_p)$ components. Ambisonics stores this information into a set of signals which cardinality depends on the order of reconstruction. Table 4.1 shows the spherical harmonic functions (semi-normalized) $\widetilde{Y}_{mn}^{\sigma SN}(\vartheta, \varphi)$ up to the third order of reconstruction.

Ambisonics encoding

The Fourier-Bessel expantion of the plane wave just discussed can be written into a more compact vectorial form. Neglecting the term responsible for time dependence and separating the term of order zero, p can be written as:

$$p = P_r \widetilde{Y}_{mn}^{+1}(\vartheta_p, \varphi_p) \widetilde{Y}_{mn}^{+1}(\vartheta_r, \varphi_r) j_0(kr) + \\ P_r \sum_{mn\sigma} \widetilde{Y}_{mn}^\sigma(\vartheta_p, \varphi_p) \widetilde{Y}_{mn}^\sigma(\vartheta_r, \varphi_r) i^m j_m(kr), \tag{4.22}$$

$$\text{for} \quad m, n = 1, \ldots, \infty \quad \text{and} \quad \sigma = \begin{cases} +1 \\ -1. \end{cases}$$

The (4.22) first addend is equal to $P_r j_0(kr)$ because $\widetilde{Y}_{00}^{+1}(\vartheta, \varphi)$ is equal to 1.[3]

[2]area around the listening point where the perception stays unaltered

[3]for m and n equal to 0 and $\sigma = +1$, $\cos(n\varphi) = 1$, the associated Legendre polynomial $P(\cos\vartheta) = 1$ [7] and the normalization term $\sqrt{2m+1} = 1$

Order	$\binom{\sigma}{mn}$	$Y_{mn}^{\sigma SN}(\mathbf{u})$	$Y_{mn}^{\sigma SN}(\vartheta,\varphi)$
0	$\binom{1}{00}$	1	1
	$\binom{+1}{11}$	u_x	$\cos\vartheta\cos\varphi$
1	$\binom{-1}{11}$	u_y	$\cos\vartheta\sin\varphi$
	$\binom{1}{10}$	u_z	$\sin\vartheta$
	$\binom{+1}{22}$	$\sqrt{3}(u_x^2-u_y^2)$	$\frac{\sqrt{3}}{2}\cos(2\varphi)\cos^2\vartheta$
	$\binom{-1}{22}$	$\sqrt{3}(u_xu_y)$	$\frac{\sqrt{3}}{2}\sin(2\varphi)\cos^2\vartheta$
2	$\binom{+1}{21}$	$\sqrt{3}(u_xu_z)$	$\frac{\sqrt{3}}{2}\cos(2\varphi)\sin\vartheta$
	$\binom{-1}{21}$	$\sqrt{3}(u_yu_z)$	$\frac{\sqrt{3}}{2}\sin(2\varphi)\sin\vartheta$
	$\binom{1}{20}$	$\frac{3(u_z^2-1)}{2}$	$\frac{3(\sin^2\vartheta-1)}{2}$
	$\binom{+1}{33}$	$\sqrt{\frac{5}{8}}u_x(u_x^2-3u_y^2)$	$\frac{\sqrt{5}}{8}\cos(3\varphi)\cos^3\vartheta$
	$\binom{-1}{33}$	$\sqrt{\frac{5}{8}}u_y(3u_x^2-u_y^2)$	$\frac{\sqrt{5}}{8}\sin(3\varphi)\cos^3\vartheta$
	$\binom{+1}{32}$	$\sqrt{15}u_z\frac{u_x^2-u_y^2}{2}$	$\frac{\sqrt{15}}{2}\cos(2\varphi)\sin\vartheta\cos^2\vartheta$
3	$\binom{-1}{32}$	$\sqrt{15}u_xu_yu_z$	$\frac{\sqrt{15}}{2}\sin(2\varphi)\sin\vartheta\cos^2\vartheta$
	$\binom{+1}{31}$	$\sqrt{\frac{3}{8}}u_x(5u_z^2-1)$	$\frac{\sqrt{3}}{8}\cos\varphi\cos\vartheta(5\sin^2\vartheta-1)$
	$\binom{-1}{31}$	$\sqrt{\frac{3}{8}}u_y(5u_z^2-1)$	$\frac{\sqrt{3}}{8}\sin\varphi\cos\vartheta(5\sin^2\vartheta-1)$
	$\binom{1}{30}$	$\frac{u_z(5u_z^2-3)}{2}$	$\frac{\sin\vartheta(5\sin^2\vartheta-3)}{2}$

Table 4.1: Semi-normalized spherical harmonic functions up to 3rd order taken from [2] – note that ϑ direction is inverted according to the convention adopted, defined in figure 4.4.

The vectorial form can be defined introducing the vectors:

$$\begin{aligned}\mathbf{c}^T &= \begin{bmatrix}1 & \widetilde{Y}_{11}^{+1}(\vartheta_p,\varphi_p) & \widetilde{Y}_{11}^{-1}(\vartheta_p,\varphi_p) & \cdots & \widetilde{Y}_{mn}^{\sigma}(\vartheta_p,\varphi_p) & \cdots\end{bmatrix}\\ &= \begin{bmatrix}c_0 & c_1 & c_2 & \cdots & c_m & \cdots\end{bmatrix},\end{aligned}$$

$$\mathbf{h}^T = \begin{bmatrix}j_0(kr) & ij_1(kr)\widetilde{Y}_{11}^{+1}(\vartheta_r,\varphi_r) & ij_1(kr)\widetilde{Y}_{11}^{-1}(\vartheta_r,\varphi_r) & \cdots & i^m j_m(kr)\widetilde{Y}_{mn}^{\sigma}(\vartheta_r,\varphi_r) & \cdots\end{bmatrix},$$

hence the (4.22) becomes:

$$p(r,\vartheta,\varphi) = P_r\mathbf{c}^T\cdot\mathbf{h}. \tag{4.23}$$

This formulation enables a decoupling between the direction $\mathbf{u_r}$ and $\mathbf{u_p}$. The plane wave information about spatial distribution, related to angles ϑ_p and φ_p is totally described by the vector $\mathbf{c}$. Ambisonics encoding signals can be simply obtained multiplying vector $\mathbf{c}$ by the the plane wave amplitude P_r:

$$\begin{aligned}\mathbf{b}^T = P_r\mathbf{c}^T &= P_r\begin{bmatrix}1 & \widetilde{Y}_{11}^{+1} & \widetilde{Y}_{11}^{-1} & \cdots & \widetilde{Y}_{mn}^{\sigma} & \cdots\end{bmatrix}\\ &= P_r\begin{bmatrix}c_0 & c_1 & c_2 & \cdots & c_m & \cdots\end{bmatrix}\\ &= \begin{bmatrix}W & X & Y & Z & \cdots\end{bmatrix}.\end{aligned} \tag{4.24}$$

Vector $\mathbf{b}$ contains the so called *Ambisonics signals* (*B-format* encoding).

The Ambisonics encoding stage estimates the elements in the vector $\mathbf{c}$. These elements can be somehow measured or created using a set of encoding equations that act as a spherical harmonics sampler for the acoustic field. This second approach is the one been followed for the project development, considering only the sound-field pressure as known so that each source has been implemented as a mono wave file $s(t)$.

A monophonic signal can be encoded in Ambisonics, fixing its direction of arrival (DOA) ϑ_p, φ_p or $\mathbf{u_p}$, by using a simplified version of encoding equations (4.23) where P_r can be substituted with $s(t)$, obtaining the equations set showed in table 4.2.

Ambisonics decoding

The notation used in (4.23) can be employed also for the sound-field generated by a single loudspeaker:

$$p_i(r,\vartheta,\varphi) = P_{ri}\mathbf{c_i}^T\cdot\mathbf{h}, \tag{4.25}$$

$$\text{where}\quad \mathbf{c_i}^T = \begin{bmatrix}c_{0i} & c_{1i} & c_{2i} & \ldots & c_{mi} & \ldots\end{bmatrix}.$$

Order	B_{mn}^{σ}	$\binom{\sigma}{mn}$	Encoding Functions
0	W	$\binom{1}{00}$	1
1	X	$\binom{+1}{11}$	$s(t)\cos\vartheta\cos\varphi$
	Y	$\binom{-1}{11}$	$s(t)\cos\vartheta\sin\varphi$
	Z	$\binom{1}{10}$	$s(t)\sin\vartheta$
2	R	$\binom{+1}{22}$	$s(t)\frac{\sqrt{3}}{2}\cos(2\varphi)\cos^2\vartheta$
	S	$\binom{-1}{22}$	$s(t)\frac{\sqrt{3}}{2}\sin(2\varphi)\cos^2\vartheta$
	T	$\binom{+1}{21}$	$s(t)\frac{\sqrt{3}}{2}\cos(2\varphi)\sin\vartheta$
	U	$\binom{-1}{21}$	$s(t)\frac{\sqrt{3}}{2}\sin(2\varphi)\sin\vartheta$
	V	$\binom{1}{20}$	$s(t)\frac{3(\sin^2\vartheta-1)}{2}$
3	K	$\binom{+1}{33}$	$s(t)\frac{\sqrt{5}}{8}\cos(3\varphi)\cos^3\vartheta$
	L	$\binom{-1}{33}$	$s(t)\frac{\sqrt{5}}{8}\sin(3\varphi)\cos^3\vartheta$
	M	$\binom{+1}{32}$	$s(t)\frac{\sqrt{15}}{2}\cos(2\varphi)\sin\vartheta\cos^2\vartheta$
	N	$\binom{-1}{32}$	$s(t)\frac{\sqrt{15}}{2}\sin(2\varphi)\sin\vartheta\cos^2\vartheta$
	O	$\binom{+1}{31}$	$s(t)\frac{\sqrt{3}}{8}\cos\varphi\cos\vartheta(5\sin^2\vartheta-1)$
	P	$\binom{-1}{31}$	$s(t)\frac{\sqrt{3}}{8}\sin\varphi\cos\vartheta(5\sin^2\vartheta-1)$
	Q	$\binom{1}{30}$	$s(t)\frac{\sin\vartheta(5\sin^2\vartheta-3)}{2}$

Table 4.2: Ambisonics encoding functions for a monophonic signal $s(t)$, derived from semi-normalized spherical harmonics up to the 3rd order taken from [2].

The whole sound-field perceived in the listening area (sweet-spot around the system origin) becomes:

$$\begin{aligned} \widetilde{p} = \sum_{i=1}^{N} P_i = \sum_{i=1}^{N} P_{ri}\mathbf{c_i}^T \cdot \mathbf{h} \\ &= P_{r1}\mathbf{c_1}^T \cdot \mathbf{h} + P_{r2}\mathbf{c_2}^T \cdot \mathbf{h} + \cdots + P_{rN}\mathbf{c_N}^T \cdot \mathbf{h} \\ &= \mathbf{a}^T\mathbf{C}^T\mathbf{h}, \end{aligned} \tag{4.26}$$

$$\text{with} \quad \mathbf{a}_T = \begin{bmatrix} P_{r1} & P_{r2} & \cdots & P_{rN} \end{bmatrix},$$

$$\begin{aligned} \mathbf{C} &= \begin{bmatrix} \mathbf{c_1} & \mathbf{c_2} & \cdots & \mathbf{c_N} \end{bmatrix} \\ &= \begin{bmatrix} c_{01} & c_{02} & \cdots & c_{0N} \\ c_{11} & c_{22} & \cdots & c_{1N} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}. \end{aligned}$$

For the perfect reconstruction the matching conditions have to be satisfied so from (4.18):

$$\begin{gathered} p = \widetilde{p} \\ P_r\mathbf{c}^T\mathbf{h} = \mathbf{b}^T\mathbf{h} = \mathbf{a}^T\mathbf{C}^T\mathbf{h}, \end{gathered} \tag{4.27}$$

from which one can derive the decoding equations in matrix form:

$$\begin{aligned} &\mathbf{b}^T\mathbf{h} = \mathbf{a}^T\mathbf{C}^T\mathbf{h} \\ &\Rightarrow \mathbf{b} = \mathbf{C}\mathbf{a}, \end{aligned} \tag{4.28}$$

where **b** represent the Ambisonics signals encoded from the sound source and **a** the vector of speaker-feeds to be found. The equations set(4.28) can be solved with least norm technique:[4]

$$\begin{aligned} \Rightarrow \mathbf{a} &= \mathbf{C}^T(\mathbf{C}\mathbf{C}^T)^{-1}\mathbf{b} \\ &= \mathbf{C}^\dagger\mathbf{b} \\ &= \mathbf{D}\mathbf{b}, \end{aligned} \tag{4.29}$$

where $\mathbf{D} = \mathbf{C}^\dagger$ is called the *pseudo-inverse* or *left inverse* of matrix **C**.

[4]finding the optimal solution $\mathbf{a_{ln}}$ for **a** such that $\|\mathbf{a}\|$ is minimized – optimal in least norm sense.

To compute the speaker-feeds in (4.28), the matrix $\mathbf{C}$ with dimension $(m+1)^2 \times N$ has to be fat, i.e.

$$N \geq (m+1)^2, \tag{4.30}$$

otherwise the system becomes over specified – more equations than variables. This is an hardware constraint for the Ambisonics decoding stage because a N-loudspeakers system will be able to reproduce a sound-field with maximum accuracy equal to the spherical harmonic order m that satisfies (4.30)

Regularity property

A loudspeakers set-up for Ambisonics reproduction can be defined:

- regular if the basis $\mathbf{c_i}$ preserve the orthonormal property, i.e. if vectors $\mathbf{c_i}$ are orthonormal in the scalar product sense

  $$\langle \mathbf{c_i}, \mathbf{c_{i'}} \rangle_N = \delta_{mm'} \delta_{nn'} \delta_{\sigma\sigma'},$$

  that can be traduced in the compact form

  $$\frac{1}{N} \mathbf{C}\mathbf{C}^T = \mathbf{I_k}, \tag{4.31}$$

  where $\mathbf{I_k}$ is the identity matrix of rank $K = (m+1)^2$,

- semi-regular if the basis $\mathbf{c_i}$ preserve the orthogonality property. In this case $\mathbf{C}\mathbf{C}^T$ becomes a diagonal matrix.

  It follows that, for regular layout systems, in the Ambisonics decoding stage the matrix $\mathbf{C}^\dagger$ can be obtained just as $\frac{1}{N}\mathbf{C}^T$. In fact substituting (4.31) in the pseudo-inverse formula it comes that

  $$\mathbf{C}^\dagger = \mathbf{C}^T (\mathbf{C}\mathbf{C}^T)^{-1} = \mathbf{C}^T (N \mathbf{I_k}^{-1}) = \frac{1}{N} \mathbf{C}^T.$$

A 2D reproduction layout is regular when its loudspeakers are placed as vertices of a regular polygon.

The case of 3D spatial sampling is more complicated, in fact the limited number of existing regular polyhedra together with the constraint (4.30), that fixes the number of vertices, don't allow an easy solution for the layout shape research. For example a regular polyhedron of 20 vertices don't represent a strictly regular support for a 3rd order reproduction that has

16 components. [2] A better sampling of the sphere, allowing to obtain a 'more regular' reproduction support, can be done applying triangular tessellation to the sphere (as shown in figure 4.7), increasing the reproduction quality with the number of triangles (decreasing their size) and the order m respectively.

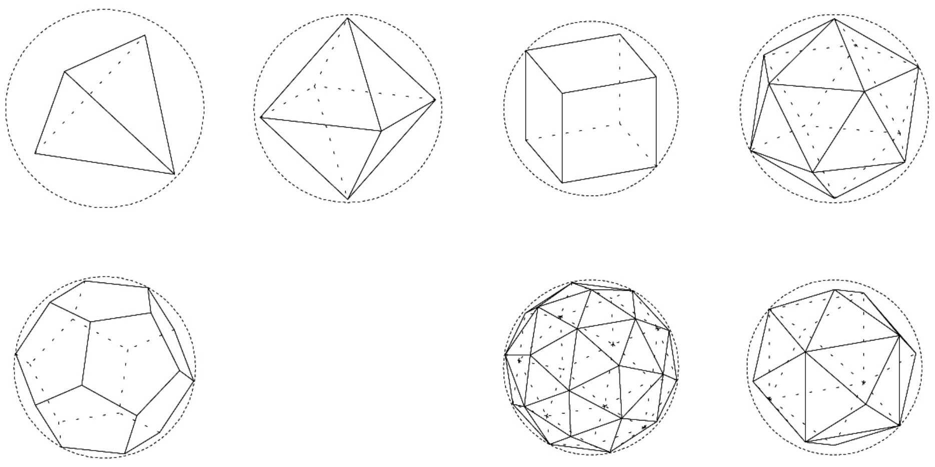


Figure 4.7: From the top to the bottom and from left to right, the five existent regular polyhedra (also called platonic solids): tetrahedron, octahedron, cube, icosahedron and dodecahedron. Then two examples of tessellated spheres with 32 vertices and 14 vertices respectively

Independently from the loudspeakers number, the kind of systems shown in *figure*, i.e. reproduction systems that include the listener, go inevitably to deal with several practical problems: the loudspeakers placement under listener feet and the interpersonal masking problem. For these reasons the configuration that are typically employed in 3D sound reproductions are the hemispherical ones (see figure 4.8). Apart from the cubical one and parallelepipeds, used with a reduced audience and for first order systems.

4.1.3 Near field effects

Starting from expression (4.24), the spherical decomposition of a plane wave of incidence (ϑ_p, φ_p) carrying a signal S, leads to the simple expression of Ambisonics components:

$$B^{\sigma}_{mn} = S \cdot \widetilde{Y}^{\sigma}_{mn}. \tag{4.32}$$

Thus a far field source signal S is encoded by simply applying real encoding gains, which are the spherical harmonic functions. By the way, this means

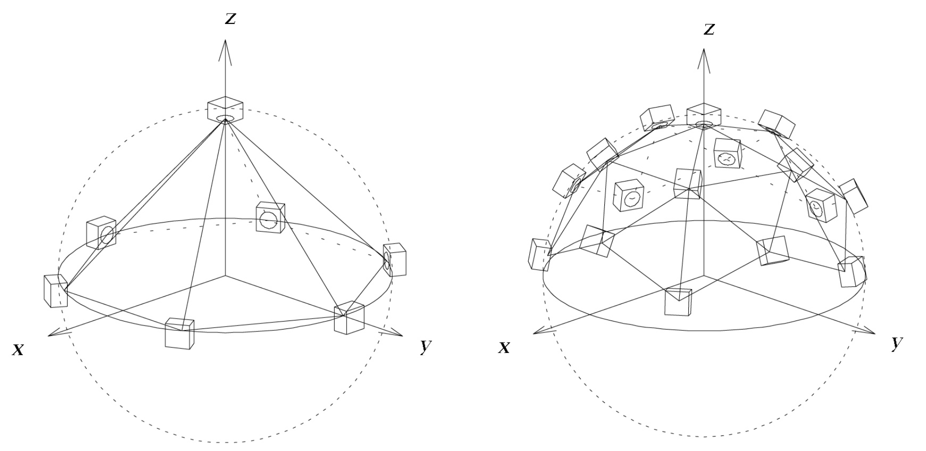


Figure 4.8: Two examples of hemispherical Ambisonics reproduction layout: a regular pyramid with hexagonal base and the top part of the 32-vertices polyhedron in figure 4.7.

that the sound-field "derivatives" (spherical components of order bigger then 0) properties don't vary with the frequency.

Spherical wave decomposition: near field effect

The modelling of the near field effect due to finite distance sources points out a fundamental issue of natural or realistic sound fields that must be considered as spherical waves. In [2] it is shown that the spherical decomposition for this kind of wave, radiated by a point source at $(\rho, \vartheta_p, \varphi_p)$, leads to:

$$B^{\sigma}_{mn} = S \cdot \Gamma_m(k\rho) \cdot \widetilde{Y}^{\sigma}_{mn}(\vartheta_p, \varphi_p), \tag{4.33}$$
$$\text{with} \quad \Gamma_m(k\rho) = kd_{ref} i^{-(m+1)} h^{-}_m(k\rho),$$

where $h^{-}_m(k\rho) = j_m(k\rho) - in_m(k\rho)$ are the divergent spherical Hankel functions, and d_{ref} is a reference distance. More conveniently it can be considered S as the pressure field captured at the origin O, so that the $1/\rho$ attenuation and the delay ρ/c due to the finite distance propagation, which are reflected by $\Gamma_0(k\rho)$ are supposed to be already modelled. By removing the latter form (4.33), the encoding equations of a source at finite distance

ρ become:

$$B^{\sigma}_{mn} = S \cdot F^{(\rho/c)}_m(\omega) \cdot \widetilde{Y}^{\sigma}_{mn}(\vartheta_p, \varphi_p), \quad \omega = 2\pi f \tag{4.34}$$

$$\text{where} \quad F^{(\rho/c)}_m(\omega) = \frac{\Gamma_m(k\rho)}{\Gamma_0(k\rho)} = \sum_{n=0}^{m} \frac{(m+n)!}{(m-n)!n!} \left(\frac{-ic}{\omega\rho}\right)^n . \tag{4.35}$$

Such a finite distance encoding involves transfer functions $F_m(\omega)$ that affect Ambisonics components especially at low frequencies, as shown in figure 4.9. In other words by comparison with the plane wave case of (4.32): the near field disturbs the sound field derivatives as much as the source distance (i.e. the curvature radius) is small regarding the wavelength, and as the derivative order m is high. [6]

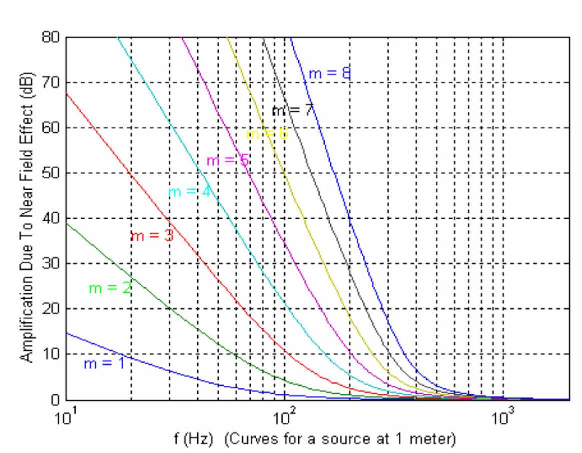


Figure 4.9: Low frequency infinite boost ($m \times dB/octave$) of Ambisonics components due to near field effect.

Encoding format limitations

The transfer functions $F_m(\omega)$ typically reflect integrating filters (for $m \geq 1$), which are unstable by nature (infinite bass-boost shown in figure 4.9). First order encoding may still remain practicable provided that every encoded signal S is centred (null mean value), but it is no longer the case for higher orders.

Not only (4.34) involves impracticable filters for virtual source encoding, but since it also models the physical reality, it would imply that the Ambisonics representation of any natural sound field may have infinite amplitude components. This finally means that in spite of being mathematically powerful, the currently adopted HOA encoding format is unable to physically represent and convey (i.e. by finite amplitude signals) natural or realistic sound fields, since these always include more or less near field sources.

By addressing the decoding and reproduction issues, and introducing the loudspeaker near field modelling at this stage, the following section suggests a key to a viable encoding format.

Wave front curvature distortion and bass-boost effect

Figure 4.10 shows the case of an encoded plane wave. Its left parts, which report a traditional decoding, show that the synthetic wave has the expected propagation direction from the centred listener point of view. Nevertheless, it clearly appears that with a high (15th) order rendering, this is not a plane wave that is reconstructed, but a spherical one, as being radiated by a point on the loudspeaker boundary. Therefore off-centred listeners localise the virtual source on this point and not in the direction of the original plane wave. This wave curvature distortion seems to have little impact on the directional effect for a centred listener. Nevertheless, even for this position and depending on the actual array radius, the difference with a true plane may be audible as the so-called "bass-boost effect", and also as an emphasised *Interaural Level Difference* (*ILD*).

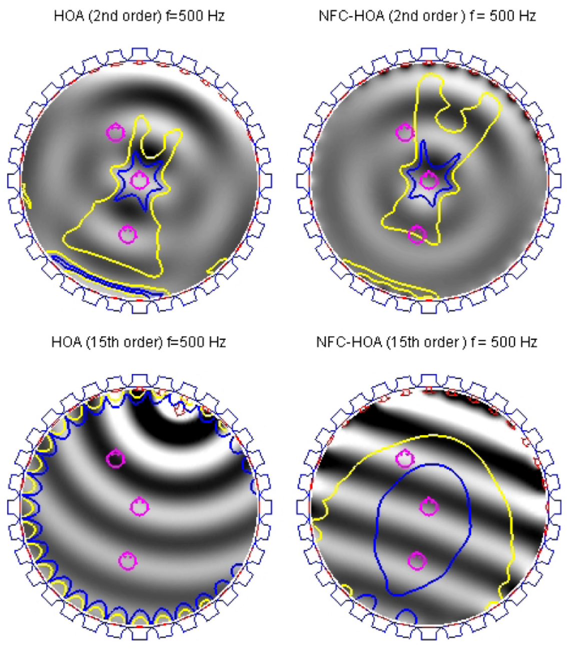

HOA (2nd order) f=500 Hz
NFC-HOA (2nd order) f= 500 Hz
HOA (15th order) f=500 Hz
NFC-HOA (15th order) f= 500 Hz

Figure 4.10: Reproduction of an encoded plane wave without (left) and with (right) loudspeaker near field compensation (NFC). 2nd order (top) and 15th order (bottom) Ambisonics.

Loudspeaker near field compensation

In the context of earlier first Ambisonics systems, Gerzon recommended to compensate for the bass-boost effect due to the finite distance of loudspeakers. Considering higher orders and with the more general aim to preserve the original curvature of the encoded wave fronts, it is now suggested to introduce the loudspeaker near field modelling into the re-encoding equation (4.29). That means that the elements $\widetilde{Y}^{\sigma}_{mn}(\vartheta_{pi}, \varphi_{pi})$ of the re-encoding matrix $\mathbf{C}$ would have to be "multiplied" by the near field transfer function $F^{R/c}_m(\omega)$ of same order.[5] Finally, this leads to the following decoding operation [2, 6]:

$$\mathbf{a} = \mathbf{D} \cdot diag \left[\cdots \quad \frac{1}{F^{R/c}_m(\omega)} \quad \cdots \right] \cdot \mathbf{b}, \tag{4.36}$$

where the decoding matrix is the same as defined in (4.29). Thus, this new decoding consists in applying a near field compensation $1/F^{R/c}_m(\omega)$ to the Ambisonics components B^{σ}_{mn} (elements of vector $\mathbf{b}$) before decoding them classically. Unlike the near field modelling transfer functions $F^{R/c}_m(\omega)$, filters $1/F^{R/c}_m(\omega)$ are practicable and stable.

As a result, the plane wave is actually reconstructed without curvature distortion, which is clearly illustrated for the 15th order by figure 4.10 (right-bottom part).

Distance coding / Near Filed Control Filters

After proving that for a proper sound field reconstruction, one has to compensate for the loudspeaker near field effect anyway, a reasonable question can be why not introduce this near field compensation from the encoding stage. As a matter of fact, it rapidly appears that combining it to the near field modelling of the virtual source leads to apply finite amplitude transfer functions.

The combination of near field effect (for a source distance ρ) and compensation (for a loudspeakers distance R) leads to the following transfer functions:

$$H^{NFC(\rho/c, R/c)}_m(\omega) = \frac{F^{\rho/c}_m(\omega)}{F^{R/c}_m(\omega)}. \tag{4.37}$$

Figure 4.11 shows that they cause a finite, low frequency amplification $m \times 20\log_{10}(R/\rho)$ (in dB), which is positive for enclosed sources ($\rho < R$) and negative for outside sources ($\rho > R$). They can be practically implemented

[5]Where R is the loudspeakers distance from the system origin

as stable filters called '*Near Field Coding*' or '*Control*' filters, or simply '*NFC filters*'.

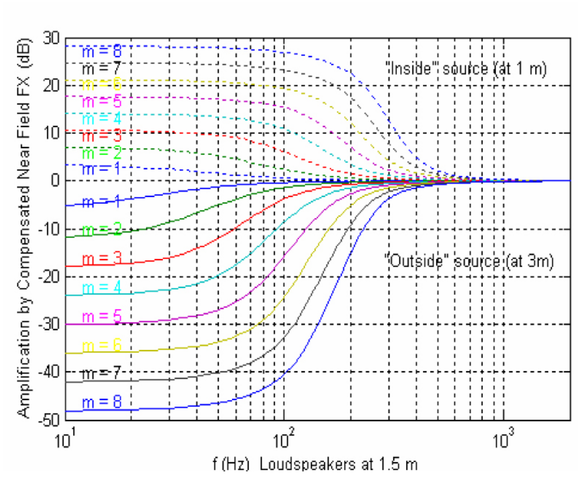


Figure 4.11: NFC filters frequency responses: finite amplification of Ambisonics components from pre-compensated Near Field Effect (dashed lines: $\rho/R = 2/3$; continuous lines: $\rho/R = 2$).

Now encoding equations (4.34) are replaced by the positional encoding equations:

$$\breve{B}_{mn}^{\sigma NFC(R/c)} = S \cdot H_m^{NFC(\rho/c,R/c)}(\omega) \cdot \widetilde{Y}_{mn}^{\sigma}(\vartheta_p, \varphi_p). \tag{4.38}$$

This new positional encoding scheme completes the earlier, purely directional one by introducing a distance-coding module (figure 4.12). The latter consist of a NFC filter bank, which is preferably placed before the directional gain control in order to factorise the filtering of same order group components. It's worth recalling that with such an encoding scheme, the encoded sound field only requires an "ordinary" matrix decoding (4.29). [6]

Design of distance coding filters

For the above defined NFC filters is preferable a lower cost, IIR (Infinite Impulse Response) filter implementation. It appears that the bilinear transform, which is well known in digital filter design, does perfectly the work. The following lines define the successive steps of NFC filters design strategy. With the final aim of describing filters with second and first order sections, their poles and zeros must be found first. For convenience, this pole-zero extraction is preferably done directly on the analogue domain filters, before applying the bilinear transform. So, one can start rewriting the near field

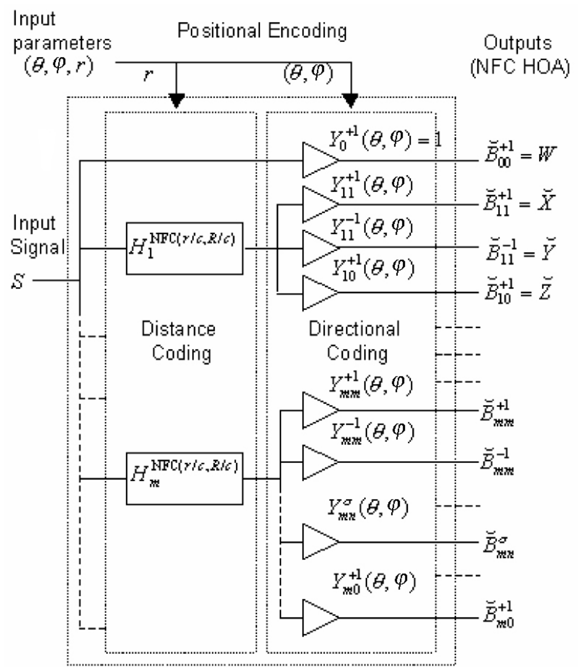

Input parameters (θ, φ, r)
Positional Encoding
r
(θ, φ)
Outputs (NFC HOA)
Input Signal
S
$H_1^{NFC(r/c,R/c)}$
$H_m^{NFC(r/c,R/c)}$
Distance Coding
Directional Coding
$Y_0^{+1}(\theta,\varphi)=1$
$Y_{11}^{+1}(\theta,\varphi)$
$Y_{11}^{-1}(\theta,\varphi)$
$Y_{10}^{+1}(\theta,\varphi)$
$Y_{mm}^{+1}(\theta,\varphi)$
$Y_{mm}^{-1}(\theta,\varphi)$
$Y_{mn}^{\sigma}(\theta,\varphi)$
$Y_{m0}^{+1}(\theta,\varphi)$
$\breve{B}_{00}^{+1} = W$
$\breve{B}_{11}^{+1} = \breve{X}$
$\breve{B}_{11}^{-1} = \breve{Y}$
$\breve{B}_{10}^{+1} = \breve{Z}$
$\breve{B}_{mm}^{+1}$
$\breve{B}_{mm}^{-1}$
$\breve{B}_{mn}^{\sigma}$
$\breve{B}_{m0}^{+1}$

Figure 4.12: NFC-HOA positional encoding of a virtual sound source: a distance-coding unit (NFC filter bank) completes the directional encoding.

modelling transfer function (4.35) as the Laplace function:

$$F_m^{(\tau)}(s) = \sum_{n=0}^{m} \frac{(m+n)!}{(m-n)!n!}(2\tau s)^{-n}, \tag{4.39}$$

with respectively $\tau = \rho/c$ or $\tau = R/c$ if the matter is to simulate the virtual source distance or to compensate for the loudspeaker near field.

Pole-zero extraction. to find the poles and zeros of filter $F_m(s)$, it is convenient to set $X = 2\tau s$ and rewrite (4.39) as:

$$\begin{aligned} &F_m(X) = X^{-m} Q_m(X), \\ \text{where}\quad &Q_m(X) = \sum_{n=0}^{m} \frac{(m+n)!}{(m-n)!n!} X^{m-n} = \prod_{q=1}^{m} (X - X_{m,q}). \end{aligned} \tag{4.40}$$

While the poles of $F_m(p)$ are clearly null, its zeros p_{mq} appear to be related to the complex roots $X_{m,q} = 2\tau p_{mq}$ (with $0 \leq q \leq m$) of the polynomial $Q_m(X)$, which is a particular case of the generalized Bessel polynomials.

Some approximated values for $X_{m,q}$ are given in table 4.3. In the following $X_{m,q}$ roots are considered as arranged in decreasing order of imaginary parts.

m	Roots $X_{m,q}$ of Q_m		
1	-2		
2	$-3.0000 \pm i1.732$		
3	$-3.6778 \pm i3.5088$;	-4.6444	
4	$-4.2076 \pm i5.3158$;	$-5.7924 \pm i1.7345$	
5	$-4.6493 \pm i7.1420$;	$-6.7039 \pm i3.4853$;	-7.2935
6	$-5.0319 \pm i8.9853$;	$-7.4714 \pm i5.2525$;	$-8.4967 \pm i1.7350$

Table 4.3: Roots of Q_m up to the first six orders m.

Applying the bilinear transform. The second step is to transpose the pole-zero filter form from the analogue (Laplace) domain to the digital domain (z-transform). For this purpose,the bilinear transform consist in applying the substitution $p = 2f_s(1 - z^{-1})/(1 + z^{-1})$:

$$\Rightarrow F_m^\tau(z) = F_m^\tau(p)\Big|_{p=2f_s\frac{(1-z^{-1})}{(1+z^{-1})}}, \tag{4.41}$$

with f_s being the sampling frequency. Therefore, it's easy to write the zeros z_{mq} of $F_m(z)$ in terms of the zeros p_{mq} of the Laplace function $F_m(p)$:

$$z_{mq}^{-1} = \frac{1 - \frac{p_{mq}}{2f_s}}{1 + \frac{p_{mq}}{2f_s}}, \qquad \text{i.e.} \quad z_{mq}(\tau) = \frac{1 + X_{m,q}/(4\tau f_s)}{1 - X_{m,q}/(4\tau f_s)}. \tag{4.42}$$

Finally, by setting $X = 2\tau p = \alpha(1 - z^{-1})/(1 + z^{-1})$, with $\alpha = 4f_s\tau$, the "near field compensating" digital filter can be written in the pole-zero form:

$$\frac{1}{F_m^{(\tau)}(z)} = \frac{(1 - z^{-1})^m}{\prod_{q=1}^m \left[\left(1 - \frac{X_{m,q}}{\alpha}\right) - \left(1 + \frac{X_{m,q}}{\alpha}\right) z^{-1}\right]}. \tag{4.43}$$

More generally, a near field control filter H_m is formed by the ratio of two version of (4.43) with different implicit parameters τ and τ'.

Second and first order sections. Any m_{th} order IIR filter can be implemented under the *Direct Form II* (4.44), with $m/2$ second order sections (or "cells") for even m, or $(m - 1)/2$ second order sections plus a first order one for odd m:

$$\begin{aligned} H_m(z) &= \prod_{q=1}^{m/2} \frac{b_0^q + b_1^q z^{-1} + b_2^q z^{-2}}{a_0^q + a_1^q z^{-1} + a_2^q z^{-2}} \times \frac{b_0^{\frac{m+1}{2}} + b_1^{\frac{m+1}{2}} z^{-1}}{a_0^{\frac{m+1}{2}} + a_1^{\frac{m+1}{2}} z^{-1}} \\ &= g \prod_{q=1}^{m/2} \frac{b'^q_0 + b'^q_1 z^{-1} + b'^q_2 z^{-2}}{a'^q_0 + a'^q_1 z^{-1} + a'^q_2 z^{-2}} \times \frac{b'^{\frac{m+1}{2}}_0 + b'^{\frac{m+1}{2}}_1 z^{-1}}{a'^{\frac{m+1}{2}}_0 + a'^{\frac{m+1}{2}}_1 z^{-1}}, \end{aligned} \tag{4.44}$$

the right factor (first order cell) being present only for odd orders m – when the product goes from $q = 1$ to $(m-1)/2$.

In order to define the NFC filter coefficients, first consider the denominator of (4.44) as related to the “near field compensation” part: it equals the denominator of (4.43), with $\tau = R/c$ as an implicit parameter. Each second order cell denominator $a_0^q + a_1^q z^{-1} + a_2^q z^{-2}$ derives from the 1st order cells of (4.43) that involve conjugate complex roots $X_{m,q}$ and $X_{m,m-q+1} = X^*_{m,q}$:

$$\begin{aligned} a_0^q &= 1 - 2\frac{\mathrm{Re}(X_{m,q})}{\alpha} + \frac{|X_{m,q}|^2}{\alpha^2}, \\ a_1^q &= -2\left(1 - \frac{|X_{m,q}|^2}{\alpha^2}\right), \qquad \text{for} \quad 1 \le q \le m/2 \text{ or } (m-1)/2 \\ a_2^q &= 1 + 2\frac{\mathrm{Re}(X_{m,q})}{\alpha} + \frac{|X_{m,q}|^2}{\alpha^2}. \end{aligned} \tag{4.45}$$

For odd order filters, the coefficients of the additional first order cell merely derive from the remaining real root $X_{m,(m+1)/2}$ as follows:

$$\begin{aligned} a_0^{\frac{m+1}{2}} &= 1 - \frac{X_{m,(m+1)/2}}{\alpha}, \\ a_1^{\frac{m+1}{2}} &= -\left(1 + \frac{X_{m,(m+1)/2}}{\alpha}\right). \end{aligned} \tag{4.46}$$

Numerator coefficients b_i^q, related to the “virtual source distance coding” part, are computed exactly the same way, but with $\tau = \rho/c$ as an implicit parameter instead of $\tau = R/c$. [6]

4.2 Virtual Studio Technology

4.2.1 VST plugins overview

The Virtual Studio Technology (VST), developed at Steinberg and first launched in 1996, allows the integration of virtual instruments and effect processors into every software or hardware audio environment supporting this protocol. They can be software reproductions of hardware effect units or instruments and even new creative effect components into a VST system. All are integrated seamlessly into the host application.

VST and similar technologies use Digital Signal Processing to implement software tools such as digital audio synthesizer or plugins. These tools can be employed to simulate traditional recording studio hardware. Plugins are generally run within a *Digital Audio Workstation* (*DAW*), enhancing the host application with additional functionality. Most of them can be classified as either instruments (VSTi) or effects, although there are other categories. VST plugins generally provide a custom GUI, displaying controls similar to the physical switches, knobs and faders of audio hardware. VST is supported

Figure 4.13: Virtual Studio Technology logo.

by a large number of audio applications: Cubase, Nuendo, Audacity, Reaper, etc. For this reason thousand of plugins exist, both commercial and freeware, for any kind of usage.

In computer music the term plugin (derived from *plugged in*) refers to an additional software component, to be used in applications for audio/video production, providing audio effects or generating new sounds. In practice it can be defined as a module containing a list of instructions for signal processing or signal generation. The audio process implemented by the plugin generally doesn't work autonomously, but needs an host application that feeds the audio input streams and receives the processed signals.[6] Thus, the host sees the plugin as a kind of "black box" with several inputs, outputs and associated parameters. Knowing nothing about the process carried out

[6]Today we can also find some examples of standalone plugins.

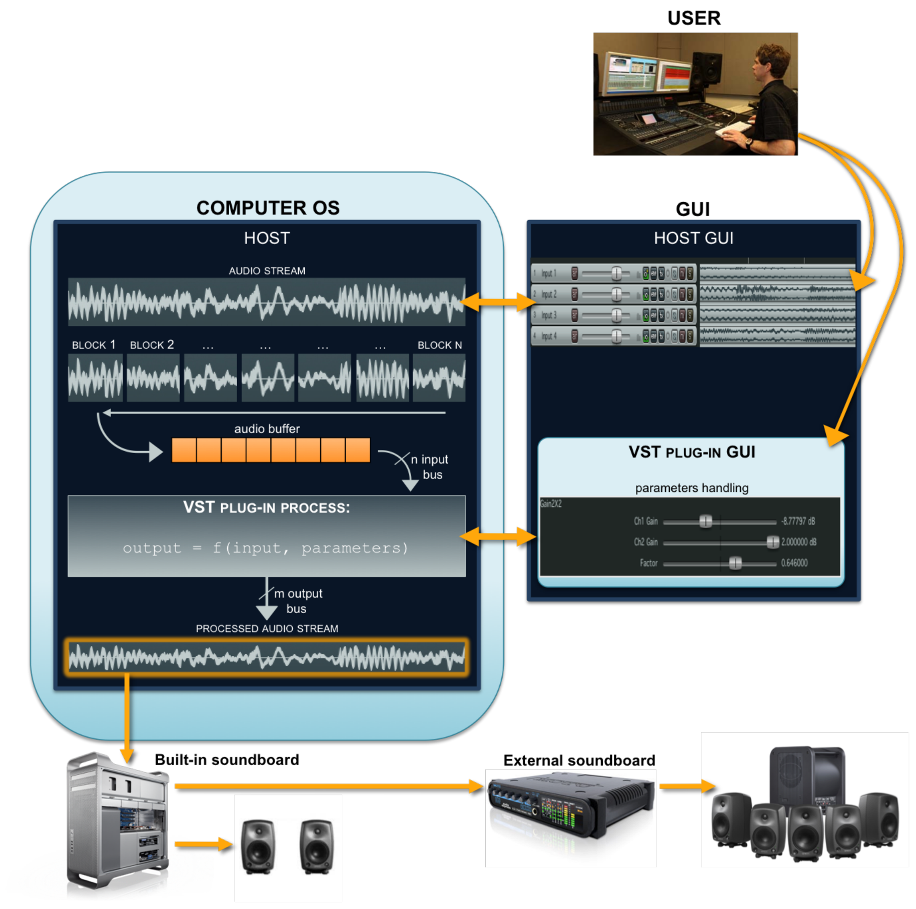


Figure 4.14: VST plugin working scheme.

by the plugin, the host is responsible only of its instantiation, destruction and routing of digital audio and MIDI to and from the plugin.

The *VST plugin standard* is the audio plugin standard created to allow any third party developers to create VST plugins. VST requires separate installations for Windows, Mac OS X and Linux. The majority of VST plugins are available for Windows having *.dll* file extension. But today most of them have a Mac OS X version with *.bundle* file extension.

A VST plugin performs its process using the computer processor, it does not necessarily needs dedicated digital signal processors. The host splits the audio streams into sequential blocks and the block size can be set by the user changing host settings. The VST-plugin maintains all parameters and status that refer to the running process: the host does not retain information about any data processed by the plugin as shown in figure 4.14.

4.2.2 Introduction to VST *Software Development Kit*

Steinberg provides a *Software Development Kit* (*SDK*), available on their site www.steinberg.net, that contains a set of C++ classes based on C API. In addition, Steinberg developed the VST GUI, another set of C++ classes, that can be employed to build a user graphical interface, with buttons, sliders, displays etc. These are low level C++ classes and the look and feel still have to be created by the plugin manufacturer.

VST SDK base classes are `AudioEffectX` and `AudioEffect`. The latter is the base class, which represent VST 1.0 plugins, and has its declarations in `audioeffect.hpp`, the implementation in `audioeffect.cpp` and structure definitions in `aeffect.h`. The files `aeffectx.h` (more structures), `audioeffectx.h` and `audioeffectx.cpp` are similar to the ones above, and extend them to the 2.0 version specifications (see figure 4.15 for SDK architecture). `AudioEffectX` is inherited by the plugin class created by the

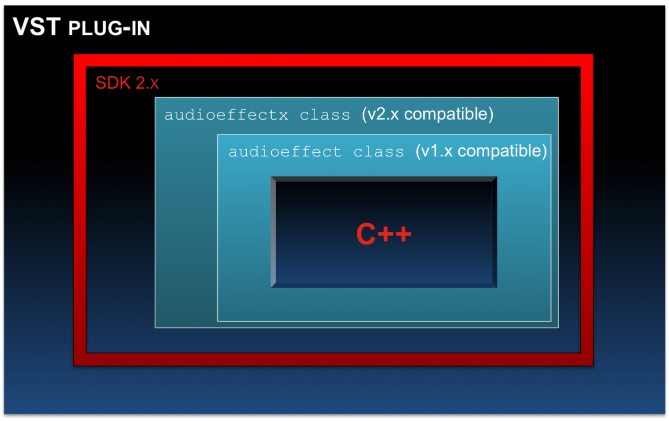

VST PLUG-IN
SDK 2.x
audioeffectx class (v2.x compatible)
audioeffect class (v1.x compatible)
C++

Figure 4.15: VST plugin working scheme.

programmer , and contains several methods that can be divided mainly into two categories:

- methods for plugin parameters handling: setting up, initialization, values getting, displaying, tagging and mapping;
- methods for signal processing (relative to effect plugins) – essentially `processRepalcing()` and `processDoubleReplacing()`, for hosts that support 64-bit floating-point numbers;

The latter two form the real plugin core, where the signal processing takes place. They take input data, apply the processing algorithm and then overwrite the output buffer. The host provides both input and output buffers for these methods.

Basic files required for a VST plugin implementation are a `.hpp` file, containing the plugin base class declaration, and a `.cpp` file containing the plugin class implementation, where the behaviour of parameters handling methods, processing methods and plugin variables are defined. One can use an additional `.cpp` file that contains the method called by the host to create a plugin instance otherwise defined into the just mentioned `.cpp` file.

4.3 Reaper DAW

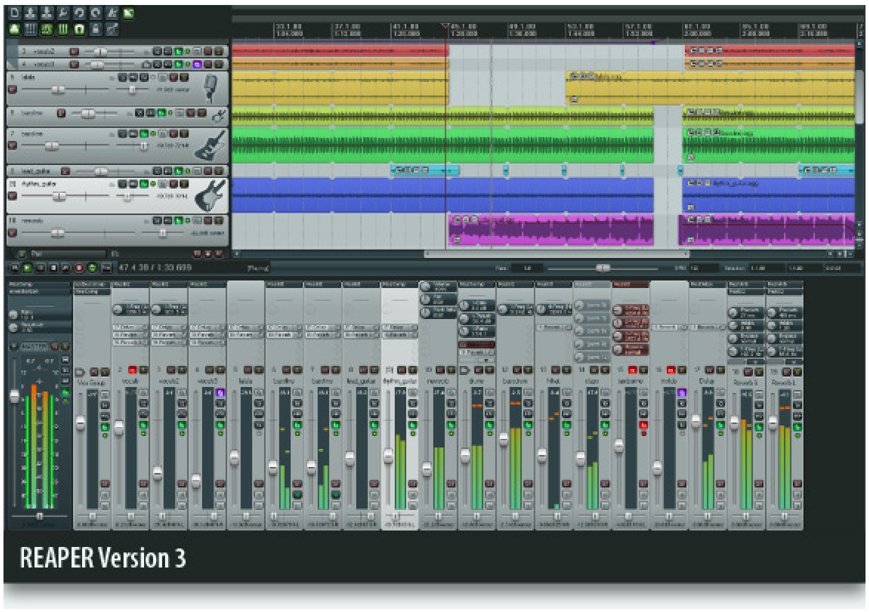


Figure 4.16: An example of Reaper working environment appearance.

REAPER (*Rapid Environment for Audio Production, Engineering, and Recording*) is a software for recording, editing, mixing, mastering and outputting audio, known as a digital audio workstation, created by Cockos. It is distributed with an uncrippled evaluation license. It is currently available for Microsoft Windows and Mac OS X. Reaper, like all DAW software, is similar in function to a digital multi-track tape recorder, digital mixing desk and effects. It implements additional features that are only possible because the software processes an audio stream before it is needed so as to reduce CPU peak loads.

4.3.1 Reaper main features and layouts

This section lists the main features that Reaper has in common with today's most popular DAWs.

Reaper supports ASIO, Kernel Streaming, WaveOut (MME) and DirectSound (WDM) for playback and recording; it is able to read and record WAV, OGG, AIFF, Wavpack, FLAC, APE, MP3 (with the lame encoder) and MIDI files, as well as many other formats. The reading and playback of the most used video formats, such as AVI, MPG, MOV, is also allowed. Reaper lets the user arrange any number of items (volume, pan controls and envelopes) in any number of tracks, limited only by the performance of the user's hardware rather than the software. An easily understandable and very usable graphic interface is provided for the user and also customizable. Audio and MIDI items (clips) can be mixed within the same track and hardware effects integration is permitted.

The company's own plug-in and FX scripting API, called Jesusonic (JS), are integrated within the DAW. JS effects are text files which, when interpreted and loaded by the DAW, function as plugins. Reaper can function both as a ReWire[7] slave and host, supporting real-time audio processing plugins with automation in addition to JS ones. The following plugin APIs are supported: VST and VSTi, DX and DXi (Windows only), AU (OS X only). As many of the most used DAWs, Reaper provides also a basic user interface for plugins that haven't one implemented, handling the plugin adjustable parameters with objects that reflect the DAW's graphic style. Is is also allowed, for plugins with their own GUI, to switch between the latter visualization and the one provided by the DAW.

As any other DAW, Reaper audio manipulations can be carried out interacting with basic working layouts.

Editing layout. For each audio stream, or "item", reaper displays an amplitude time graph on which the user can manipulate the signal with actions like split, delete, copy or move, choosing the type of manipulation action from the "Edit" menu or using keyboard short-cuts. In Reaper the editing is non-destructive. Edit is unique per item and do not alter the content of the source file. So original recorded files are safe from any modification. Reaper edit layout also permit the user to set fade-in and fade-out

[7] *ReWire* is a software protocol, jointly developed by Propellerhead and Steinberg, allowing remote control and data transfer among digital audio editing and related software. Originally appearing in the ReBirth software synthesizer in 1998, the protocol has evolved into an industry standard.

effects acting directly on the signal graph and choosing the fade curvature between six different ones (see figure 4.17). One can also increase or decrease the amplitude of the whole item by simply dragging the mouse on the item graph. All items parameters can be automated "drawing" its values time trajectory on the specific parameter associated graph.

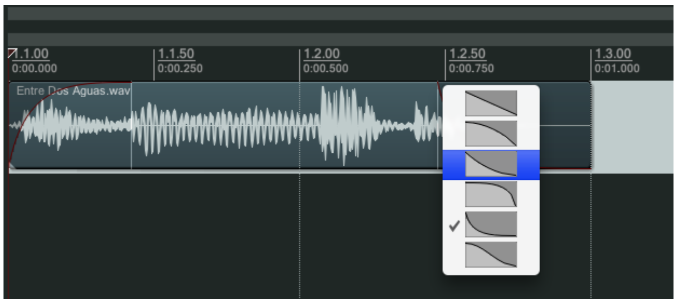


Figure 4.17: Reaper typical time graph in edit layout where the user is going to change fade-out envelope shape.

Mixing layout. This one consist in a series of graphical objects such as knobs, sliders, buttons, Vu-meters that one can use to implement some basic actions. Each item has its proper channel-strip that contain all the needed objects to perform any traditional mixing action on it. The main controls are for muting the item, soloing it, change its volume, its panning, enabling the recording. The button "Env", available in each item panel, shows the list of parameters that can be chosen to be automated.

Mixing panels can be placed both on the top part of the layout, called "Track list", and on the bottom one, called "Mixer", where the default view shows also the mixing controls for the "Master channels", so called in analogy with real mixers (for more details see figure 4.18).

Reaper allow user to customize the mixing layout deciding where to place each track panel and even to hide it. It also possible to select some controls to be visible or not by choosing them from an available "Mixer options" list. For example it can be chosen to visualize in every item panel the names of applied effects together (or not) with effect parameters controllers. Thus different mixer layouts can be set by the user. Some of them are shown in figure 4.19.

4.3.2 Additional features compared with other DAWs

The main differences between Reaper and other DAWs, such as Nuendo and Cubase, are audio routing special capabilities. In Reaper each item

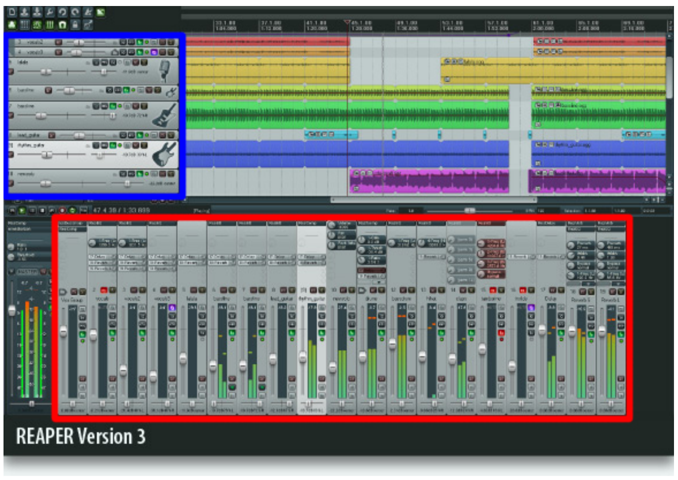
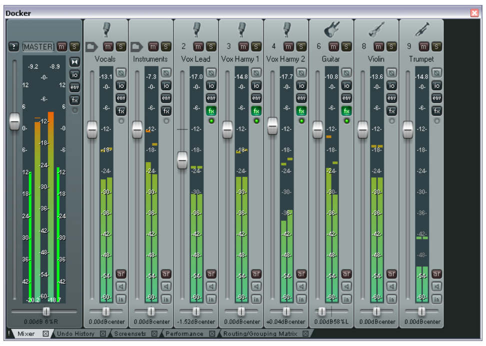


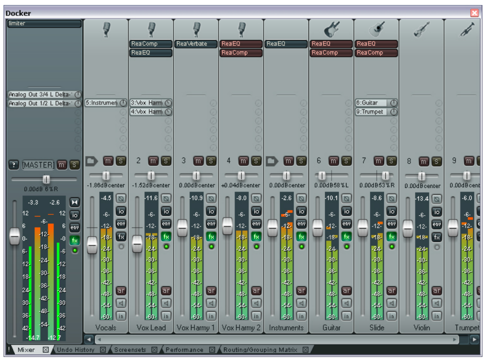
Figure 4.18: Reaper Track List (in the blue rectangle) and Mixer (in red rectangle).

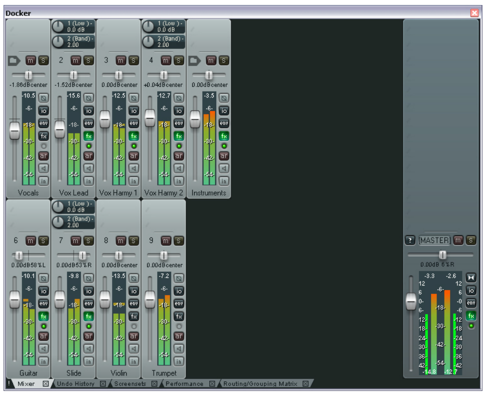
can receive several items as inputs and send its output to each item in the project. This action can be implemented using the "IO" button present in each item mixing panel (see figure 4.20). Clicking that button causes the routing for that track to be displayed in a "Track Routing window" shown in figure 4.21. The exact contents of an item Track Routing window will vary according to the project structure and DAW's hardware set-up (e.g. sound card and audio devices, MIDI devices etc.). Therefore, the window display should be similar to that shown in figure 4.21, but not necessarily identical. Notice in particular:

- Master/Parent Send – Enabling this ensures that the track's output will be directed to the Master.
- Audio Hardware Outputs – In addition to (or instead of) directing output to the Master Bus, it is possible also to direct output of any track directly to Hardware Outputs on the user audio device. If audio device has multiple outputs, this can be useful, for example, for creating a separate headphone mix, or as series of separate headphone mixes.
- MIDI Hardware Output – Useful to direct MIDI output to an external device or to the Microsoft GS Wavetable Synth, or any other synthesizer.
- Receives – Enabling project tracks output to be routed via a "Receive" to any other.

Figure 4.19: 3 Different mixer layouts


(a) Mixer 1


(b) Mixer 2


(c) Mixer 3

- Sends – Enabling output to be routed via a "Send" from any track to any others.

  When a send/receive is created, the user is are automatically presented

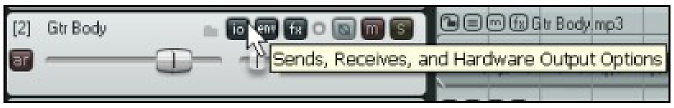
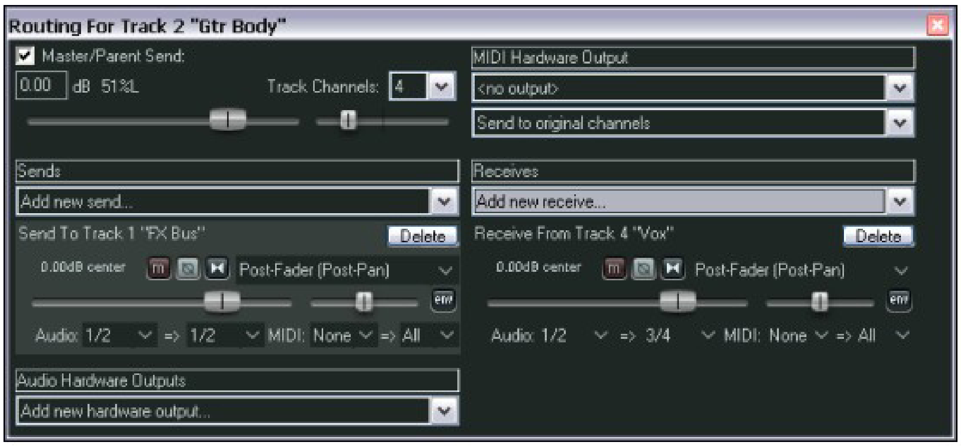


Figure 4.20: Reaper control panel in a track list where the user is going to click on IO button.


Routing For Track 2 "Gtr Body"

Master/Parent Send: ✓
0.00 dB 51%L
Track Channels: 4

MIDI Hardware Output
<no output>
Send to original channels

Sends
Add new send...
Send To Track 1 "FX Bus" Delete
0.00dB center Post-Fader (Post-Pan)
Audio: 1/2 => 1/2 MIDI: None => All

Receives
Add new receive...
Receive From Track 4 "Vox" Delete
0.00dB center Post-Fader (Post-Pan)
Audio: 1/2 => 3/4 MIDI: None => All

Audio Hardware Outputs
Add new hardware output...

Figure 4.21: Track Routing window.

with volume and pan faders which can be used to control this. It can be also specified whether to send/receive audio output, MIDI output, or both.

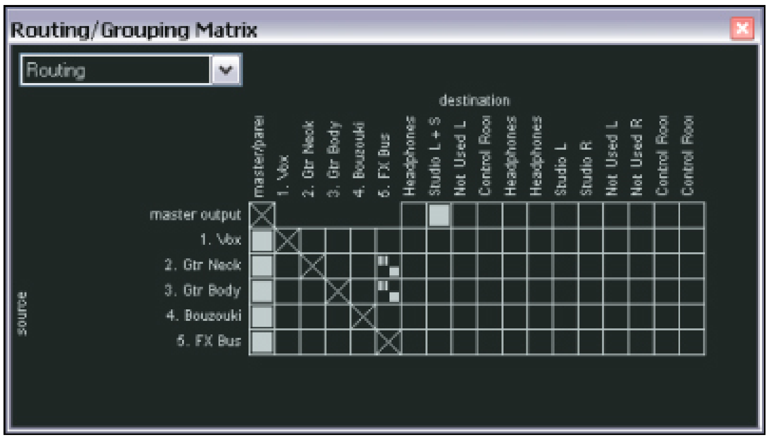
Another interesting feature is the "Routing Matrix" (see figure 4.22(a)), a window that display a summary table of the overall project's track routing. The user can act directly on this table to change tracks direction and faders, for example, send levels and panning by just right clicking on the specific routing relation cell.

4.3.3 Reaper relevant features for *AmbiSound-Spatializer* project

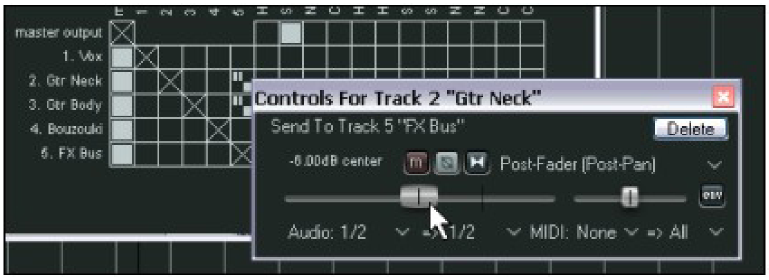
The above mentioned features have fixed Reaper as the appropriate DAW to be used as host for AmbiSound plugin. In fact, it has been implemented to take advantage of Reaper advanced routing capabilities, especially those regarding plugins send/receive options.

Reaper is up to now the only DAW able to see explicitly all inputs and outputs of multichannel plugins and to handle easily the related routing set-up. After the user has added a plugin to a project item (or items group), he can route the track different inputs to each plugin input channel by using

Figure 4.22: Routing Matrix



(a) Routing Matrix window



Controls For Track 2 "Gtr Neck"

Send To Track 5 "FX Bus"

-6.00dB center

Post-Fader (Post-Pan)

Audio: 1/2 => 1/2

MIDI: None => All

(b) Track routing controls

the plugin "Pin Connector" window shown in figure 4.23. The same tool allows also the routing of plugin output channels to the item outputs. Hence, Reaper supports plugins that can accept (and handle) more than one track as input, providing different output streams for each plugin output channel. This means that one can process N tracks to create multiple channel signal feeds using a single plugin, decreasing the processor workload. The mentioned action reflects perfectly the concept of audio mixing denoting Reaper the most suitable host for AmbiSound plugin.

To complete this section, some application examples are presented showing how to use Reaper introduced capabilities to implement actions connected to the 3D mixing employing AmbiSound plugin. The next two examples extracted from [23] illustrate in details how to route a multichannel track outputs to a multichannel plugin for signal bands splitting and how to implement a surround mix in 5.1 format.

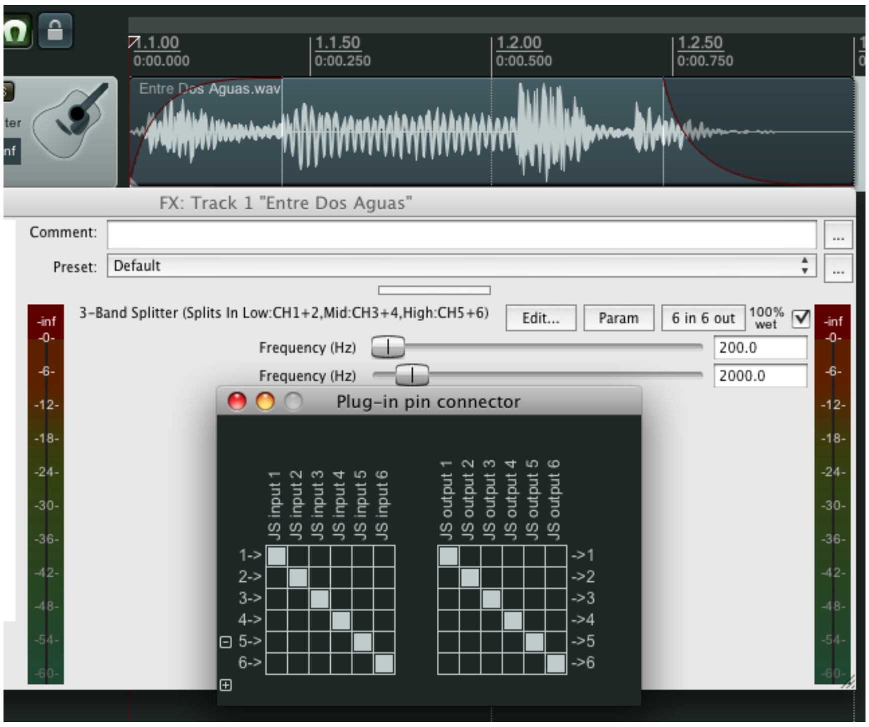


Figure 4.23: Plugin Pin Connector window.

Splitting Channels.

1. Create a Reaper project, insert a new audio track and save it with .RRP extension.

2. Create other three tracks – calling them Ghost Tracks – to mirror each of the three frequency bands the first inserted track is going to be split into. Create these three tracks and label them Low, Medium, and High. See illustration in figure 4.24.

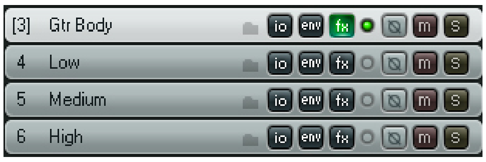
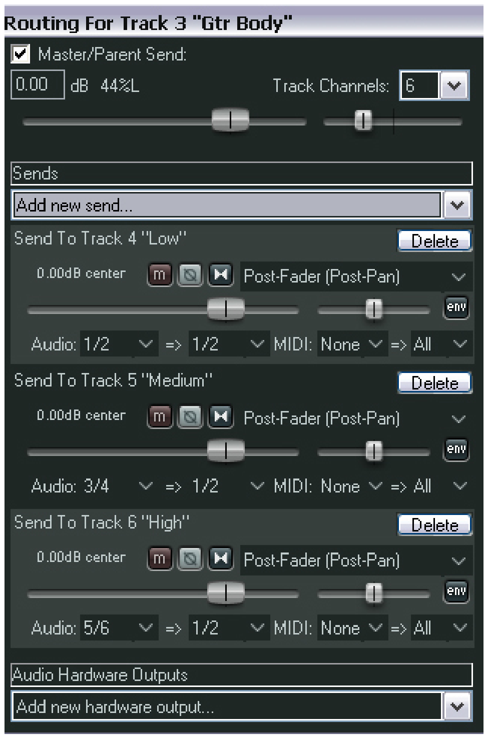


Figure 4.24: Example of track list for Splitting Channels application.

3. Display the IO Window for the first track. Set the number of Channels (near the top) to 6, and create sends to each of the just created three tracks.

4. Send Audio from Channels 1/2 to the Low track, Channels 3/4 to the Medium track, and Channels 5/6 to the High track. (see figure 4.25).

5. Now open the FX window for the first track and insert the JS LOSER/3-BandSplitter (a Reaper plugin).


Routing For Track 3 "Gtr Body"

☑ Master/Parent Send:

0.00 dB 44%L Track Channels: 6

Sends

Add new send...

Send To Track 4 "Low" Delete

0.00dB center m Post-Fader (Post-Pan)

Audio: 1/2 => 1/2 MIDI: None => All

Send To Track 5 "Medium" Delete

0.00dB center m Post-Fader (Post-Pan)

Audio: 3/4 => 1/2 MIDI: None => All

Send To Track 6 "High" Delete

0.00dB center m Post-Fader (Post-Pan)

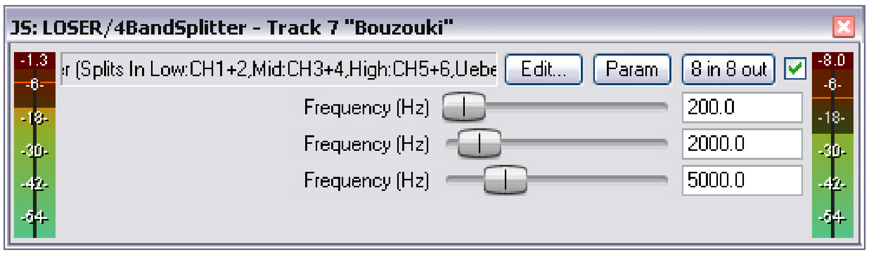
Audio: 5/6 => 1/2 MIDI: None => All

Audio Hardware Outputs

Add new hardware output...

Figure 4.25: Split Channels application send/receive assignments.

6. Set the first frequency fader to about 200 Hx and the second to about 2000 Hz (see figure 4.26).


JS: LOSER/4BandSplitter - Track 7 "Bouzouki"

r (Splits In Low:CH1+2,Mid:CH3+4,High:CH5+6,Uebe Edit... Param 8 in 8 out ☑

Frequency (Hz) 200.0

Frequency (Hz) 2000.0

Frequency (Hz) 5000.0

Figure 4.26: Plugin set-up in Splitting Channels application.

7. Solo the Low track and play. Notice that only the Low frequencies are heard. Repeat the operation also for Medium and High track.

8. Finally experiment adjusting the faders in the 3 Band Splitter and playing back, until a satisfied band mixing isn't reached.

Surround Mixing. The diagram in figure 4.27 illustrates one way that one could set up a project template ready for Surround Mixing, in this case using 5.1 showed in figure 4.28. This diagram represents the following set-up:

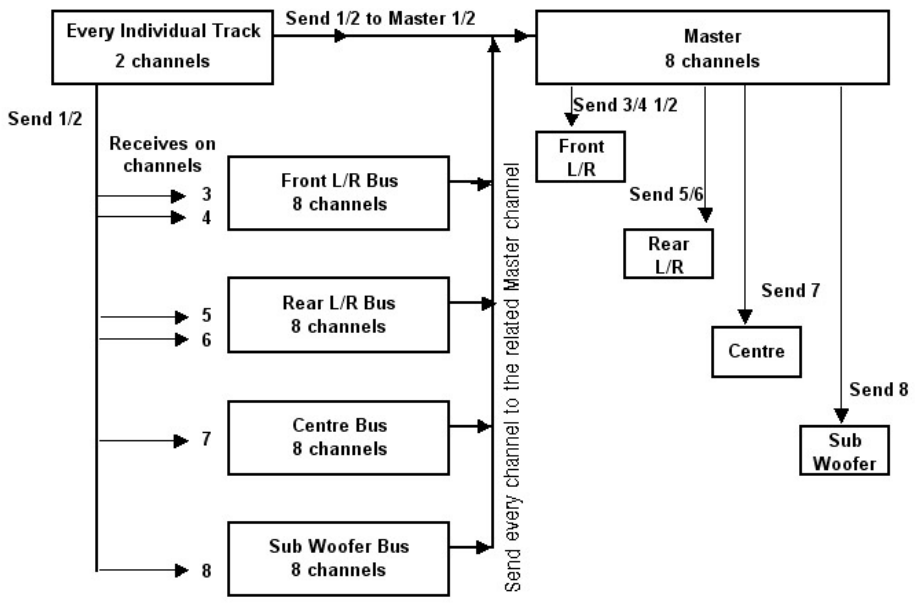


Figure 4.27: Surround Mixing diagram.

- Four special tracks have been set up to act as Surround Busses. Each of these tracks is defined as having 8 channels.
- Sends are established from every other track (or perhaps folder) to each of these busses. All receives to the Front L/R bus are thru channels 3/4, all receives to the Rear L/R bus are thru channels 5/6, and so on.
- All tracks and the four special busses have their output directed to the Master Track.
- Assuming a sound-card with sufficient audio outputs, all six speakers are physically connected to each one of six audio outputs.

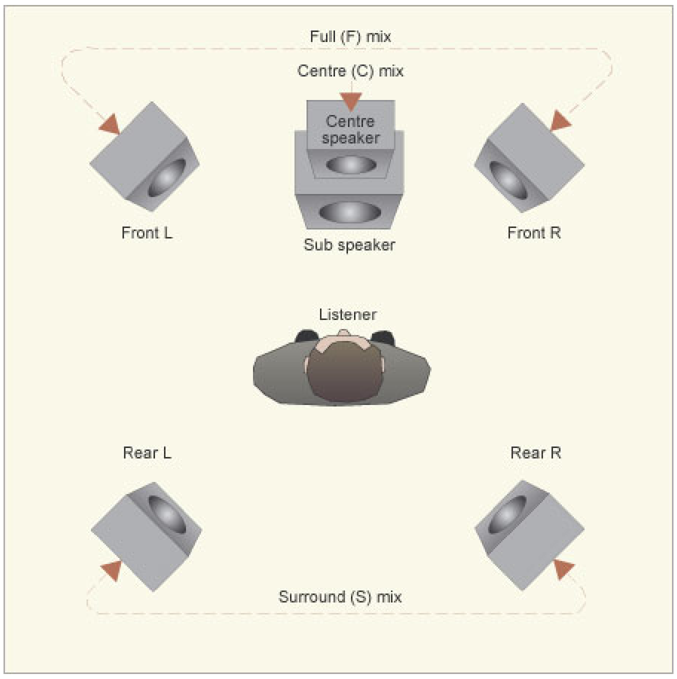


Figure 4.28: 5.1 format scheme.

- The Master is set up as an eight channel Master, with outputs routed to the various speakers as shown right.
- Mixes for the various output can be controlled using the Routing windows of the four special bus tracks.
- Relative overall output levels to each and all of the speakers can be controlled from the Master Track's output window, shown in figure 4.29.
- FX and plug-ins within the Master Track can now be assigned to any or all of various hardware outputs.
- A separate stereo mix can be still created, if wished, in this example using Channels 1 and 2 for this purpose.

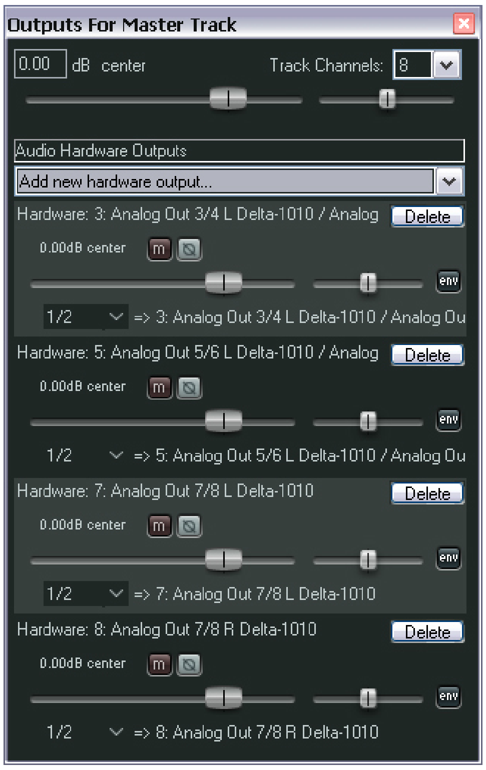

Outputs For Master Track
0.00 dB center
Track Channels: 8
Audio Hardware Outputs
Add new hardware output...
Hardware: 3: Analog Out 3/4 L Delta-1010 / Analog
Delete
0.00dB center
m
env
1/2
=> 3: Analog Out 3/4 L Delta-1010 / Analog Ou
Hardware: 5: Analog Out 5/6 L Delta-1010 / Analog
Delete
0.00dB center
m
env
1/2
=> 5: Analog Out 5/6 L Delta-1010 / Analog Ou
Hardware: 7: Analog Out 7/8 L Delta-1010
Delete
0.00dB center
m
env
1/2
=> 7: Analog Out 7/8 L Delta-1010
Hardware: 8: Analog Out 7/8 R Delta-1010
Delete
0.00dB center
m
env
1/2
=> 8: Analog Out 7/8 R Delta-1010

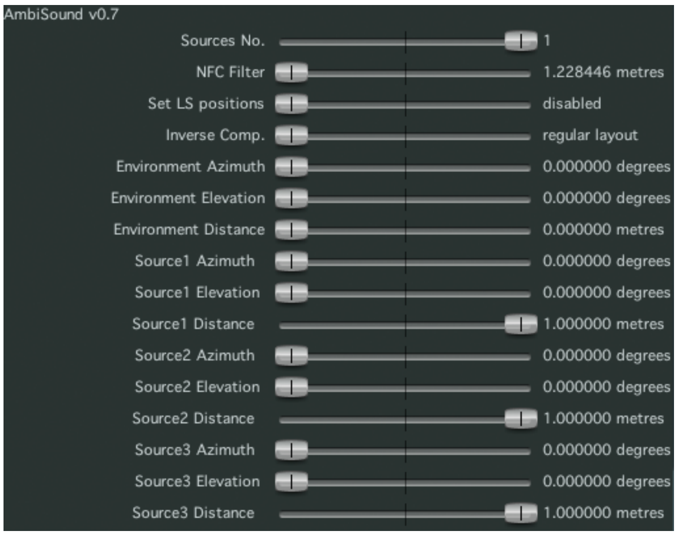
Figure 4.29: Surround Mixing application send/receive window.

Chapter 5

System Architecture

To implement 3D audio mixing, once a DAW supporting VST plugins (preferably Reaper) has been properly set, one can make the host instantiate the AmbiSound-Spatializer simply selecting it from the plugin menu. Then a window containing all plugin controls is displayed as GUI.

AmbiSound control window will appear, apart from the layout graphical style decided by the specific host, as the one shown in figure 5.1.



AmbiSound v0.7

Sources No.	1
NFC Filter	1.228446 metres
Set LS positions	disabled
Inverse Comp.	regular layout
Environment Azimuth	0.000000 degrees
Environment Elevation	0.000000 degrees
Environment Distance	0.000000 metres
Source1 Azimuth	0.000000 degrees
Source1 Elevation	0.000000 degrees
Source1 Distance	1.000000 metres
Source2 Azimuth	0.000000 degrees
Source2 Elevation	0.000000 degrees
Source2 Distance	1.000000 metres
Source3 Azimuth	0.000000 degrees
Source3 Elevation	0.000000 degrees
Source3 Distance	1.000000 metres

Figure 5.1: AmbiSound - Spatializer GUI provided by Reaper.

The user is allowed to set the virtual position of each input source adjusting the related elevation, azimuth or distance slider. Then AmbiSound provides the 3D source spatialization using Ambisonics technique and generates N output signals to feed the N-loudspeakers reproduction system.

The following sections will illustrate the architecture of the developed software starting from an high level system overview. Then each macro-block will be described in details with its functions and interactions.

5.1 *AmbiSound-Spatializer*: top level view

Summing up what has been said in Sec. 4.2, every VST plugin can be identified with a process that runs inside an host application for audio treatment. Narrowing this definition to AmbiSound plugin the following system diagram can be introduced. Figure 5.2 shows an high-level scheme of

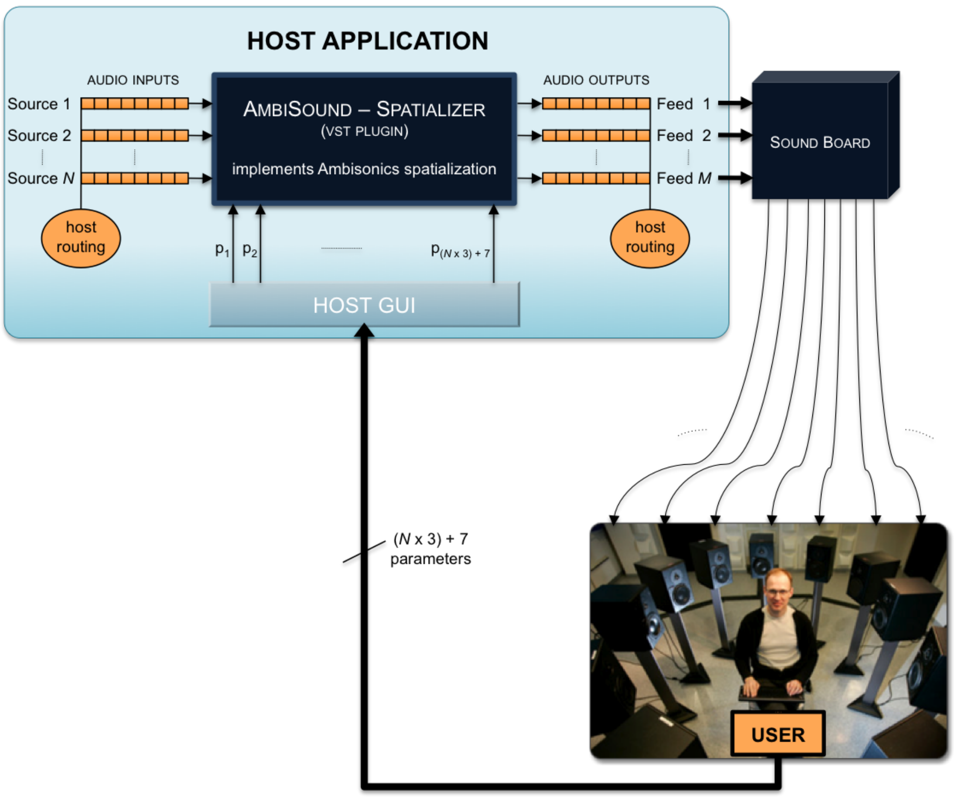


Figure 5.2: AmbiSound-Spatializer top level scheme.

AmbiSound-Spatializer and specifies its interaction with the host, the user and the soundboard (or reproduction system).

Up to 32 input sources can be routed toward each one of AmbiSound 32 inputs when the plugin is inserted into a DAW work session. Multiple input routing can be handled only if the host provides this kind of capability.

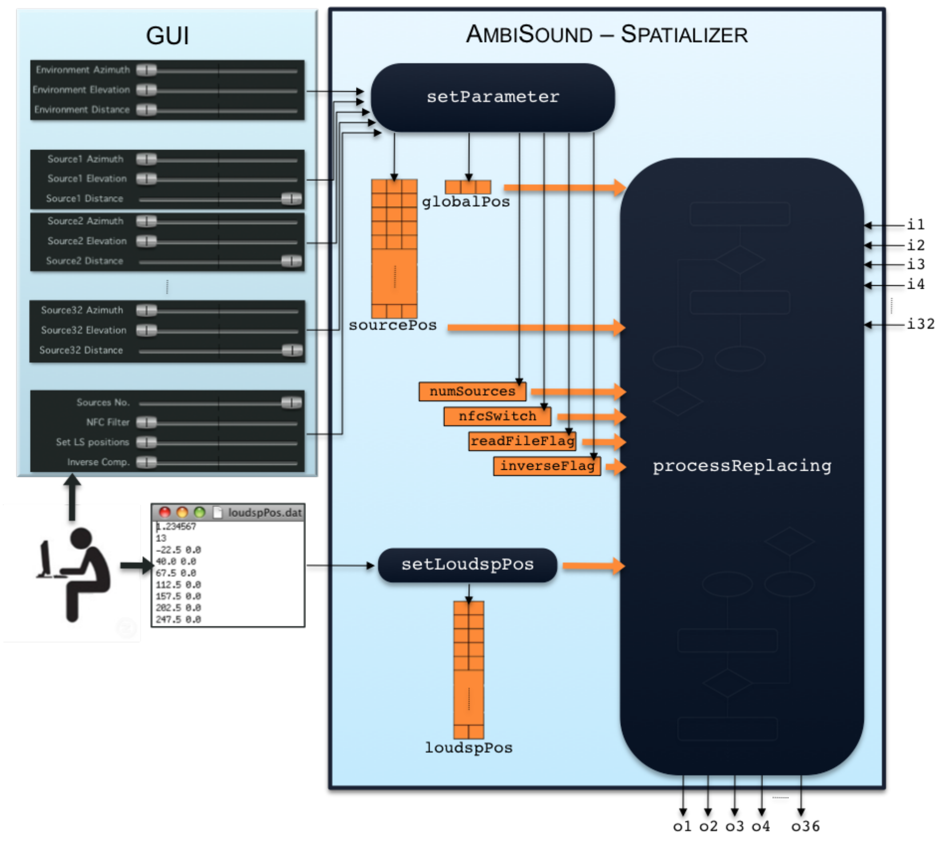
The user can control the sources virtual positioning in the surrounding

3D environment acting on AmbiSound-Spatializer GUI, setting the plugin parameters values such as: sources azimuth, elevation and distance from the listening point. Hence the plugin accepts as inputs also the parameters values for each sources positioning (up to $96 = 32 \times 3$) and other 7 values from other GUI controls that will be discussed on next section.

AmbiSound process generates up to 36 audio signals routed on 36 audio tracks respectively. These tracks are used to feed an Ambisonics 3D reproduction system employing the soundboard physical outputs, cabled with a maximum of 36 loudspeakers.

5.2 Plugin architecture overview

In this section AmbiSound-Spatializer architecture is introduced. The scheme reported in figure 5.3 shows the basic structure of the developed software.



```
1.234567
13
-22.5 0.0
40.0 0.0
67.5 0.0
112.5 0.0
157.5 0.0
202.5 0.0
247.5 0.0
```

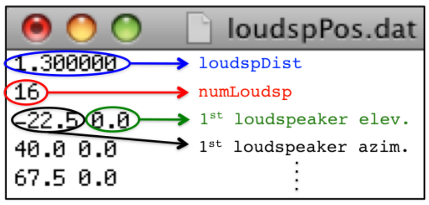
Figure 5.3: AmbiSound-Spatializer architecture.

AmbiSound is defined as a C++ class inherits `AudioEffectX` class from

VST SDK. All methods and variables declared in `ambiSound` class are involved in the plugin processing. `ambiSound` constructor (see the pseudo-code below) creates a plugin instance defining 32 inputs and 36 outputs to be processed (when activated by the user).

```
// ambiSound Class Constructor
//-----------------------------------------------------------------------
ambiSound::ambiSound():AudioEffectX
(audioMaster, kNumPrograms, kNumParameters) {
  setNumInputs(32);
  setNumOutputs(36);
  setUniqueID(kUniqueId);
  canProcessReplacing ();// supports replacing
                         // output
  canDoubleReplacing (); // supports double
                         // precision processing
  init();// does all plugin variables
         // inizialization
}
//-----------------------------------------------------------------------
```

Then the `init()` routine is recalled to load system default settings. Inside this method `setLoudspPos()` is used to read "`loudspPos.dat`" file (see figure 5.3 and 5.4) created by the user. This file specifies the reproduction system layout: loudspeakers distance from the listening point, number (up to 36) and positions in terms of azimuth and elevation. Reading these data, the method `setLoudspPos()` creates the $N \times 2$ matrix `loudspPos` to store loudspeakers position values inside it.



```
1.300000
16
22.5 0.0
40.0 0.0
67.5 0.0
```

Figure 5.4: "loudspPos.dat" file content example.

Other two basic routines, already declared in `AudioEffectX` class, are `setParameter()` and `processReplacing()` (or `processDoubleReplacing()` for host that supports 64-bit precision in computations). These methods must be overwritten in `ambiSound` class to implement plugin specific actions (see figure 5.3).

`setParameter()` and `processReplacing()` are run by the host on two separate threads. The first takes as inputs the index of the parameter to be set and the related value. So an integer index must be assigned to each plugin parameter to be identified. AmbiSound parameters handled by `setParameter()` are:

- azimuth, elevation and distance for each one of the 32 input sources;
- azimuth, elevation and distance of the entire system, to be used for global rotation and distancing actions;
- sources number, used to select the number of active sources, i.e. the ones really processed by the plugin;
- NFC filtering switch, used to enable/disable NFC function and to set the reproduction system distance which filter impulse response is computed on;
- "`loudspPos.dat`" file reading enabler;
- An indicator of the mathematical technique employed to invert **C** matrix.[1]

The host recalls `setParameter()` every time the user change one of these parameters value using GUI controllers (sliders). The variable corresponding to the controller index is then set to the controller value, as shown in the pseudo code below.

```
// setParameter() routine
//--------------------------------------------------------------------
ambiSound::setParameter (index, value){
  switch(index){
    case 0 : numSources = int(value);
    break;
    case 1 : nfcSwitch = int(value);
      break;
    case 2 : readFileFlag = int(value);
      break;
    case 3 : inverseFlag = int(value);
      break;
    case 4 : 5 : 6 :
      globalPos[index-2] = value;
      break;
    default :
      vectorIndex = (index - 7)/3;  // find what source has to be set
      positionIndex = (index - 7)%3;// find what source parameter has
                                    // to be set
      sourcePos[vectorIndex][positionIndex] = value;
      break;
  }
}
//--------------------------------------------------------------------
```

[1]Matrix of loudspeakers spherical harmonic components.

Thus `setParameter()` updates values stored in the $K \times 3$ matrix `sourcesPos` that contains the virtual position values for each one of the activated K-sources. `setParameter()` updates also `globalPos` vector that contains the three values of the entire system configuration and the integer variable `numSources`; `nfcSwitch`; `readFileFlag` and `inverseFlag` whose task will be shown later on.

All these variables matrices and vectors are created and initialized with default values by `init()` method (recalled in `ambiSound` constructor) and together with the above mentioned `setLoudpsPos()` method are employed in `processReplacing()` routine.

The latter takes in input two pointers to audio buffer vectors for input and output samples frames respectively. After the entire process – involving the input audio buffers and others methods and variables shown in details on next sections – has been carried out, `processReplacing()` rewrites the output buffers. The convention, used for `processReplacing()` scheme in the block diagram that shows AmbiSound architecture in figure 5.3, is based on the analogy with a typical analogue system block diagram that has in-coming inputs and out-coming outputs. As already told `processReplacing()`, instead, takes as arguments both input and output buffers pointers overwriting the output buffer. Then the host will read the overwritten audio buffer.

5.3 Plugin process architecture

A more correct architecture for AmbiSound's `processReplacing()` routine is illustrated in figure 5.5.

All variable shown in the scheme top part are employed in same way by the routine to implement the plugin process - Ambisonics mixing. Apart from those already mentioned in last section, they are:

- `sourcesMatrix`, a $16 \times K$ matrix (denoted with $\mathbf{Y}$) that contains the spherical harmonics functions of all activated sources up to the 3^{rd} order ($M = 16$) that have to be computed before the output replacing process;

- `loudspMatrix`, a $N \times 16$ matrix ($\mathbf{C}^{\dagger}$ or $\mathbf{C}^{T}$) used by the plugin to store the Ambisonics 3^{rd} order decoding matrix, computed for the specific reproduction system before the output replacing process;

- `gainsMatrix`, a $N \times K$ matrix (denoted with $\mathbf{G}$) used to store the result of the matrix product $\mathbf{C}^{\dagger} \times \mathbf{Y}$. Hence the value stored into

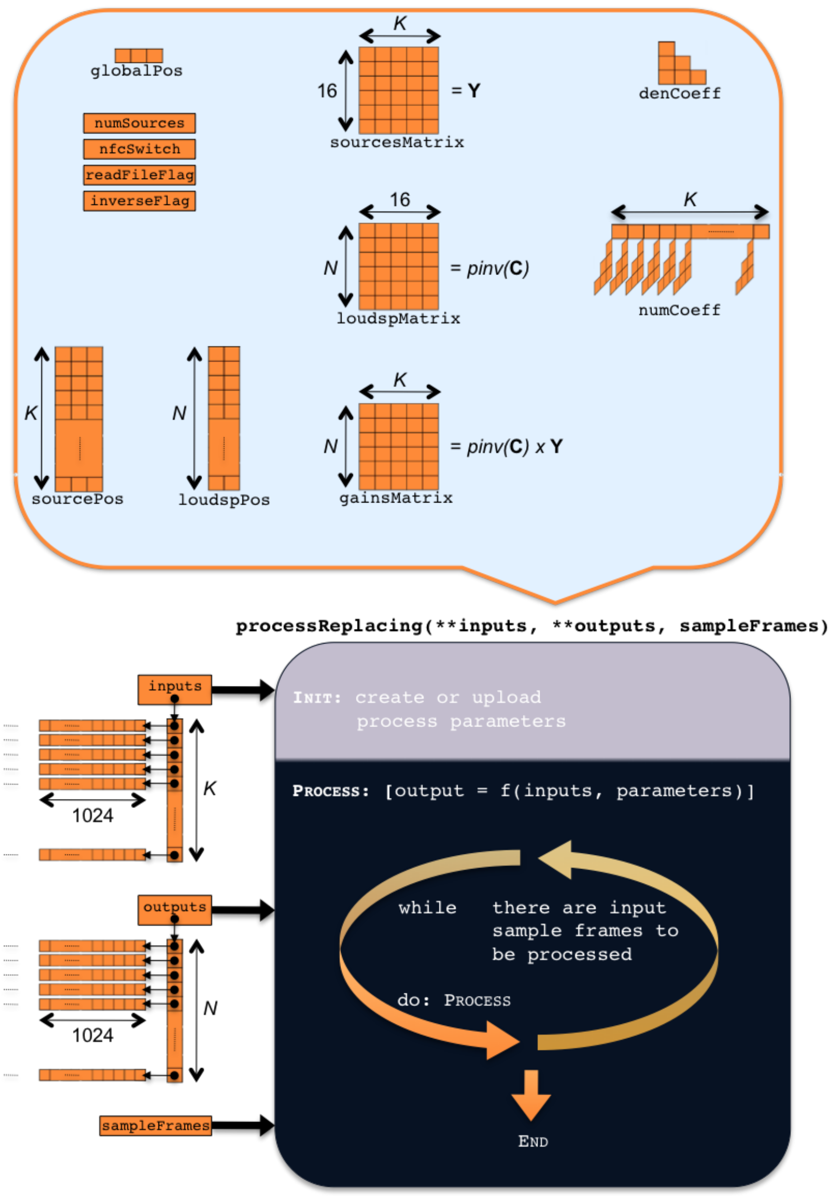


Figure 5.5: AmbiSound's `processReplacing()` method external scheme.

`gainsMatrix` *j-th* row and *i-th* column is obtained as the scalar product between $\mathbf{C}^{\dagger}$ *j-th* row and $\mathbf{Y}$ *i-th* column and represent the gain coefficient to be applied to the *i-th* input to determine its contribute to the *j-th* output signal generation. These gain coefficients are computed before the plugin output replacing process;

- `denCoeff`, a 3×3 lower triangular matrix (with non-existing zero cells) that contains the NFC filter denominator coefficients that are eventually computed and uploaded before the output replacing process;

- `numCoeff`, a 3D vector with K cells, each one containing the same structure as `denCoeff`, filled with the NFC filter numerator coefficients computed for each input sources. When the user change a source distance value, the related NFC filter numerator coefficients are computed – basing on the new distance value – before the output replacing process.

From what has been told since now, the `processReplacing()` method can be seen as divided into two subsections.

The first, before the actual output replacing process, initializes, creates, updates and checks all the conditions useful for the successive section.

The latter processes each input buffer filled with an audio samples frame. It writes the results into each output buffer and keeps on performing this operation until the frame end. This condition is checked after every replacing cycle is completed (output buffers are totally rewritten) using `sampleFrames` integer variable, which keeps track of how many samples in each frame are left to be processed.

The flow diagram in figure 5.6 shows what are the specific actions taken by `processReplacing()` in both "Init" and "Process" phase. The following paragraphs illustrate in details the methods introduced in figure 5.6 describing also their interaction.

`setLoudspPos().` Recalled if `loudspPos.dat` file reading is enabled by `readFileFlag`. This method, already discussed in Sec. 5.2, will update the layout system configuration resetting values in `loudspPos` matrix (see the pseudo-code below).

```
// setLoudspPos() routine
//----------------------------------------------------------------------
ambiSound::setLoudspPos (){
  open file = ``loudspPos.dat'';
  loudspDist = file[0];
  numLoudsp = file[1];

  for(i=0; i<=(numLoudsp); i++){
    loudspPos[i-th row]= i-th loudspeaker file data;
  }
  setLoudspMatrix();
}
//----------------------------------------------------------------------
```

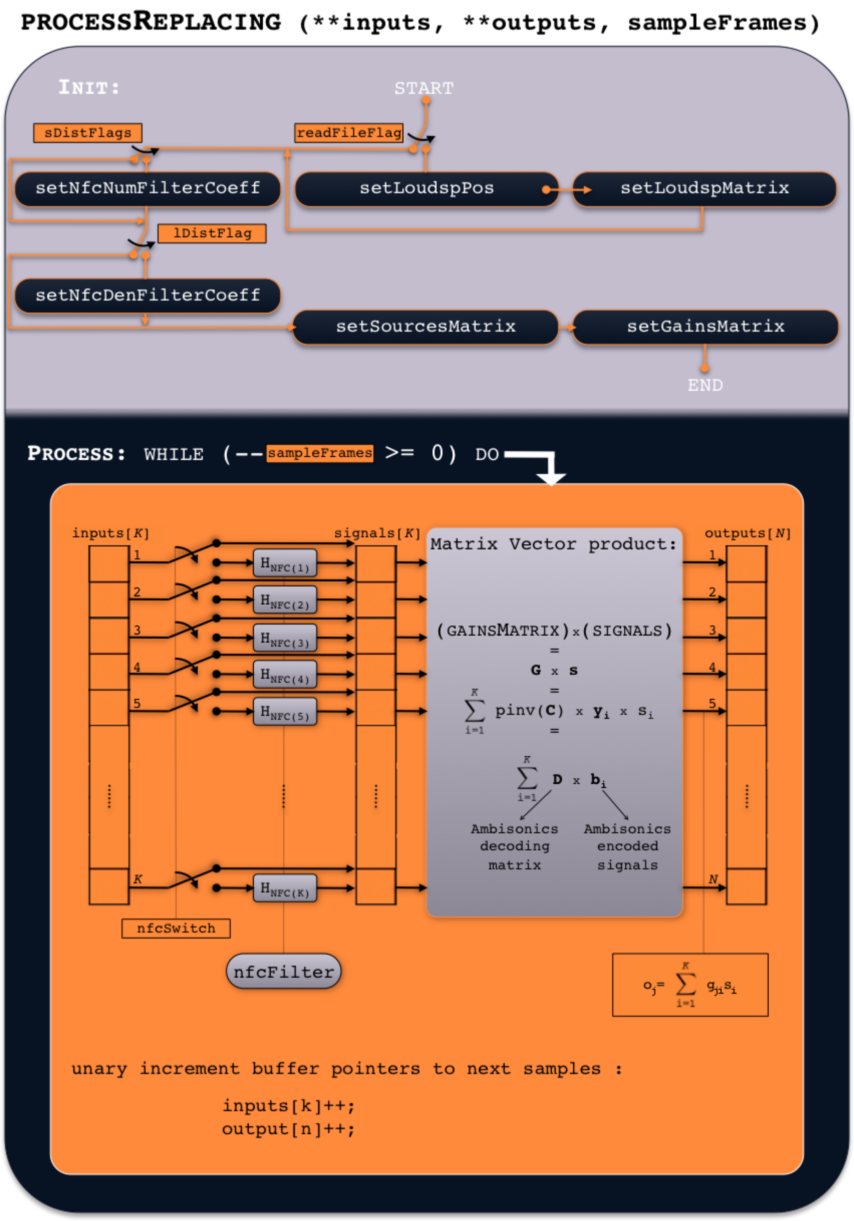

PROCESSREPLACING (**inputs, **outputs, sampleFrames)
INIT:
START
sDistFlags
readFileFlag
setNfcNumFilterCoeff
setLoudspPos
setLoudspMatrix
lDistFlag
setNfcDenFilterCoeff
setSourcesMatrix
setGainsMatrix
END
PROCESS: WHILE (--sampleFrames >= 0) DO
inputs[K]
signals[K]
outputs[N]
H_NFC(1)
H_NFC(2)
H_NFC(3)
H_NFC(4)
H_NFC(5)
H_NFC(K)
Matrix Vector product:
(GAINSMATRIX)x(SIGNALS) = G x s = Σ_{i=1}^{K} pinv(C) x y_i x s_i = Σ_{i=1}^{K} D x b_i
Ambisonics decoding matrix
Ambisonics encoded signals
nfcSwitch
nfcFilter
o_j = Σ_{i=1}^{K} g_ji s_i
unary increment buffer pointers to next samples :
inputs[k]++;
output[n]++;

Figure 5.6: AmbiSound's `processReplacing()` method internal scheme.

`setLoudspMatrix()`. Recalled in `setLoudspPos()`, computes the matrix $\mathbf{C}$ of loudspeakers spherical harmonic functions for the new system layout configuration. Then computes the Ambisonics decoding matrix $\mathbf{D}$ $(= \mathbf{C}^{\dagger})$ storing its values in `loudspMatrix`.

As shown in the pseudo-code below $\mathbf{D}$ computation technique can be selected by the user between the usual **C** pseudo-inverse or a simple **C** transposition. The first is implemented using the `pinv()` routine of *Armadillo* API[2] [24]. This one takes respectively the inverse, left-inverse, right-inverse of a matrix basing on its dimensions.

```
// setLoudspMatrix() routine
//-----------------------------------------------------------------------
ambiSound::setLoudspMatrix(){
  reset loudspMatrix;

  for(i=0; i<=(numLoudsp); i++){ //computation of trans(C)
    compute spharm = i-th loudspeaker sph. harmonic functions;
    loudspMatrix[i-th row] = spharm;
  }
  if(inverseFlag == 1){
    C = trans(loudspMatrix);
    loudspMatrix = pinv(C);
  }
}
//-----------------------------------------------------------------------
```

setNfcNumFilterCoeff() & setNfcDenFilterCoeff(). These methods are recalled when the user change respectively a source distance or the loudspeaker system distance. When a source distance is set into a new value, `setNfcNumFilterCoeff()`, that takes as arguments the source id and the new distance value, computes NFC filter numerator coefficients for the specific source and updates the related `numCoeff` vector cell (see the pseudocode below).

```
// NfcNumFilterCoeff() routine
//-----------------------------------------------------------------------
ambiSound::NfcNumFilterCoeff(sourceIndex, sourceDist){
  c = 340 m/s;
  fs = Host sample frequency;
  sourceDist += globalPos[2];
  alpha = (4*fs*sourceDist)/c;// factor introduced in NFC filter design
  coeffients[];
  i = sourceIndex;

  compute H1_bCoefficients(alpha);
  coeffients = H1_bCoefficients;
  numCoeff[sourceIndex][0] = coefficients;

  compute H2_bCoefficients(alpha);
  coeffients = H2_bCoefficients;
  numCoeff[sourceIndex][1] = coefficients;
```

[2] A C++ API for linear Algebra computations based on *Lapack* and *Blas* libraries.

```
  compute H3_bCoefficients(alpha);
  coeffients = H3_bCoefficients;
  numCoeff[sourceIndex][2] = coefficients;

}
//------------------------------------------------------------------------
```

Similarly as loudspeakers distance is changed `setNfcDenFilterCoeff().`, taking the new distance value as argument, provides computing and uploading new NFC filter denominator coefficients into `denCoeff` matrix as in the pseudo-code below.

```
// NfcDenFilterCoeff() routine
//------------------------------------------------------------------------
ambiSound::NfcDenFilterCoeff(distance){
  c = 340 m/s;
  fs = Host sample frequency;
  alpha = (4*fs*distance)/c;// factor introduced in NFC filter design
  coeffients[];

  compute H1_aCoefficients(alpha);
  coeffients = H1_aCoefficients;
  denCoeff[0] = coefficients;

  compute H2_aCoefficients(alpha);
  coeffients = H2_aCoefficients;
  denCoeff[1] = coefficients;

  compute H3_aCoefficients(alpha);
  coeffients = H3_aCoefficients;
  denCoeff[2] = coefficients;

}
//------------------------------------------------------------------------
```

The coefficients computational technique implemented in C++ within these two methods has been fully described in Sec. 4.1.3 and makes use of Table 4.3 values to obtain filter zeroes.

setSourcesMatrix(). Always recalled in `processReplacing()` before the ‘while’ cycle, it computes values to be stored in `sourcesMatrix`. Starting from each sources azimuth and elevation value stored in `sourcesPos` matrix, the spherical harmonic functions are computed according to the formulas introduced in Table 4.1. Then all **c** vectors (defined in Section 4.1.2) are stored as `sourcesMatrix` columns (see the following pseudo-code).

```
// setSourcesMatrix() routine
//-----------------------------------------------------------------------------
ambiSound::setSourcesMatrix(sources){
  reset sourcesMatrix;

  for(i=0; i<=(sources); i++){ //computation of Y
    // variables used in spharm computation
    azimuth = sourcesPos[i][0] + globalPos[0];
    elevation = sourcesPos[i][1] + globalPos[1];
    attenuation = 1/(sourcesPos[i][2] + globalPos[2]);

    compute spharm = i-th source sph. harmonic functions;
    sourcesMatrix[i-th col] = spharm;
}
//-----------------------------------------------------------------------------
```

From now on the vector **c** of the *i-th* source spherical harmonic functions will be denoted with $\mathbf{y_i}$ to avoid the confusion with those related to loudspeaker spherical harmonic function. Hence `sourcesMatrix` will be often denoted as matrix $\mathbf{Y}$.

`setGainsMatrix()`. Always recalled in `processReplacing()` before the 'while' cycle to implement the matrix product between `loudspMatrix` ($\mathbf{D} = \mathbf{C}^{\dagger}$) and `sourcesMatrix` ($\mathbf{Y}$). The values obtained are then stored into `gainsMatrix` ($\mathbf{G}$) as shown in the routine's pseudo-code below.

```
// setGainsMatrix() routine
//-----------------------------------------------------------------------------
ambiSound::setGainMatrix(sources, nfcFlag){
  reset gainsMatrix;
  row[];
  if(nfcFlag=0){
    for(j=0; j<=(numLoudsp); j++){// pinv(C) x Y
      for(i=0; i<=(sources); i++){// ``expanded'' scalar product
        compute g = ...
        ...expand loudspMatrix[j-th row]*sourcesMatrix[i-th col];
        row[i] = g;
      }
      gainsMatrix[j-th row] = row;
    }
  }
  else{
    for(j=0; j<=(numLoudsp); j++){// pinv(C) x Y
      for(i=0; i<=(sources); i++){// usual scalar product
        compute g=loudspMatrix[j-th row]*sourcesMatrix[i-th col];
        row[i] = g;
      }
      gainsMatrix[j-th row] = row;
    }
  }
}
//-----------------------------------------------------------------------------
```

The g_{ji} coefficient of `gainsMatrix` (placed at the *j-th* row *i-th* column) represents the gain factor to be applied on the *i-th* input signal to obtain its contribute to the *j-th* loudspeaker signal feed generation.

This implementation of Ambisonics technique differs a little from the one explained in Sec. 4.1.2 for the order signals encoding and decoding are taken. In fact Ambisonics theory suggests to keep the encoding operation ($\mathbf{y_i} s(t)_i = \mathbf{b_i}$) separated from the decoding one: $\mathbf{C}^\dagger \mathbf{b_i} = \mathbf{o_i}$, where $\mathbf{b_i}$ is the Ambisonics signals vector defined in Sec. 4.1.2 for the *i-th* input signal $s(t)_i$. This product gives the vector $\mathbf{o_i}$ whose components are the contribute of the *i-th* source to each one of the N system loudspeakers feeding signal. To determine the total loudspeaker feeds vector $\mathbf{o}$ (displayed as *outputs* in figure 5.6), the following sum has to be computed:

$$\mathbf{o} = \sum_{i=1}^{K} \mathbf{o_i}, \tag{5.1}$$

so that $\mathbf{o}$ *j-th* component is obtained summing up all the *j-th* components of vectors in (5.1) sum's argument.

Implementing this technique in AmbiSound `processReplacing()` routine requires to do the following steps into the 'while' cycle – which keeps busy the computer processor: for each source compute

1. Ambisonics signals $\mathbf{b_i} = \mathbf{y_i} \cdot s_i(t)$;
2. its outputs contribute $\mathbf{o_i} = \mathbf{C}^\dagger \mathbf{b_i}$;
3. the total output signals $\mathbf{o} \mathrel{+}= \mathbf{o_i}$.

Thus $(16 + K^2 \cdot N) \cdot K = 16K + K^3 N$ multiplications are computed in each cycle.

The actual technique employed in AmbiSound-Spatializer is more compact and permits to keep the most part of Ambisonics encoding and decoding operations before the 'while' cycle (output replacing) decreasing the processor usage. In fact, once $\mathbf{G} = \mathbf{C}^\dagger \times \mathbf{Y}$ is computed in `setGainsMatrix()`, the only operation left to be done into the 'while' cycle is the product $\mathbf{G} \times \mathbf{s}$ [3] which implies a computation of NK^2 multiplications.

The scheme in figure 5.7 compares the operational stages for the two different Ambisonics techniques.

What follows is the demonstration that these two technique are equivalent.

[3]$\mathbf{s}$ denotes the input signals vector

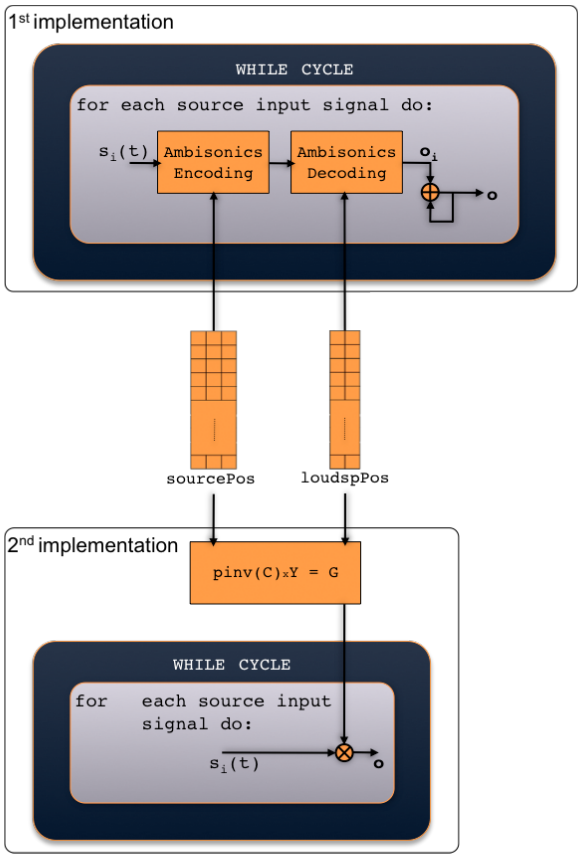


Figure 5.7: Two possible VST plugin implementations of Ambisonics encoding/decoding technique: the first, usual technique, requires more multiplications in the plugin 'while' cycle than the latter.

Starting from Ambisonics components of the *i-th* source signal, the encoding is obtained with:

$$\mathbf{b_i} = \mathbf{y_i} \cdot s_i(t), \tag{5.2}$$

where $\mathbf{y_i}$ is the vector of the spherical harmonic functions computed up to the 3^{rd} order basing on the virtual position set for the *i-th* source. The N-dim vectors $\mathbf{o_i}$ and $\mathbf{o}$, already defined in (5.1), are reported here for convenience:

$$\mathbf{o_i} = \mathbf{C}^{\dagger}\mathbf{b_i} = \begin{bmatrix} o_{1i} & o_{2i} & \cdots & o_{ji} & \cdots & o_{Ni} \end{bmatrix}^T, \tag{5.3}$$

$$\text{and} \qquad \mathbf{o} = \sum_{i=1}^{K} \mathbf{o_i}, \tag{5.4}$$

where K represents the number of active input signal.

Substituting $\mathbf{o_i}$ with (5.3) and using (5.2) the expression (5.4) can be written as

$$\begin{aligned}\mathbf{o} = \sum_{i=1}^{K} \mathbf{C}^{\dagger}\mathbf{b_i} &= \sum_{i=1}^{K} \mathbf{C}^{\dagger}(\mathbf{y_i} \cdot s_i(t)) \\ &= \mathbf{C}^{\dagger}(\mathbf{y_1} \cdot s_1(t)) + \mathbf{C}^{\dagger}(\mathbf{y_2} \cdot s_2(t)) + \cdots + \mathbf{C}^{\dagger}(\mathbf{y_K} \cdot s_K(t)).\end{aligned} \tag{5.5}$$

Now defining the *i-th* addend of (5.5) as the N-dim vector $\mathbf{g_i} = \mathbf{C}^{\dagger}\mathbf{y_i}$, expression (5.5) becomes

$$\mathbf{o} = \mathbf{g_1} \cdot s_1(t) + \mathbf{g_2} \cdot s_2(t) + \cdots + \mathbf{g_K} \cdot s_K(t),$$

that can be written in a more compact form using matrix formalism:

$$\mathbf{o} = \mathbf{G'}\mathbf{s}, \tag{5.6}$$

where $\mathbf{G'}$ columns are vectors $\mathbf{g_1}, \mathbf{g_2}, \ldots, \mathbf{g_K}$, and $\mathbf{s}$ components are input signals $\mathbf{s_1(t)}, \mathbf{s_2(t)}, \ldots, \mathbf{s_K(t)}$.

Having defined $\mathbf{C}^{\dagger}$ as the Ambisonics decoding matrix $\mathbf{D}$ (`loudspMatrix`), and $\mathbf{Y}$ (`sourcesMatrix`) as the matrix whose K columns are the spherical harmonics vectors $\mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_K}$ related to each source signal, one can state that

$$\begin{aligned}\mathbf{G'} &= \begin{bmatrix} | & | & & | \\ \mathbf{g_1} & \mathbf{g_2} & \cdots & \mathbf{g_K} \\ | & | & & | \end{bmatrix} \\ &= \begin{bmatrix} & & \\ & \mathbf{C}^{\dagger} & \\ & & \end{bmatrix} \times \begin{bmatrix} | & | & & | \\ \mathbf{y_1} & \mathbf{y_2} & \cdots & \mathbf{y_K} \\ | & | & & | \end{bmatrix} \\ &= \mathbf{C}^{\dagger} \times \mathbf{Y}.\end{aligned}$$

That is also the way $\mathbf{G}$ (`gainsMatrix`) has been defined.

`nfcFilter()`. Recalled inside the ‘while’ cycle of `processReplacing()` routine when NFC filtering is enabled by `nfcSwitch`. This method implements near field compensation on input signals before they are converted in loudspeakers feeds. From the scheme introduced in Figure 4.12 (here reported for convenience) one can deduce that each source must be filtered separately with three different filters (as the encoding/decoding order):

$$H_{1i}^{NFC}(\omega); \quad H_{2i}^{NFC}(\omega); \quad H_{3i}^{NFC}(\omega) \qquad \text{, for the } i\text{-}th \text{ source.}$$

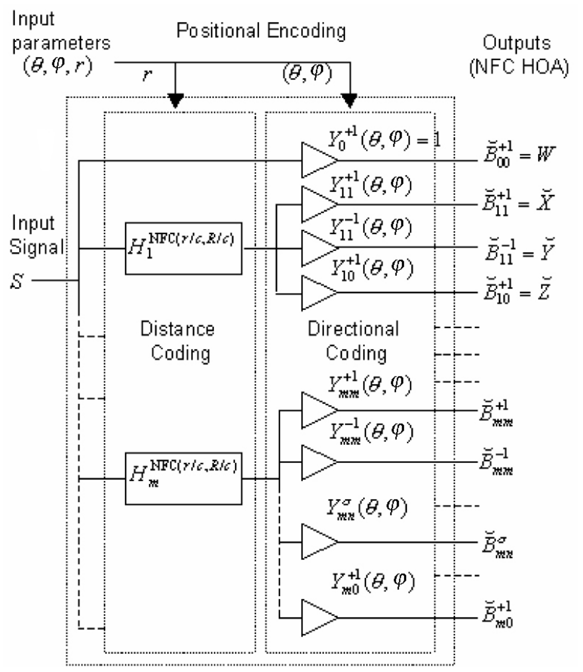

Input parameters (θ,φ,r)
Positional Encoding
r
(θ,φ)
Outputs (NFC HOA)
Input Signal
S
$H_1^{NFC(r/c,R/c)}$
$H_m^{NFC(r/c,R/c)}$
Distance Coding
Directional Coding
$Y_0^{+1}(\theta,\varphi)=1$
$Y_{11}^{+1}(\theta,\varphi)$
$Y_{11}^{-1}(\theta,\varphi)$
$Y_{10}^{+1}(\theta,\varphi)$
$Y_{mm}^{+1}(\theta,\varphi)$
$Y_{mm}^{-1}(\theta,\varphi)$
$Y_{mn}^{\sigma}(\theta,\varphi)$
$Y_{m0}^{+1}(\theta,\varphi)$
$\breve{B}_{00}^{+1}=\breve{W}$
$\breve{B}_{11}^{+1}=\breve{X}$
$\breve{B}_{11}^{-1}=\breve{Y}$
$\breve{B}_{10}^{+1}=\breve{Z}$
$\breve{B}_{mm}^{+1}$
$\breve{B}_{mm}^{-1}$
$\breve{B}_{mn}^{\sigma}$
$\breve{B}_{m0}^{+1}$

Figure 5.8: NFC-HOA positional encoding of a virtual sound source: a distance-coding unit (NFC filter bank) completes the directional encoding.

This is the reason of choosing the storage structures previously described for these filter coefficients.

Hence `nfcFilter()` takes as arguments the signal to be filtered and the filter type index. The filtering operation is then implemented using the relation:

$$S_{mi}^{NFC}(\omega) = H_{mi}^{NFC}(\omega) \cdot S_i(\omega).$$

This is implemented in the time domain with its related difference equation and solved for $s_{mi}^{NFC}(t)$ term, that represents H_{mi}^{NFC} filter output. A generic form of this equation is (written in discrete time n):

$$\begin{aligned} s_{mi}^{NFC}(n) =& \frac{b_0}{a_0} s_i(n) + \frac{b_1}{a_0} s_i(n-1) + \frac{b_2}{a_0} s_i(n-2) + \cdots \\ &+ \frac{a_1}{a_0} s_{mi}^{NFC}(n-1) + \frac{a_2}{a_0} s_{mi}^{NFC}(n-2) + \cdots , \end{aligned}$$

where all coefficients computation has been already defined in Sec. 4.1.3.

This filtering technique requires the storage of filter input and output p past samples, with p specified by the filter order. For this purpose two storage matrices (`pastInsMatrix` and `pastOutsMatrix`) have been used within `nfcFilter()` method. Thus another input argument is used by the latter to index correctly these matrices (see the following pseudo-code).

```
// nfcFilter() routine
//-----------------------------------------------------------------------------
filtSignal ambiSound::nfcFilter(pastOutsIndex, signal, filterIndex){
  filtSignal;
  i = pastOutsIndex;
  j = filterIndex;

  switch(j){// convolution(h(n)*signal(n-i))
    case 0 : filtSignal = ...
             ... convolve signal, pastInsMatrix[i], pastOutsMatrix[i]
                     with numCoeff[i][j], denCoeff[j];// h1(n)
             update pastInsMatrix;
             update pastOutsMatrix;
      break;
    case 1 : filtSignal = ...
             ... convolve signal, pastInsMatrix[i], pastOutsMatrix[i]
                      with numCoeff[i][j], denCoeff[j];// h2(n)
             update pastInsMatrix;
             update pastOutsMatrix;
      break;
    case 2 : filtSignal = ...
             ... convolve signal, pastInsMatrix[i], pastOutsMatrix[i]
                     with numCoeff[i][j], denCoeff[j];// h3(n)
             update pastInsMatrix;
             update pastOutsMatrix;
      break;
  }

  return filtSignal;
}
//-----------------------------------------------------------------------------
```

As already shown in Sec. 4.1.3 the Ambisonics signals achievable from a near field compensated source are computed multiplying each NFC filter output for the source spherical harmonic functions of order corresponding to the specific filter index. This means that, when NFC filter is enabled, the *i-th* source's spherical harmonic functions, in vector $\mathbf{y_i}$, aren't multiplied for the same $s_i(n)$ signal. The multiplication can be split – for the 3^{rd} case – into four sub-multiplication that give the following near field compensated Ambisonics signals $\breve{\mathbf{b}}_{\mathbf{i}}^{NFC}$ for the *i-th* source:

$$\breve{\mathbf{b}}_{\mathbf{i}}^{NFC} = \begin{bmatrix} W_i^{NFC} \\ \\ X_i^{NFC} \\ Y_i^{NFC} \\ Z_i^{NFC} \\ \\ R_i^{NFC} \\ S_i^{NFC} \\ T_i^{NFC} \\ U_i^{NFC} \\ V_i^{NFC} \\ \\ K_i^{NFC} \\ L_i^{NFC} \\ M_i^{NFC} \\ N_i^{NFC} \\ O_i^{NFC} \\ P_i^{NFC} \\ Q_i^{NFC} \end{bmatrix},$$

obtained from the 'split filtering'

$$\begin{aligned}
\mathbf{s_{0i}}^{NFC} &= Y_0^1 s_i(n) = s_i(n) = W_i = W_i^{NFC}, \\
\mathbf{s_{1i}}^{NFC} &= \begin{bmatrix} Y_{11}^{+1} \\ Y_{11}^{-1} \\ Y_{10}^{1} \end{bmatrix} s_{1i}^{NFC}(n) = \begin{bmatrix} X_i^{NFC} \\ Y_i^{NFC} \\ Z_i^{NFC} \end{bmatrix}, \\
\mathbf{s_{2i}}^{NFC} &= \begin{bmatrix} Y_{22}^{+1} \\ Y_{22}^{-1} \\ Y_{21}^{+1} \\ Y_{21}^{-1} \\ Y_{20}^{1} \end{bmatrix} s_{2i}^{NFC}(n) = \begin{bmatrix} R_i^{NFC} \\ S_i^{NFC} \\ T_i^{NFC} \\ U_i^{NFC} \\ V_i^{NFC} \end{bmatrix}, \\
\mathbf{s_{3i}}^{NFC} &= \begin{bmatrix} Y_{33}^{+1} \\ Y_{33}^{-1} \\ Y_{32}^{+1} \\ Y_{32}^{-1} \\ Y_{31}^{+1} \\ Y_{31}^{-1} \\ Y_{30}^{1} \end{bmatrix} s_{3i}^{NFC}(n) = \begin{bmatrix} K_i^{NFC} \\ L_i^{NFC} \\ M_i^{NFC} \\ N_i^{NFC} \\ O_i^{NFC} \\ P_i^{NFC} \\ Q_i^{NFC} \end{bmatrix}.
\end{aligned}$$

This splitting cause a little modification of the previously described technique employed to encode/decode a signal in AmbiSound plugin. When NFC filters are enabled, the computation of `gainsMatrix` ($\mathbf{G}$) and consecutively of the loudspeakers feeds ($\mathbf{G} \times \mathbf{s}$) are simply split or "expanded" in four parts. Hence, after being filtered, $\mathbf{s}$ dimension is quadruplicated obtaining

$$\mathbf{s}^{NFC} = \begin{bmatrix} s_1(n) \\ s_{11}^{NFC}(n) \\ s_{21}^{NFC}(n) \\ s_{31}^{NFC}(n) \\ \\ s_2(n) \\ s_{12}^{NFC}(n) \\ s_{22}^{NFC}(n) \\ s_{32}^{NFC}(n) \\ \\ \vdots \\ \\ s_K(n) \\ s_{1K}^{NFC}(n) \\ s_{2K}^{NFC}(n) \\ s_{3K}^{NFC}(n) \end{bmatrix} .$$

This requires also an expantion of matrix $\mathbf{G} = \mathbf{C}^{\dagger} \times \mathbf{Y}$, obtained splitting the scalar product $\mathbf{c_j}^{\dagger} \cdot \mathbf{y_i}$ (where $\mathbf{c_j}^{\dagger}$ is $\mathbf{C}^{\dagger}$ *j-th* row and $\mathbf{y_i}$ is $\mathbf{Y}$ *i-th* column) into the following four sub-scalar products:

$$g_{ji}^{(0)} = c_0^\dagger \cdot y_0;$$

$$g_{ji}^{(1)} = \begin{bmatrix} c_1^\dagger & c_2^\dagger & c_3^\dagger \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix};$$

$$g_{ji}^{(2)} = \begin{bmatrix} c_4^\dagger & c_5^\dagger & c_6^\dagger & c_7^\dagger & c_8^\dagger \end{bmatrix} \begin{bmatrix} y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix};$$

$$g_{ji}^{(3)} = \begin{bmatrix} c_9^\dagger & c_{10}^\dagger & c_{11}^\dagger & c_{12}^\dagger & c_{13}^\dagger & c_{14}^\dagger & c_{15}^\dagger \end{bmatrix} \begin{bmatrix} y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \end{bmatrix}.$$

Hence the term g_{ji} of $\mathbf{G}$ non-expanded matrix is replaced with the row vector

$$\begin{bmatrix} g_{ji}^{(0)} & g_{ji}^{(1)} & g_{ji}^{(2)} & g_{ji}^{(3)} \end{bmatrix} = \mathbf{G_{ji}}.$$

With this procedure one can obtain a $\mathbf{G}$ matrix with dimensions $N \times 4K$

$$\mathbf{G} = \begin{bmatrix} \mathbf{G_{11}} & \mathbf{G_{12}} & \cdots & \mathbf{G_{1K}} \\ \mathbf{G_{21}} & \mathbf{G_{22}} & \cdots & \mathbf{G_{2K}} \\ \vdots & \vdots & \cdots & \vdots \\ \mathbf{G_{N1}} & \mathbf{G_{N2}} & \cdots & \mathbf{G_{NK}} \end{bmatrix},$$

that can be correctly multiplied for $\mathbf{s}^{NCF}$ obtaining loudspeaker feeds for a reproduction system compensated from near field effects.

***Linear Combination:* $\mathbf{G}\times\mathbf{s}$.** Inside `processReplacing()` routine 'while' cycle, after being eventually filtered, the input sources vector $\mathbf{s}$ is multiplied by the `gainsMatrix` $\mathbf{G}$ to obtain loudspeakers feeds vector $\mathbf{o}$.

Chapter 6

Testing and evaluation

AmbiSound plugin has been tested using the Ambisonics system installed at *'Laboratorio di Elaborazione e produzione dei segnali Audio e Musicali'* of 'Politecnico di Milano' in Como.

The following sections describe the test performed on the system trying to define and evaluate the software behaviour as far as the reproduction system concerns.

6.1 AmbiSound-Spatializer testing

6.1.1 System overview

Figure 6.1 shows the Ambisonics reproduction system installed in the above mentioned laboratory recording room.

The system employs 12 loudspeakers arranged in hemispherical fashion displaced at 1.30 meters away from the origin (sweet-spot). 8 loudspeakers are placed at elevation 0°, approximately at same height of the listener head and 4 loudspeakers are placed at elevation 45° above the listener.

Reaper DAW has been used as host for the plugin testing. Each test has been implemented within a different Reaper 3D mixing session where the AmbiSound has been applied to a single empty track that receives the audio sources as inputs and sends the processed outputs to 12 empty output tracks. Then each output track is routed to the related input channel of the external sound-device used to send audio signals to the loudspeakers.

6.1.2 Tests realization

In order to test AmbiSound 34 listeners have been involved once a time to answer a multiple-choice test realized with four questions, each one with

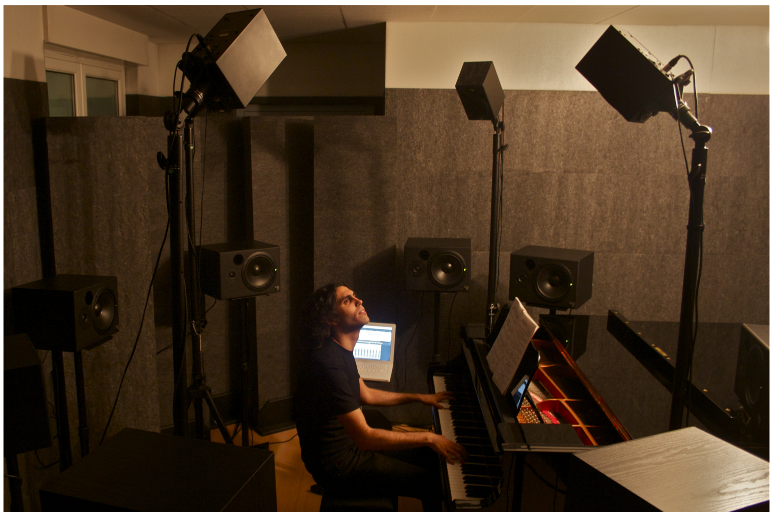


Figure 6.1: Ambisonics reproduction system installation in the recording room of the sound laboratory'Laboratorio di Elaborazione e produzione dei segnali Audio e Musicali of 'Politecnico di Milano' in Como.

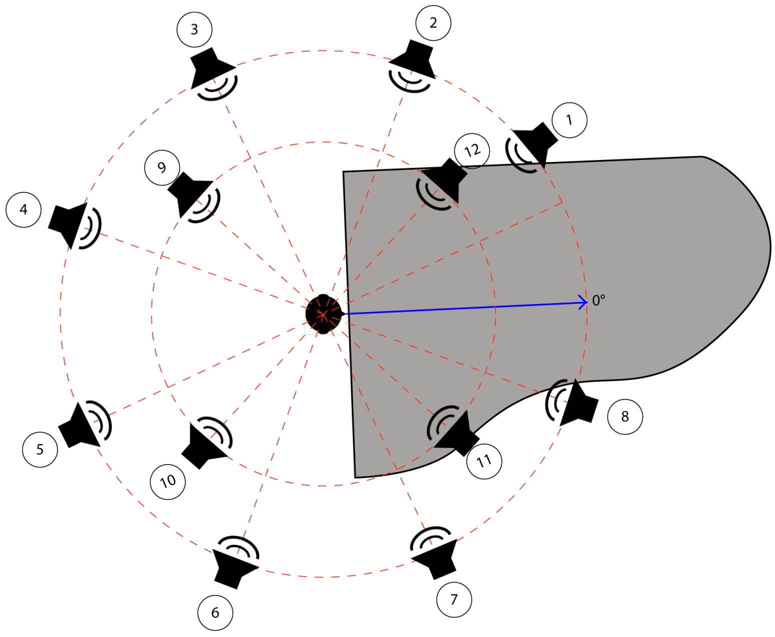


Figure 6.2: Ambisonics reproduction system plan.

four possible answers, two of them completely wrong and only one correct. The fourth choice, similar to the correct one, is introduced to confuse the listener. Therefore, if it is taken as correct, it can still be considered for the system positive evaluation.

ID	$azimth(°)$	$elevation(°)$
1	40.0	0.0
2	67.5	0.0
3	112.5	0.0
4	157.5	0.0
5	202.5	0.0
6	247.5	0.0
7	292.5	0.0
8	−22.5	0.0
9	135.0	45.0
10	−135.0	45.0
11	−45.0	45.0
12	45.0	45.0

Table 6.1: System loudspeakers positions.

The test consist in proposing to the listener four 3D spatialized audio samples (one for each question). The listening of each sample will be repeated 2 times before answering, one with the listener inside the sweet-spot and one outside. So the same question could be answered in two different ways for the two different positions.

The following paragraphs describe tests implementations and questions.

Test 1. A video track has been added together with an audio track into a Reaper session. The video shows four little spheres of different colours (white, yellow, blue and red) that move along different trajectories. AmbiSound has been set to take as input a beaten 440 Hz sinusoid and spatialize it to follow one of the four sphere trajectories. By watching the video and hearing the spatialized track the listener has to understand which sphere has been rendered.

Test 2. A beaten 440 Hz sinusoid has been spatialized using AmbiSound to simulate a very fast circular trajectory around the listener (40 m/s for 20 m perimeter). He has to indicate which type of trajectory (Test 2/a) he thinks to have heard between those shown in figure 6.3 and how many spins the source has done around him (Test 2/b).

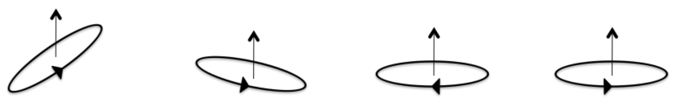


Figure 6.3: Test 2 possible choices.

Test 3. Six audio tracks have been added into a Reaper session to be used as input sources for the plugin. Five of them are street environment recordings and are virtually positioned in different places around the listening point. The sixth is a police siren that is spatialized to follow a specific trajectory, around the listener, between those in figure 6.4. The listener has to indicate which type of trajectory he thinks to have heard.

At the end of this test each listener has been asked to describe the feeling of immersion, inside the rendered sound scene, after having heard once again the system reproduction, this time having disabled the NFC filtering (enabled in all tests mixing). Therefore, each listener has also been asked to do a comparison of the immersion degree between the last and the previous listenings.

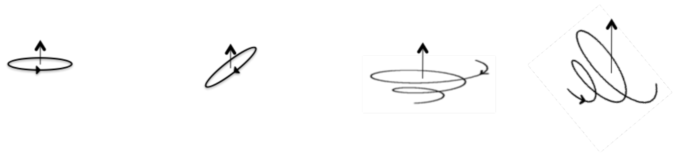


Figure 6.4: Test 3 possible choices.

Test 4. A white noise signal has been spatialized using AmbiSound to simulate a semicircular trajectory above the listener. The latter has to indicate which type of trajectory he thinks to have heard between those shown in figure 6.5.

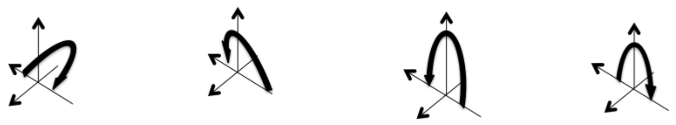


Figure 6.5: Test 4 possible choices.

6.1.3 Test results

Figure 6.6 shows the plugin test results a makes a comparison between the different question showing each answer chart.

Before doing the plugin evaluation (next section) it's better to clarify some aspects regarding the reproduction system arrangement, to justify some unexpected results remarkable in the two graphs of figure 6.6.

First of all the number of loudspeakers is not sufficient for a proper Ambisonics reproduction. As already explained in Sec. 4.1.2 the reproduction layout must respect the hardware constraint (4.30) here reported for convenience:

$$N \geq (m+1)^2,$$

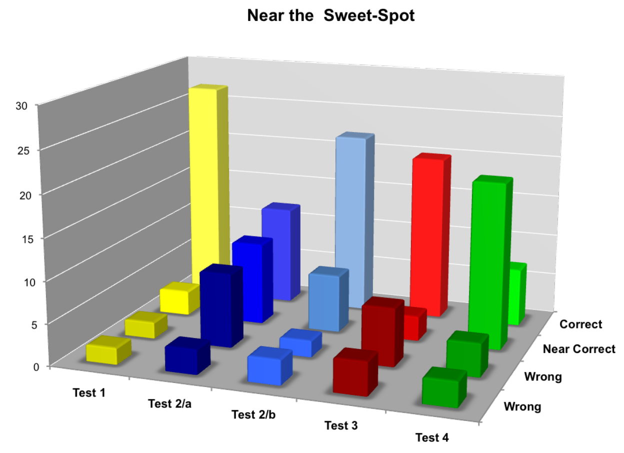
where N is the loudspeakers number to be used for a correct m^{th} order Ambisonics decomposition. Therefore, for a 3^{rd} order reproduction, a system of at least 16 loudspeakers should have been used instead of 12. This absence has produced some uncertainties and misunderstandings in detecting sound sources trajectories due to the fact that the system (4.28) is solved using least square approximation instead of least norm one, finding a different solution for (4.29).

Then, also the reproduction layout configuration (in figure 6.1 and 6.2) must be noticed. From what has been said in Sec. 4.1.2 it's deducible that not only the used loudspeakers arrangement isn't a regular one but it also lacks one of the crucial reproduction points in 3D rendering, the loudspeaker placed precisely above the listening point, at 90 degrees of elevation. This lack has increased the uncertainty in detecting vertical trajectories.

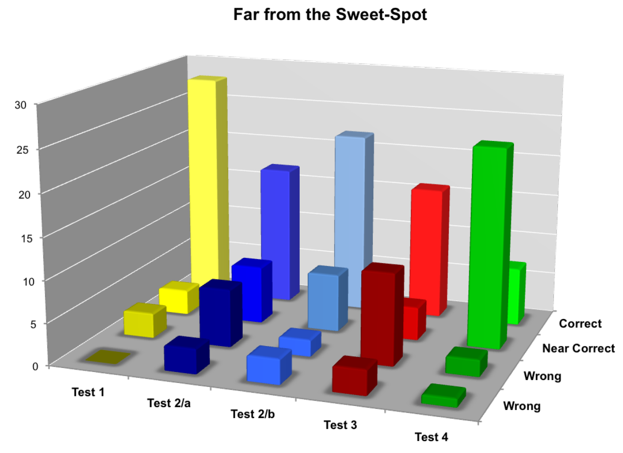
Finally, as shown in figure 6.1 and 6.2, the most part of the reproduction space is taken up by a piano, and a LCD monitor has been positioned in place of the piano note-holder to make the video of test 1 visible. The interaction between the reproduced sound-field and these objects has produced some side-effects due to the introduction inside the listening space of sound reflected components and sound masking. The latter have affected the detection, especially inside the sweet-spot, of sources placed behind or in front of the piano and the monitor. This can justify the fact that the best results in some tests has been obtained for listenings done outside the sweet-spot.

All the above mentioned system irregularities have been introduced for a specific purpose: testing the plugin behaviour in adverse conditions, as close as possible to reality.

Figure 6.6: AmbiSound-Spatializer test results graphs



(a) Near the sweet-spot listenings



(b) Far from the sweet-spot listenings

6.2 Project evaluation

At first sight, from the graphs reported in figure 6.1 and 6.2, it can be noticed that the most part of listeners have taken the correct or near correct answer for each question. So it can be said that the plugin test has globally given a positive result.

Analysing in details each question chart, it can be said that:

1. the interaction between audio and video helps the listener to easily

detect the correct sources position and trajectory. This suggest that AmbiSound can be particularly suitable also for cinema music productions;

2. when implementing very fast sound circular trajectories the listener may have some problem in the detection of the turning direction, but the circular movement perception stays unaltered;

3. without a video integration the listener may have some uncertainty in detecting complex trajectories as spirals. This defect together with the one in reported in point 2 can be widely attributable to the reproduction aliasing previously described as sound masking and reflections;

4. without the loudspeaker placed above him at 90 elevation degrees, the listener has great difficulties in detecting correctly a totally vertical trajectory. This can be easily deduced from Test 2 chart, that shows how the most part of the listeners were convinced to have heard a slightly diagonal trajectory instead of the vertical one;

5. the employment of filters that compensate near field effects cause a marked increasing of the listener immersion feeling in the sound scene. The last has been described from many listeners as a sensation of sound sources major presence associated with a lower frequencies amplification.

In conclusion, taking into consideration that the testing results have been obtained using a non-appropriate reproduction system and the test questions themselves have been thought to stress the listener auditive capabilities, AmbiSound-Spatializer can be regarded as a suitable tool to be used for 3DA mixing.

The implemented NFC filtering has resulted as an efficient tool for simulate sound scene placed nearby the listening point. It allows an advanced manipulation of the signal frequency content, in order to obtained an increase of the "sound presence" perception, obtained automatically only by setting loudspeakers and sources distance from the system origin.

In addition the test results show, at least on sound perception level, a sizeable sweet-spot area although some sound reflecting and masking objects have been placed in the reproduction space.

Chapter 7

Next research directions and conclusions

After the testing phase, the project last step has been to detect the lacks and limitations of the developed software tool. Starting from this critical view it has been possible to draw a conclusion on the whole work done, showing the research directions that may be taken into account for AmbiSound plugin further developments.

Before listing and discuss the above mentioned research directions, other plugin aspects has to be considered to complete the critical analysis done during the evaluation phase.

The following aspects haven't nothing to do with the goodness of AmbiSound plugin process, but concern some side issues that anyway should be considered as crucial.

First of all the implemented Ambisonics decomposition order, that is responsible of audio reproduction fidelity independently from the plugin process, has to be considered. To reach an higher defined sound spatialization using AmbiSound, the realization of higher order Ambisonics (HOA) encoding/decoding stages is aimed, with a corresponding increase of loudspeakers number in the reproduction layout.

A good solution could be to implement Legendre polynomials recursive formulas ([2] [7]) to compute dynamically the spherical harmonic function for different Ambisonics decomposition orders. This will allow the user to decide the reproduction accuracy from time to time, making the most of the employed system .

Other aspects involved in AmbiSound diffusion as a 3DA mixing tool are the plugin usability and the Ambisonics format support diffusion.

The plugin usability is strongly dependent from its graphical user interface (GUI). Therefore, one can increase AmbiSound usability implementing a more user-friendly GUI to be used instead of the one provided by the host (typically a very poor and basic one).

In the specific case of 3D sound spatialization it can be very useful to realize a GUI that, once established a correspondence between sound sources and 3D geometric shapes, makes use of this parallelism to directly handle graphical objects as sound sources. In this way the user will have an immediate visual feedback that, together with the auditive one, will help him to better understand the sources configuration in space.

An example of such GUI is shown in figure 7.1, where the following scene is displayed: a 3D Cartesian system with the listener placed in the origin and sound sources represented as spheres of different colors. All this can be easily implemented using computer graphics developing tools such as Processing or OpenGL APIs. These permit to complete the GUI in figure 7.1 through the realization of control panels where the user can change sources position and graphical appearance by means of sliders, knobs, menus, etc.

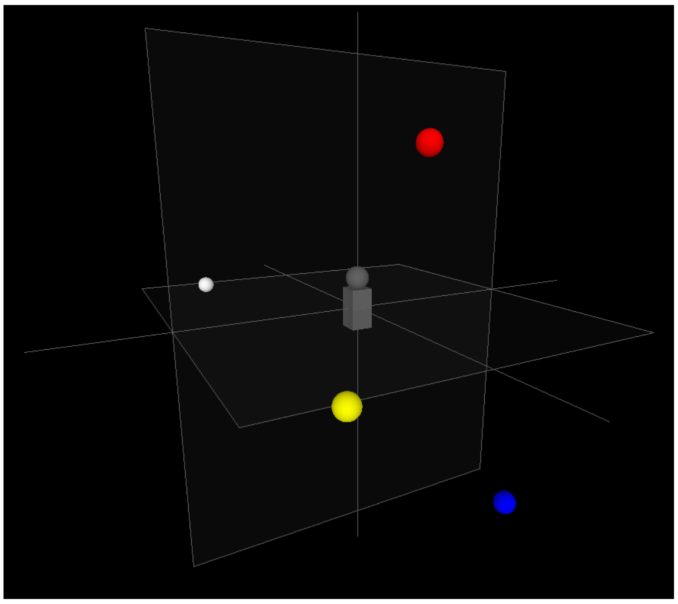


Figure 7.1: An example of a possible AmbiSound GUI.

A further step could be to consider input sources first as graphical objects inside the plugin process itself, using computer graphics techniques to determine sources position in space and related spherical harmonic functions. This introduces the possibility of an easy and low computational cost implementation for system global positioning actions such as translations, rotations around the three axis, etc. In fact a source positioning stage can be developed and placed before the loudspeaker feeds computation (see figure 7.2). This positioning stage could use only the software methods defined for graphical objects operations in order to compute each source spherical coordinates and set it in a 3D virtual scene. Then the spherical coordinates will be used by the plugin to compute each source spherical harmonic functions, avoiding the implementation of translation and rotation matrix operation higher than first order to obtain the same result at higher computational costs. Obviously this is possible only while using, as input signals, artificial sources for which spherical harmonic functions have to be computed.

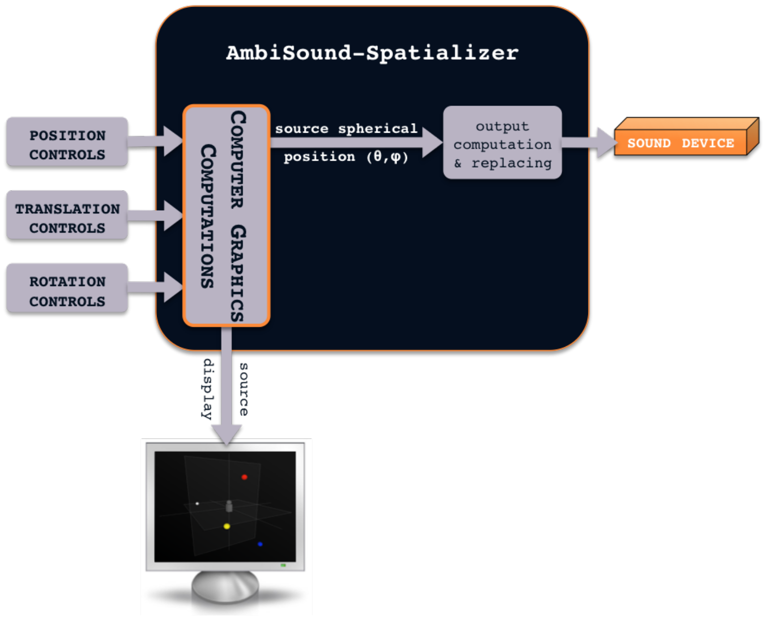


Figure 7.2: Example of a possible AmbiSound plugin architecture employing a source positioning stage that use computer graphics defined operations to compute sources azimuth and elevation.

The last aspect is the already mentioned diffusion of Ambisonics format support by today's multimedia player software and devices. This one can be considered as an indirect improving factor because it is clear that the more Ambisonics mixed audio is supported the more tools implementing Ambisonics mixing will be employed in 3DA rendering and the more these tools will be improved to obtain better performances.

A first step in Ambisonics format diffusion could be achieved realizing three things:

- a digital audio encoding format for Ambisonics mixed audio that fixes the maximum number of Ambisonics signals to be encoded and stored in an audio file;

- a software encoding tool able to compute and store the data related to Ambisonics signals (*B-format*) according to the above mentioned audio format, obtaining a multi-channels audio file container. This operation should be implemented at the end of a 3DA mixing session exporting all the data as a reproducible audio file;
- a digital audio tool that serve as decoder for audio files encoded in the previous format, able to reproduce the entire mixed audio stream, exploiting the knowledge of the specific loudspeakers configuration and the decomposition order used for the encoding operation.

Appendix A

Listing

Basic files required for a VST plugin implementation are a .hpp file, containing the plugin base class declaration, and a .cpp file containing the plugin class implementation, where the behaviour of parameters handling methods, processing methods and plugin variables are defined.

The following script has been developed as header source file for *AmbiSound-Spazializer* plugin. It contains all the usual VST plugin variable and routine declarations inside `AmbiSound` class declaration. The latter contains also declarations of plugin specific variables and methods already discussed in Chapter 5.

```
/*
 *  AmbiSound_Spatializer - AmbiSound_Spat.h
 *  Created by Daniele Magliozzi on 24/06/11
 *  Copyright (c) 2011 Politecnico Milano, All rights reserved
 */

#ifndef __AmbiSound_Spat__
#define __AmbiSound_Spat__

#ifndef __audioeffect__
#include "audioeffectx.h"
#endif



//-------------------------------------------------------------------------------------
// PREPROCESSORS
//-------------------------------------------------------------------------------------


// Libraries
//*******************************************

#include <vector>
#include "armadillo"
#include <Accelerate/Accelerate.h>
using namespace std;
using namespace arma;
#include <iostream>
using std::cout;
using std::endl;
#include <cstdlib>
using std::exit;
#include <fstream>
using std::ifstream;
using std::ofstream;
using std::fstream;
using std::ios_base;
#include <string>
#include <math.h>
#include <sstream>


// Conversion Constants
//*******************************************

#define DEGTORAD_CONV 2*3.14159/360
#define RADTODEG_CONV 360/(2*3.14159)
#define FTORAD_CONV 2*3.14159
#define RADTOF_CONV 1/(2*3.14159)
#define FTOSRAD_CONV 3.14159
#define FTOMET_CONV 99
#define MAX_LOUDSP_NUM 36
#define MAX_SOURCES_NUM 32
#define MAX_LOUDSP_DIST 30.f


//-------------------------------------------------------------------------------------
// GLOBAL VARIABLES DECLARATION
//-------------------------------------------------------------------------------------


// Plugin Variables
//*******************************************

const int kNumInputs = MAX_SOURCES_NUM;
const int kNumOutputs = MAX_LOUDSP_NUM;
const int kNumParameters = kNumInputs*3 + 7;
const int kNumPrograms = 0;
const unsigned long kUniqueId = 'AmbiSp';


//------------------------------------------------------------------------------------
// CLASSES DECLARATION
//------------------------------------------------------------------------------------


// Plugin Class
//*****************************************

class ambiSound : public AudioEffectX {


public:
  ambiSound(audioMasterCallback audioMaster);
  ~ambiSound();

  // Plugin Initialization Method
  virtual void init();

  // Processing Methods
  virtual void processReplacing (float** inputs, float** outputs, ...
                              ... VstInt32 sampleFrames);
  virtual void processDoubleReplacing (double** inputs, double** outputs, ...
                                    ... VstInt32 sampleFrames);

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

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

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

  // AmbiSettings Methods (for float-based computations)

  /* NOTE: every methods or variable that takes the prefix d- is meant to be
   * used when 64-bit precision is required by the host in computations hence
   * it handles double variables but it does/is used for the same action as its
   * 'double' without the d- prefix.
   */

  virtual void setLoudspPos();    // initializes the reproduction system
                                  // loudspeakers arrangement

  virtual void setLoudspMatrix(); // creates the matrix C - spherical
                                  // harmonics of each loudspeaker - and
                                  // computes C' or pinv(C) depending on
                                  // inverseFlag value (0 - 1)

  virtual void setSourcesMatrix(int sources); // create a matrix Y containing
                                              // sheprical harmonics of each
                                              // source

  virtual void setGainsMatrix(int sources, int nfcFlag); // computes the matrix
```

```
                                                        // product G=(C'/N)*Y
                                                        // or G=pinv(C)*Y


  virtual void loudspPosFileUpdate(); // updates loudspeaker distance value
                                      // stored in "loudspPos.dat" file

  // AmbiSettings Methods (for double-based computations)
  virtual void setdLoudspPos();
  virtual void setdLoudspMatrix();
  virtual void setdSourcesMatrix(int sources);
  virtual void setdGainsMatrix(int sources, int nfcFlag);

  // NFC Methods
  virtual void initNumCoeffMatrix(); // methods to be called in init() for the
  virtual void initdNumCoeffMatrix();// allocation of numCoeff cells

  virtual void setNfcDenFilterCoeff(float distance);  // methods to compute NFC
  virtual void setdNfcDenFilterCoeff(double distance);// denominator filters
                                                      // coefficients

  virtual void setNfcNumFilterCoeff(int sourceIndex, float sourceDist);
  virtual void setdNfcNumFilterCoeff(int sourceIndex, double sourceDist);
                                                      // methods to compute NFC
                                                      // numerator filters
                                                      // coefficients

  float nfcFilter(int pastOutsIndex, float signal, int filtIndex);
  double dNfcFilter(int dPastOutsIndex, double signal, int filtIndex);
                                                      // methods to implement
                                                      // the NFC filtering
                                                      // action

protected:

  // Program Variables
  char programName[kVstMaxProgNameLen + 1];

  // AmbiSettings Variables
  int encodingOrder;  // Ambisonics enc/dec order
  int numSphHarm;     // spherical harmonics used in Ambisonics decomposition
  int numSources;     // number of active sources
  int numLoudsp;      // number of active loudspeakers
  int readFileFlag;   // enable/disable flag for file "loudspPos.dat" reading
  int inverseFlag;    // enable/disable flag for psudo-inverse computation
  int oldInverseFlag; // var for loudspMatrix update
  float loudspDist;   // loudspeakers distance from the system origin
  float oldLoudspDist;// var for loudspDist variable update
  float globalPos [3];// sources global manipulations storing

  vector < vector <float> > loudspPos;      // storing cells for loudspeaker
  vector < vector <double> > dLoudspPos;    // positions data

  vector <float> oldSourcesDist;            // storing cells for NFC filters
                                            // numerator updating var

  vector < vector <float> > sourcesPos;     // storing cells for source
                                            // positions data

  vector < vector <float> > sourcesMatrix;  // matrix Y - contains sources
  vector < vector <double> > dSourcesMatrix;// shperical harmonics

  fmat loudspMatrix;                        // matrix C'or pinv(C) - C contains
  mat dLoudspMatrix;                        // loudspeakers spherical harmonics

  vector < vector <float> > gainsMatrix;    // matrix G - used to store the
  vector < vector <double> > dGainsMatrix;  // result of the product pinv(C)*Y
```

```
                                                // or C'*Y

  // NFC variables
  int nfcSwitch;      // enable/disable flag for NFC filtering

  vector < vector <vector <float> > > numCoeff;         // storing cells for
  vector < vector <vector <double> > > dNumCoeff;       // the numerator
                                                        // cofficients of each
                                                        // NFC filter

  vector < vector <float> > denCoeff;                   // storing cells for
  vector < vector <double> > dDenCoeff;                 // the denominator
                                                        // coefficients of each
                                                        // NFC filter

  vector < vector <float> > pastInsMatrix;              // matrices used to
  vector < vector <double> > dPastInsMatrix;            // implement NFC filter
                                                        // input delay line

  vector < vector < vector <float> > > pastOutsMatrix;  // matrices used to
  vector < vector < vector <double> > > dPastOutsMatrix;// implement NFC filter
                                                        // output delay line


};

#endif
```

The following script shows the content of `AmbiSound_Spat.cpp` source file. Both VST plugin usual methods and AmbiSound plugin specific ones are implemented within this file. The plugin main methods working has been already deepened in Chapter 5. Here one can read the entire source developed for the project to understand all the process implementation details.

This script is a ready-to-use one, i.e. it can be copied together with `AmbiSound_Spat.h`, introduced before, into a VST plugin software project and just built to obtain a perfectly working *AmbiSound-Spatializer* plugin.

Additional comments have been inserted into the illustrated source files to describe the principles operation and to provide a better comprehension. As already discussed the process implementation and all related variables and methods are doubled to create a plugin that is supported both by hosts that use single precision in computation or by those that use double one. Thus, to avoid useless and redundant descriptions, in the major part of the script related to the implementation of the process supporting double precision computation, comments have been neglected.

```
/*
 *  AmbiSound_Spatializer - AmbiSound_Spat.cpp
 *  Created by Daniele Magliozzi on 24/06/11
 *  Copyright (c) 2011 Politecnico Milano, All rights reserved
 */

#ifndef __AmbiSound_Spat__
#include "AmbiSound_Spat.h"
#endif



//------------------------------------------------------------------------------------
// PLUGIN RECALLING METHOD
//------------------------------------------------------------------------------------


//------------------------------------------------------------------------------------
/* This method is called by the host application when the user select a plugin from
 * the releated list creating an istance of the specific effect - VST plugin
 */
AudioEffect* createEffectInstance(audioMasterCallback audioMaster) {

  return new ambiSound(audioMaster);
}
//------------------------------------------------------------------------------------






//------------------------------------------------------------------------------------
// PLUGIN CLASS
//------------------------------------------------------------------------------------


// Class Constructor
//*****************************************
//------------------------------------------------------------------------------------
ambiSound::ambiSound(audioMasterCallback audioMaster)
: AudioEffectX(audioMaster, kNumPrograms, kNumParameters) {
  setNumInputs(kNumInputs);  // set plugin available input channels
  setNumOutputs(kNumOutputs);// set plugin available output channels
  setUniqueID(kUniqueId);    // set plugin ID
  canProcessReplacing ();    // supports replacing output
  canDoubleReplacing ();     // supports double precision processing
  init();                    // initialize all plugin variables
}
//------------------------------------------------------------------------------------




// Class Destructor
//*****************************************
//------------------------------------------------------------------------------------
ambiSound::~ambiSound() {}
//------------------------------------------------------------------------------------




// Plugin Initialization Method
//*****************************************
// Used to set plugin parameters to default values
//------------------------------------------------------------------------------------
```

```
void ambiSound::init() {

  encodingOrder = 3;                              // 3rd Ambisonics enc/dec order
  numSphHarm = pow(float(encodingOrder + 1), 2); // 16 spherical harmonics
  loudspDist = 0.f;                               // loudspeakers default distance 1m
  oldLoudspDist = loudspDist;
  nfcSwitch = 0;                                  // default NFC filter on
  numSources = 1;                                 // input source default 1
  numLoudsp = 16;                                 // output loudspeakers default 16
  readFileFlag = 0;                               // "loudspPos.dat" reading disabled
  inverseFlag = 0;                                // default use transposition rule
  oldInverseFlag = inverseFlag;                   // for system solving

  //Setting sources global position to az=0°; el=0°; d=0m
  for(int i=0; i<=2; i++)
    globalPos[i]=0.0;

  //Setting the system speaker arrangement and computing matrix C'
  vector <float> initPos;
  initPos.push_back(0.0); initPos.push_back(0.0);
  vector <double> initdPos;
  initdPos.push_back(0.0); initdPos.push_back(0.0);
  for(int n=0; n<=(kNumOutputs - 1); n++){
    loudspPos.push_back(initPos);
    dLoudspPos.push_back(initdPos);
  }
  setLoudspPos();
  setdLoudspPos();

  //Setting NFC filters impulse responses denominator
  setNfcDenFilterCoeff(loudspDist);
  setdNfcDenFilterCoeff(double(loudspDist));

  //Allocating space for NFC filters numerator coefficients cells
  initNumCoeffMatrix();
  initdNumCoeffMatrix();

  //Setting default source positions to az=0°; el=0°; d=1m, and NFC filter numerators
  vector <float> defaultSourcesPos;

  defaultSourcesPos.push_back(0.f);                    // azimuth
  defaultSourcesPos.push_back(0.f);                    // elevation
  defaultSourcesPos.push_back(1.f);                    // distance

  for(int i=0; i<=(kNumInputs - 1); i++){
    sourcesPos.push_back(defaultSourcesPos);           // i-th source position
    oldSourcesDist.push_back(defaultSourcesPos.at(2));// i-th source old distance

    //Setting NFC filters impulse responses numerator coefficients
    setNfcNumFilterCoeff(i, oldSourcesDist.at(i));
    setdNfcNumFilterCoeff(i, double(oldSourcesDist.at(i)));
  }

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

  //Setting pastOutsMatrix - for filters output delay line
  vector <float> row;
  vector < vector <float> > initPastOuts;
  for (int j=0; j<=2; j++){
    row.push_back(0.0);     // default sample amplitude 0.0
    initPastOuts.push_back(row);
  }
  for(int i=0; i<=(kNumInputs - 1); i++)
    pastOutsMatrix.push_back(initPastOuts);

  //Setting dPastOutsMatrix - for dfilters output delay line
  vector <double> dRow;
  vector < vector <double> > dInitPastOuts;
  for (int j=0; j<=2; j++){
    dRow.push_back(0.0);   // default sample amplitude 0.0
    dInitPastOuts.push_back(dRow);
  }
  for(int i=0; i<=(kNumInputs - 1); i++)
    dPastOutsMatrix.push_back(dInitPastOuts);

  //Setting pastInsMatrix - for filters input delay line
  row.clear();
  for (int j=0; j<=2; j++){
    row.push_back(0.0);    // default sample amplitude to 0.0
  }
  for(int i=0; i<=(kNumInputs - 1); i++)
    pastInsMatrix.push_back(row);

  //Setting dPastInsMatrix - for dfilters input delay line
  dRow.clear();
  for (int j=0; j<=2; j++){
    dRow.push_back(0.0);   // default sample amplitude to 0.0
  }
  for(int i=0; i<=(kNumInputs - 1); i++)
    dPastInsMatrix.push_back(dRow);
}
//-------------------------------------------------------------------------------------




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

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




// Parameters Handling Methods
//*******************************************
// NOTE: the 'index' variable contain the parameter ID - an integer between 0 and
// kNumParameters-1
//-------------------------------------------------------------------------------------
void ambiSound::setParameter (VstInt32 index, float value){
  // set par. values into variables

  switch (index) {
    case 0:                 // set sources number par.
      numSources = int(floor(value*(kNumInputs - 1)) + 1);
      break;

    case 1:{                // loudspeakers distance parameter setting
      if(value<=0.5)
        loudspDist = (value/0.5)+1;
      if(value>0.5 && value<=0.779999)
        loudspDist = ((value-0.5)/0.25)*8 + 2;
      if(value>0.779999)
        loudspDist = ((value-0.75)/0.25)*22 + 8;
      loudspPosFileUpdate();

      if(loudspDist >= 30.0)// NFC filter enabling flag parameter setting
        nfcSwitch = 1;
      else
        nfcSwitch = 0;
    }
      break;

    case 2:                 // set read "loudspPos.dat" enabling/disabling flag par.
      int a = int(floor(value));
      if(readFileFlag == 2 && a==1)
        readFileFlag = 0;
      else
        readFileFlag = a;
      break;

    case 3:{                // set matrix inverse computation technique flag par.
      if(value <= 0.500000)
        inverseFlag = 0;
      if(value > 0.500000)
        inverseFlag = 1;
    }
      break;

    case 4: case 5: case 6: // set global azimuth, elevation and distance par.
      switch(index - 4){
        case 0: case 1:     // global azimuth and elevation
          globalPos[index - 4]=value*FTORAD_CONV;
          break;

        case 2:             // global distance
          if(value<=0.125)
            globalPos[2] = (value/0.125)*0.99+0.01;
          if(value>0.125 && value<=0.5)
            globalPos[2] = ((value-0.125)/(0.5-0.125))+1;
          if(value>0.5 && value<=0.759999)
            globalPos[2] = ((value-0.5)/0.25)*8 + 2;
          if(value>0.759999)
            globalPos[2] = ((value-0.75)/0.25)*92 + 8;
          break;
      }
      break;

    default: {              // set sources azimuth, elevation and distance par.

      int vectorIndex = (index - 7)/3;  // find what source has to be set
      int positionIndex = (index - 7)%3;// find what source parameter has to be set
      if(positionIndex==2){ // sources distance setting (positionIndex=2)
        if(value<=0.125)
          sourcesPos.at(vectorIndex).at(positionIndex) = (value/0.125)*0.99+0.01;
        if(value>0.125 && value<=0.5)
          sourcesPos.at(vectorIndex).at(positionIndex) = ...
                                                     ... ((value-0.125)/(0.5-0.125))+1;
        if(value>0.5 && value<=0.759999)
          sourcesPos.at(vectorIndex).at(positionIndex) = ((value-0.5)/0.25)*8 + 2;
        if(value>0.759999)
          sourcesPos.at(vectorIndex).at(positionIndex) = ((value-0.75)/0.25)*92 + 8;
      }
      else{                 // source azimuth or elevation setting
        sourcesPos.at(vectorIndex).at(positionIndex) = value*FTORAD_CONV;
      }
    }
      break;
  }
}
//-------------------------------------------------------------------------------------
```

```
//-------------------------------------------------------------------------------------
float ambiSound::getParameter (VstInt32 index){
  // get par. values

  float v = 0;

  switch (index) {
    case 0:               // get sources number par. value
      v = (float(numSources) - 1)/(kNumInputs - 1);
      break;

    case 1:               // get loudspeakers distance par. value
      if(loudspDist<=2.0)
        v = (loudspDist - 1)*0.5;
      if(loudspDist>2.0 && loudspDist<=10.639912)
        v = ((loudspDist - 2)/8)*0.25 + 0.5;
      if(loudspDist>10.639912)
        v = ((loudspDist - 8)/22)*0.25 + 0.75;
      break;

    case 2:               // get sources number par. value
      v = readFileFlag;
      break;

    case 3:               // get NFC filtering flag value
      v = inverseFlag;
      break;

    case 4:case 5:case 6:// get global sources position par. value
      switch(index - 4){
        case 0: case 1:  // global azimuth and elevation
          v = globalPos[index - 4]*RADTOF_CONV;
          break;

        case 2:          // global distance
          if(globalPos[2]<=1)
            v = ((globalPos[2] - 0.01)/0.99)*0.125;
          if(globalPos[2]>1 && globalPos[2]<=2)
            v = ((globalPos[2] -1)*(0.5-0.125)) + 0.125;
          if(globalPos[2]>2 && globalPos[2]<=10.319968)
            v = ((globalPos[2] - 2)/8)*0.25 + 0.5;
          if(globalPos[2]>10.319968)
            v = ((globalPos[2] - 8)/92)*0.25 + 0.75;
          break;
      }
      break;

    default: {            // get source positions par. value
                          // azimuth(positionIndex=0)
                          // elevation(positionIndex=1)
                          // distance(positionIndex=2)
      int vectorIndex = (index - 7)/3;   // find what source has to be set
      int positionIndex = (index - 7)%3; // find what source parameter has to be set
      if(positionIndex==2){
        if(sourcesPos.at(vectorIndex).at(positionIndex)<=1)
          v = ((sourcesPos.at(vectorIndex).at(positionIndex) - 0.01)/0.99)*0.125;
        if(sourcesPos.at(vectorIndex).at(positionIndex)>1 && ...
            ... sourcesPos.at(vectorIndex).at(positionIndex)<=2)
          v = ((sourcesPos.at(vectorIndex).at(positionIndex) -1)*(0.5-0.125))+0.125;

        if(sourcesPos.at(vectorIndex).at(positionIndex)>2 && ...
            ...sourcesPos.at(vectorIndex).at(positionIndex)<=10.319968)
          v = ((sourcesPos.at(vectorIndex).at(positionIndex) - 2)/8)*0.25 + 0.5;

        if(sourcesPos.at(vectorIndex).at(positionIndex)>10.319968)
          v = ((sourcesPos.at(vectorIndex).at(positionIndex) - 8)/92)*0.25 + 0.75;
      }
      else{
        v = sourcesPos.at(vectorIndex).at(positionIndex)*RADTOF_CONV;
      }
    }
      break;
  }

  return v;
}
//-----------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------
void ambiSound::getParameterName (VstInt32 index, char* label){
  // display par. names

  char name [50] ;

  switch (index) {
    case 0:                // display sources number par. name
      sprintf(name, "Sources No.");
      break;

    case 1:                // display NCF filter flag name
      sprintf(name, "NFC Filter");
      break;

    case 2:                // display "loudspPos.dat" reading flag name
      sprintf(name, "Set LS positions");
      break;

    case 3:                // display inverse computation technique flag name
      sprintf(name, "Inverse Comp.");
      break;

    case 4:case 5:case 6:{ // display global sources position par. name
      switch(index - 4){
        case 0:
          sprintf(name, "Environment Azimuth");
          break;
        case 1:
          sprintf(name, "Environment Elevation");
          break;
        case 2:
          sprintf(name, "Environment Distance");
          break;
        default:
          break;
      }
    }
      break;

    default: {             // display source positions par. name
      int vectorIndex = (index - 7)/3;
      int positionIndex = (index - 7)%3;

      switch (positionIndex) {
        case 0:
          sprintf(name, "Source%d Azimuth  ", int(vectorIndex + 1));
          break;
        case 1:
          sprintf(name, "Source%d Elevation ", int(vectorIndex + 1));
          break;
        case 2:
          sprintf(name, "Source%d Distance  ", int(vectorIndex + 1));
          break;
        default:
          break;
        }
      }
        break;
    }

    vst_strncpy (label, name, 30);
}
//-----------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------
void ambiSound::getParameterDisplay (VstInt32 index, char* text){
  //display par. values

  switch (index) {
    case 0:                 // display the number of active sources
      int2string(numSources, text, 30);
      break;

    case 1:{                // display the NFC flag/loudspeakers distance value

      if(nfcSwitch == 1)    // display NFC filtering disabled
        vst_strncpy (text, "off", 30);
      else                  // display loudspeakers distance - NFC enabled
        float2string(loudspDist, text, 30);
    }
      break;
    case 2:                 // display read "loudspPos.dat" flag status
      if(readFileFlag == 0)
        vst_strncpy (text, "disabled", 30);
      else
        vst_strncpy (text, "enabled", 30);
      break;

    case 3:{                // display the technique used to compute C inverse
      switch(inverseFlag){
        case 0:
          vst_strncpy (text, "regular layout", 30);
          break;
        case 1:
          vst_strncpy (text, "non-regular layout", 30);
          break;

      }
    }
      break;

    case 4:case 5:case 6:{ // display global sources position values
      float v = 0.0;

      switch(index - 4){
        case 0:             // diplay global azimuth value
          v = globalPos[0]*RADTODEG_CONV;
          float2string(v, text, 30);
          break;
        case 1:             // diplay global elevation value
          v = globalPos[1]*RADTODEG_CONV;
          float2string(v, text, 30);
          break;
        case 2:             // diplay global distance value
          v = globalPos[2];
          if(v<=0.01999)
            v=0.f;
          float2string(v, text, 30);
          break;
        default:
          break;
      }
    }
      break;

    default: {              // diplay source positions par. values
      int vectorIndex = (index - 7)/3;
      int positionIndex = (index - 7)%3;
      float v = 0.0;

      switch (positionIndex) {
        case 0:             // display source azimuth value
          v = sourcesPos.at(vectorIndex).at(positionIndex)*RADTODEG_CONV;
          float2string(v, text, 30);
          break;
        case 1:             // display source elevation value
          v = sourcesPos.at(vectorIndex).at(positionIndex)*RADTODEG_CONV;
          float2string(v, text, 30);
          break;
        case 2:             // display source distance value
          v = sourcesPos.at(vectorIndex).at(positionIndex);
          if(v<=0.01999)
            v=0.f;
          float2string(v, text, 30);
          break;
      }
    }
      break;
  }
}
//-----------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------
void ambiSound::getParameterLabel (VstInt32 index, char* label){
  // display par. unity measure (um)

  switch (index) {
    case 0:                // display no um for sources number
      break;
    case 1:                // display no um if NFC filter is disabled
      if(nfcSwitch==0)     // display loudspeakers distance um - NFC enabled
        vst_strncpy(label, "metres", 20);
      break;
    case 2:                // display no um for "loudspPos.dat" read flag
      break;
    case 3:                // display no um for inverse computation technique flag
      break;
    case 4:case 5:case 6:{// display global sources position um
      switch(index - 4){
        case 0:            // global source azimuth um
          vst_strncpy(label, "degrees" , 20);
          break;
        case 1:            // global source elevation um
          vst_strncpy(label, "degrees", 20);
          break;
        case 2:            // global source distance um
          vst_strncpy(label, "metres", 20);
          break;
        default:
          break;
      }
    }
      break;
    default: {             // display source positions um
      int positionIndex = (index - 7)%3;

      switch (positionIndex) {
        case 0:            // source azimuth um
```

```
          vst_strncpy(label, "degrees" , 20);
          break;
        case 1:           // source elevation um
          vst_strncpy(label, "degrees", 20);
          break;
        case 2:           // source distance um
          vst_strncpy(label, "metres", 20);
          break;
        default:
          break;
      }
    }
      break;
  }
}
//-------------------------------------------------------------------------------------




// Tagging Methods
//*****************************************
//-------------------------------------------------------------------------------------
bool ambiSound::getEffectName (char* name){
  vst_strncpy (name, "AmbiSound Spatializer", kVstMaxEffectNameLen);
  return true;
}
//-------------------------------------------------------------------------------------

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

//-------------------------------------------------------------------------------------
bool ambiSound::getVendorString (char* text){
  vst_strncpy (text, "Daniele Magliozzi", kVstMaxVendorStrLen);
  return true;
}
//-------------------------------------------------------------------------------------

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




// Processing Methods
//*****************************************
//-------------------------------------------------------------------------------------
void ambiSound::processReplacing (float** inputs, float** outputs, ...
                                  ...VstInt32 sampleFrames){

  vector<float*> ins; // vector that will contain pointers at the beginning of each
                      // input buffer

  vector<float*> outs;// vector that will contain pointers at the beginning of each
                      // output buffer

  float q = 1/4.00;

  int sources = numSources;                // define active sources number in this
```

```
                                          // scope

  int filtFlag = nfcSwitch;               // define NFC filter state in this scope

  int readLoudspPos = int(getParameter(2));// define read "loudspPos.dat" flag state
                                          // in this scope

  int systemFlag = int(getParameter(3));  // define the Ambisonics system solving
                                          // method in this scope

  for(int i=0; i<=(sources - 1); i++)     // assign inputs buffer pointers
    ins.push_back(inputs[i]);

  if(readLoudspPos==1){                   // update loudspeaker positions
    readFileFlag = 2;
    setLoudspPos();
  }
  else{
    if(systemFlag == 1)
      setLoudspMatrix();
  }

  for(int i=0; i<=(numLoudsp - 1); i++)   // assign output buffer pointers
    outs.push_back(outputs[i]);

  // Compute NFC filter coefficients

  for(int k=0; k<=(sources - 1); k++){
    if(oldSourcesDist.at(k)!=sourcesPos.at(k).at(2)){// update filter numerator
      oldSourcesDist.at(k) = sourcesPos.at(k).at(2); // coefficients for sources
      setNfcNumFilterCoeff(k, oldSourcesDist.at(k)); // whose distance has been
                                                     // changed
    }
  }

  if(filtFlag==0 && oldLoudspDist!=loudspDist){      // upadate filter denominator
    oldLoudspDist = loudspDist;                      // coefficients if loudspeakers
    setNfcDenFilterCoeff(oldLoudspDist);             // distance has been changed
  }

  // Create inputs spherical harmonic functions matrix Y
  setSourcesMatrix(sources);

  // Compute and store matrix product [pinv(C)*Y] or [(C'/N)*Y] in G
  setGainsMatrix(sources, filtFlag);

  // Next cycle is used by the plugin to implement the process - Ambisonics
  // spatialization - on inputs buffer samples replacing outputs buffer content
  while (--sampleFrames >= 0)
  {
    // Process with NFC filtering
    if(filtFlag == 0){

      vector <float> signals;               // vector of input signals samples

      /* NOTE:
       * the NFC filter separates each sample of the input signal into 4 components,
       * one for each order of spherical harmonics (0-1-2-3° order), for this reason
       * also the G matrix should devide each gain component into 4 so that the
       * moltiplication G*S will return the vector of outputs properly filtered
       */

      // NFC filtering
      for(int k=0; k<=(sources*4 - 1); k+=4){// the cycle step is 4 to live free
                                            // space inside vector 'signals' for
                                            // the 3 filtered components
```

```
        signals.push_back(*ins.at(k*q)++);   // the 0th component is left unfiltered
                                             // 'cause it correspond to the
                                             // spherical harmonic of order zero
                                             // that hasn't to be  filtered

        float f1 = nfcFilter(k*q, signals.at(k), 0); // apply the NFC for the 1st
        signals.push_back(f1);                       // order harmonics

        float f2 = nfcFilter(k*q, signals.at(k), 1); //apply the NFC for the 2nd
        signals.push_back(f2);                       // order harmonics

        float f3 = nfcFilter(k*q, signals.at(k), 2); //apply the NFC for the 3rd
        signals.push_back(f3);                       // order harmonics
      }

      // Gains application and output feeding
      for(int n=0; n<=(numLoudsp - 1); n++){
        float outputSignal = 0.0;

        for(int k=0; k<=(sources*4 - 1); k++){       // scalar product g_(ji)*s
          outputSignal += (gainsMatrix.at(n).at(k))*(signals.at(k));
        }

        (*outs.at(n)++) = outputSignal;              // output feeding
      }

      signals.clear();
    }

    // Process without NFC filtering
    else{

      vector <float> signals;                // vector of input signals samples

      for(int k=0; k<=(sources - 1); k++)    // 'signal' feeding
        signals.push_back(*ins.at(k)++);

      // Gains application and output feeding
      for(int n=0; n<=(numLoudsp - 1); n++){
        float outputSignal = 0.0;

        for(int k=0; k<=(sources - 1); k++){ // scalar product g_(ji)*s
          outputSignal += (gainsMatrix.at(n).at(k))*(signals.at(k));
        }

        (*outs.at(n)++) = outputSignal;      // output feeding
      }

      signals.clear();
    }
  }
}
//-------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------
void ambiSound::processDoubleReplacing (double** inputs, double** outputs, ...
                                        ... VstInt32 sampleFrames){

  vector<double*> ins;
  vector<double*> outs;
  float q = 1/4.00;

  int sources = numSources;
  int filtFlag = nfcSwitch;
  int readLoudspPos = int(getParameter(2));
  int systemFlag = int(getParameter(3));
```

```
for(int i=0; i<=(sources - 1); i++)
  ins.push_back(inputs[i]);

if(readLoudspPos==1){
  readFileFlag = 2;
  setdLoudspPos();
}
else{
  if(systemFlag == 1 && systemFlag != oldInverseFlag){
    setdLoudspMatrix();
    oldInverseFlag = systemFlag;
  }

  if(systemFlag == 0 && systemFlag != oldInverseFlag){
    setdLoudspMatrix();
    oldInverseFlag = systemFlag;
  }
}

for(int i=0; i<=(numLoudsp - 1); i++)
  outs.push_back(outputs[i]);

for(int k=0; k<=(sources - 1); k++){
  if(oldSourcesDist.at(k)!=sourcesPos.at(k).at(2)){
    oldSourcesDist.at(k) = sourcesPos.at(k).at(2);
    setdNfcNumFilterCoeff(k, double(oldSourcesDist.at(k)));

  }
}

if(filtFlag==0 && oldLoudspDist!=loudspDist){
  oldLoudspDist = loudspDist;
  setdNfcDenFilterCoeff(double(oldLoudspDist));
}

setdSourcesMatrix(sources);

setdGainsMatrix(sources, filtFlag);

while (--sampleFrames >= 0)
{
  if(filtFlag == 0){
    vector <double> signals;

    for(int k=0; k<=(sources*4 - 1); k+=4){
      signals.push_back(*ins.at(k*q)++);

      double d1 = dNfcFilter(k*q, signals.at(k), 0);
      signals.push_back(d1);
      double d2 = dNfcFilter(k*q, signals.at(k), 1);
      signals.push_back(d2);
      double d3 = dNfcFilter(k*q, signals.at(k), 2);
      signals.push_back(d3);

    }

    for(int n=0; n<=(numLoudsp - 1); n++){
      double outputSignal = 0.0;

      for(int k=0; k<=(sources*4 - 1); k++){
        outputSignal += (dGainsMatrix.at(n).at(k))*(signals.at(k));
      }
      (*outs.at(n)++) = outputSignal;
    }
    signals.clear();
  }
  else{
      vector <double> signals;

      for(int k=0; k<=(sources - 1); k++)
        signals.push_back(*ins.at(k)++);

      for(int n=0; n<=(numLoudsp - 1); n++){
        double outputSignal = 0.0;

        for(int k=0; k<=(sources - 1); k++){
          outputSignal += (dGainsMatrix.at(n).at(k))*(signals.at(k));
        }
        (*outs.at(n)++) = outputSignal;
      }
      signals.clear();
    }
  }
}
//-------------------------------------------------------------------------------------




// AmbiSettings Methods (float methods)
//*******************************************
//-------------------------------------------------------------------------------------
void ambiSound::setLoudspPos() {
  // Open "loudspPos.dat" file
  ifstream inFile( "/Library/Audio/Plug-Ins/VST/loudspPos.dat", ios::in );

  // Exit program if could not open file
  if ( !inFile )
  {
    cerr << "File could not be opened" << endl;
    exit( 1 );
  }

  inFile.seekg(0);               // point file beginning
  inFile >> loudspDist;                // read loudspeakers distance
  if(loudspDist > MAX_LOUDSP_DIST){    // upper bound loudspeakers distance
    loudspDist = MAX_LOUDSP_DIST;
    nfcSwitch = 1;
  }
  else{                          // lower bound loudspeakers distance
    if(loudspDist < 1.00)
      loudspDist = 1.f;
  }

  inFile >> numLoudsp;           // read active loudspeakers number
  if(numLoudsp > MAX_LOUDSP_NUM) // upper bound active loudspeakers number (36)
    numLoudsp = MAX_LOUDSP_NUM;
  else                           // lower bound active loudspeakers number
    if(numLoudsp < 2)
      numLoudsp = 2;

  for (int n=0; n<=(numLoudsp -1); n++){ // read loudspeaker positions data: az; el
    float position;
    if (!inFile.eof()){                  // feed 'loudspPos' vector with position
                                         // infos for each loudspeaker

      inFile >> position;                // azimuth
      loudspPos.at(n).at(0) = (fmod(position, float(360.0))) * DEGTORAD_CONV;

      inFile >> position;                // elevation
      loudspPos.at(n).at(1) = (fmod(position, float(360.0))) * DEGTORAD_CONV;
    }
  }
  setLoudspMatrix();                // update C matrix
}
//----------------------------------------------------------------------------------

//----------------------------------------------------------------------------------
void ambiSound::setLoudspMatrix() {

  loudspMatrix.reset();
  loudspMatrix.set_size(numLoudsp, numSphHarm);

  for(int i=0; i<=(numLoudsp - 1); i++){// Compute the spherical harmonic function
                                     // for each loudspeaker till 3rd order and
                                     // fill C' rows
    loudspMatrix(i,0) = 1.0;

    loudspMatrix(i,1) = sqrt(3.0)*cos(loudspPos.at(i).at(0))* ...
                        ... cos(loudspPos.at(i).at(1));

    loudspMatrix(i,2) = sqrt(3.0)*sin(loudspPos.at(i).at(0))* ...
                        ... sin(loudspPos.at(i).at(1));

    loudspMatrix(i,3) = sqrt(3.0)*sin(loudspPos.at(i).at(1));

    loudspMatrix(i,4) = (sqrt(24.0)/2)*cos(2*loudspPos.at(i).at(0))* ...
                        ... pow(cos(loudspPos.at(i).at(1)),2);

    loudspMatrix(i,5) = (sqrt(24.0)/2)*sin(2*loudspPos.at(i).at(0))* ...
                        ... pow(cos(loudspPos.at(i).at(1)),2);

    loudspMatrix(i,6) = (sqrt(24.0)/2)*cos(loudspPos.at(i).at(0))* ...
                        ... sin(2*loudspPos.at(i).at(1));

    loudspMatrix(i,7) = (sqrt(24.0)/2)*sin(loudspPos.at(i).at(0))* ...
                        ... sin(2*loudspPos.at(i).at(1));

    loudspMatrix(i,8) = (sqrt(5.0)/2)*((3*pow(sin(loudspPos.at(i).at(1)),2)) - 1);

    loudspMatrix(i,9) = (sqrt(35.0/8.0))*cos(3*loudspPos.at(i).at(0))* ...
                        ... pow(cos(loudspPos.at(i).at(1)),3);

    loudspMatrix(i,10) = (sqrt(35.0/8.0))*sin(3*loudspPos.at(i).at(0))* ...
                         ... pow(cos(loudspPos.at(i).at(1)),3);

    loudspMatrix(i,11) = (sqrt(7.0*15.0)/2)*cos(2*loudspPos.at(i).at(0))* ...
                         ... sin(loudspPos.at(i).at(1))* ...
                         ... pow(cos(loudspPos.at(i).at(1)),2);

    loudspMatrix(i,12) = (sqrt(7.0*15.0)/2)*sin(2*loudspPos.at(i).at(0))* ...
                         ... sin(loudspPos.at(i).at(1))* ...
                         ... pow(cos(loudspPos.at(i).at(1)),2);

    loudspMatrix(i,13) = (sqrt(21.0/8.0))*cos(loudspPos.at(i).at(0))* ...
                         ... cos(loudspPos.at(i).at(1))* ...
                         ... (5*pow(sin(loudspPos.at(i).at(1)),2) - 1);

    loudspMatrix(i,14) = (sqrt(21.0/8.0))*sin(loudspPos.at(i).at(0))* ...
                         ... cos(loudspPos.at(i).at(1))* ...
                         ... (5*pow(sin(loudspPos.at(i).at(1)),2) - 1);

    loudspMatrix(i,15) = (sqrt(7.0)/2)*sin(loudspPos.at(i).at(1))* ...
                         ... (5*pow(sin(loudspPos.at(i).at(1)),2) - 3);
  }

  if(inverseFlag == 1){              // Compute C pseudo-inverse (left or right)
    fmat C = trans(loudspMatrix);
    loudspMatrix.reset();
    loudspMatrix = pinv(C);
  }

}
//--------------------------------------------------------------------------------------

//--------------------------------------------------------------------------------------
void ambiSound::setSourcesMatrix(int sources) {

  float azimuth ;
  float elevation ;
  float distance ;
  float sphHarm;
  float attenuation;

  sourcesMatrix.clear();

  for (int k=0; k<=(sources - 1) ; k++){// Compute the spherical harmonic function
                                        // for each source till 3rd order taking
                                        // into account also the distance attenuation
                                        // and fill Y matrix colums

    azimuth = fmod(float(sourcesPos.at(k).at(0) + globalPos[0]), float(FTORAD_CONV));
    elevation = fmod(float(sourcesPos.at(k).at(1) + globalPos[1]), ...
                     ...float(FTORAD_CONV));

    distance = sourcesPos.at(k).at(2) + globalPos[2];
    attenuation = 1.000/distance;

    vector <float> column;
    sphHarm = 1.0*attenuation;
    column.push_back(sphHarm);
    sphHarm = sqrt(3.0)*cos(azimuth)*cos(elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = sqrt(3.0)*sin(azimuth)*sin(elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = sqrt(3.0)*sin(elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(24.0)/2)*cos(2*azimuth)*pow(cos(elevation),2)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(24.0)/2)*sin(2*azimuth)*pow(cos(elevation),2)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(24.0)/2)*cos(azimuth)*sin(2*elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(24.0)/2)*sin(azimuth)*sin(2*elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(5.0)/2)*((3*pow(sin(elevation),2)) - 1)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(35.0/8.0))*cos(3*azimuth)*pow(cos(elevation),3)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(35.0/8.0))*sin(3*azimuth)*pow(cos(elevation),3)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(7.0*15.0)/2)*cos(2*azimuth)*sin(elevation)* ...
              ... pow(cos(elevation),2)*attenuation;

    column.push_back(sphHarm);
    sphHarm = (sqrt(7.0*15.0)/2)*sin(2*azimuth)*sin(elevation)* ...
              ... pow(cos(elevation),2)*attenuation;

    column.push_back(sphHarm);
    sphHarm = (sqrt(21.0/8.0))*cos(azimuth)*cos(elevation)* ...
              ... (5*pow(sin(elevation),2) - 1)*attenuation;

    column.push_back(sphHarm);
    sphHarm = (sqrt(21.0/8.0))*sin(azimuth)*cos(elevation)* ...
              ... (5*pow(sin(elevation),2) - 1)*attenuation;

    column.push_back(sphHarm);
    sphHarm = (sqrt(7.0)/2)*sin(elevation)*(5*pow(sin(elevation),2) - 3)*attenuation;
    column.push_back(sphHarm);

    sourcesMatrix.push_back(column);
    column.clear();
  }

}
//------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------
void ambiSound::setGainsMatrix(int sources, int nfcFlag) {

  gainsMatrix.clear();
  float normFactor;                       //output normalization factor
  if(inverseFlag == 0)
    normFactor = 1/float(numLoudsp);

  // Matrix product (loudspMatrix*sourcesMatrix) = (C'/N)*Y or pinv(C)*Y
  if(nfcFlag == 0){                      // NFC filter enabled

    /* if NFC filter is enabled the scalar product between the i-th row of C' and
     * the j-th column of Y is split into 4 parts giving as result a 4 components
     * vector, one for each spherical harmonic order that will be filtered with a
     * different h(t) filter
     */

    for(int n=0; n<=(numLoudsp - 1); n++){// gains computation
      vector <float> row;

      for(int k=0; k<=(sources - 1); k++){
        float gain = 0.0;
        // 0th spherical harmonic group
        gain += loudspMatrix(n,0)*(sourcesMatrix.at(k).at(0));
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);

        // 1st spherical harmonics group
        for(int m=1; m<=3; m++){
          gain += (loudspMatrix(n,m))*(sourcesMatrix.at(k).at(m));
        }
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);

        // 2nd spherical harmonics group
        for(int m=4; m<=8; m++){
          gain += (loudspMatrix(n,m))*(sourcesMatrix.at(k).at(m));
        }
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);

        // 3rd spherical harmonics group
        for(int m=9; m<=15; m++){
          gain += (loudspMatrix(n,m))*(sourcesMatrix.at(k).at(m));
        }
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);
      }
      gainsMatrix.push_back(row);
      row.clear();
    }
  }
  else{                                   // usual scalar product - NFC filter
                                          // disabled

    for(int n=0; n<=(numLoudsp - 1); n++){// gains computation
      vector <float> row;

      for(int k=0; k<=(sources - 1); k++){
        float gain = 0.0;

        for(int m=0; m<=(numSphHarm - 1); m++){
          gain += (loudspMatrix(n,m))*(sourcesMatrix.at(k).at(m));
        }
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);
      }

      gainsMatrix.push_back(row);
      row.clear();
    }
  }

}
//-------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------
void ambiSound::loudspPosFileUpdate(){

  fstream outFile( "/Library/Audio/Plug-Ins/VST/loudspPos.dat", ios::in  | ios::out );
  // exit program if fstream cannot open file
  if ( !outFile )
  {
    cerr << "File could not be opened." << endl;
    exit( 1 );
  }

  outFile.seekp(0);
  outFile << dist;
  outFile.close();
}
//-------------------------------------------------------------------------------------



// AmbiSettings Methods (double methods)
//*****************************************
//-------------------------------------------------------------------------------------
void ambiSound::setdLoudspPos() {

  ifstream inFile( "/Library/Audio/Plug-Ins/VST/loudspPos.dat", ios::in );

  if ( !inFile )
  {
    cerr << "File could not be opened" << endl;
    exit( 1 );
  }

  inFile.seekg(0);
  inFile >> loudspDist;
  if(loudspDist > MAX_LOUDSP_DIST){
    loudspDist = MAX_LOUDSP_DIST;
    nfcSwitch = 1;
  }
  else{
    if(loudspDist < 1.00)
      loudspDist = 1.f;
  }

  inFile >> numLoudsp;
  if(numLoudsp > MAX_LOUDSP_NUM)
    numLoudsp = MAX_LOUDSP_NUM;
  else
    if(numLoudsp < 2)
      numLoudsp = 2;

  for (int n=0; n<=(numLoudsp -1); n++){
    double position;
    if (!inFile.eof()){
      inFile >> position;
      dLoudspPos.at(n).at(0) = fmod(position, 360) * DEGTORAD_CONV;
      inFile >> position;
      dLoudspPos.at(n).at(1) = fmod(position, 360) * DEGTORAD_CONV;
    }
  }
  setdLoudspMatrix();
}
//-------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------
void ambiSound::setdLoudspMatrix() {

  dLoudspMatrix.reset();
  dLoudspMatrix.set_size(numLoudsp, numSphHarm);

  for(int i=0; i<=(numLoudsp - 1); i++){
    dLoudspMatrix(i,0) = 1.0;

    dLoudspMatrix(i,1) = sqrt(3.0)*cos(loudspPos.at(i).at(0))* ...
                         ... cos(loudspPos.at(i).at(1));

    dLoudspMatrix(i,2) = sqrt(3.0)*sin(loudspPos.at(i).at(0))* ...
                         ... sin(loudspPos.at(i).at(1));

    dLoudspMatrix(i,3) = sqrt(3.0)*sin(loudspPos.at(i).at(1));

    dLoudspMatrix(i,4) = (sqrt(24.0)/2)*cos(2*loudspPos.at(i).at(0))* ...
                         ... pow(cos(loudspPos.at(i).at(1)),2);

    dLoudspMatrix(i,5) = (sqrt(24.0)/2)*sin(2*loudspPos.at(i).at(0))* ...
                         ... pow(cos(loudspPos.at(i).at(1)),2);

    dLoudspMatrix(i,6) = (sqrt(24.0)/2)*cos(loudspPos.at(i).at(0))* ...
                         ... sin(2*loudspPos.at(i).at(1));

    dLoudspMatrix(i,7) = (sqrt(24.0)/2)*sin(loudspPos.at(i).at(0))* ...
                         ... sin(2*loudspPos.at(i).at(1));

    dLoudspMatrix(i,8) = (sqrt(5.0)/2)*((3*pow(sin(loudspPos.at(i).at(1)),2)) - 1);

    dLoudspMatrix(i,9) = (sqrt(35.0/8.0))*cos(3*loudspPos.at(i).at(0))* ...
                         ... pow(cos(loudspPos.at(i).at(1)),3);

    dLoudspMatrix(i,10) = (sqrt(35.0/8.0))*sin(3*loudspPos.at(i).at(0))* ...
                          ... pow(cos(loudspPos.at(i).at(1)),3);

    dLoudspMatrix(i,11) = (sqrt(7.0*15.0)/2)*cos(2*loudspPos.at(i).at(0))* ...
                          ... sin(loudspPos.at(i).at(1))* ...
                          ... pow(cos(loudspPos.at(i).at(1)),2);
    dLoudspMatrix(i,12) = (sqrt(7.0*15.0)/2)*sin(2*loudspPos.at(i).at(0))* ...
                          ... sin(loudspPos.at(i).at(1))* ...
                          ... pow(cos(loudspPos.at(i).at(1)),2);

    dLoudspMatrix(i,13) = (sqrt(21.0/8.0))*cos(loudspPos.at(i).at(0))* ...
                          ... cos(loudspPos.at(i).at(1))* ...
                          ... (5*pow(sin(loudspPos.at(i).at(1)),2) - 1);

    dLoudspMatrix(i,14) = (sqrt(21.0/8.0))*sin(loudspPos.at(i).at(0))* ...
                          ... cos(loudspPos.at(i).at(1))* ...
                          ... (5*pow(sin(loudspPos.at(i).at(1)),2) - 1);

    dLoudspMatrix(i,15) = (sqrt(7.0)/2)*sin(loudspPos.at(i).at(1))* ...
                          ... (5*pow(sin(loudspPos.at(i).at(1)),2) - 3);
  }

  if(inverseFlag == 1){
    mat C = trans(dLoudspMatrix);
    dLoudspMatrix.reset();
    dLoudspMatrix = pinv(C);
  }
}
//------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------
void ambiSound::setdSourcesMatrix(int sources) {

  double azimuth ;
  double elevation ;
  double distance ;
  double sphHarm;
  double attenuation;
  double dist[2];

  dSourcesMatrix.clear();

  for (int k=0; k<=(sources - 1) ; k++){

    azimuth = fmod(double(sourcesPos.at(k).at(0) + globalPos[0]), ...
               ...double(FTORAD_CONV));

    elevation = fmod(double(sourcesPos.at(k).at(1) + globalPos[1]), ...
                 ...double(FTORAD_CONV));

    distance = sourcesPos.at(k).at(2) + globalPos[2];
    attenuation = 1.000/distance;

    vector <double> column;
    sphHarm = 1.0*attenuation;
    column.push_back(sphHarm);
    sphHarm = sqrt(3.0)*cos(azimuth)*cos(elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = sqrt(3.0)*sin(azimuth)*sin(elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = sqrt(3.0)*sin(elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(24.0)/2)*cos(2*azimuth)*pow(cos(elevation),2)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(24.0)/2)*sin(2*azimuth)*pow(cos(elevation),2)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(24.0)/2)*cos(azimuth)*sin(2*elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(24.0)/2)*sin(azimuth)*sin(2*elevation)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(5.0)/2)*((3*pow(sin(elevation),2)) - 1)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(35.0/8.0))*cos(3*azimuth)*pow(cos(elevation),3)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(35.0/8.0))*sin(3*azimuth)*pow(cos(elevation),3)*attenuation;
    column.push_back(sphHarm);
    sphHarm = (sqrt(7.0*15.0)/2)*cos(2*azimuth)*sin(elevation)* ...
               ... pow(cos(elevation),2)*attenuation;

    column.push_back(sphHarm);
    sphHarm = (sqrt(7.0*15.0)/2)*sin(2*azimuth)*sin(elevation)* ...
               ... pow(cos(elevation),2)*attenuation;

    column.push_back(sphHarm);
    sphHarm = (sqrt(21.0/8.0))*cos(azimuth)*cos(elevation)* ...
               ... (5*pow(sin(elevation),2) - 1)*attenuation;

    column.push_back(sphHarm);
    sphHarm = (sqrt(21.0/8.0))*sin(azimuth)*cos(elevation)* ...
               ... (5*pow(sin(elevation),2) - 1)*attenuation;

    column.push_back(sphHarm);
    sphHarm = (sqrt(7.0)/2)*sin(elevation)*(5*pow(sin(elevation),2) - 3)*attenuation;
    column.push_back(sphHarm);

    dSourcesMatrix.push_back(column);
    column.clear();
  }

}
//-----------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------
void ambiSound::setdGainsMatrix(int sources, int nfcFlag) {

  dGainsMatrix.clear();
  double normFactor;
  if(inverseFlag == 0)
    normFactor = 1/double(numLoudsp);

  if(nfcFlag == 0){
    for(int n=0; n<=(numLoudsp - 1); n++){
      vector <double> row;

      for(int k=0; k<=(sources - 1); k++){
        double gain = 0.0;
        gain += dLoudspMatrix(n,0)*(dSourcesMatrix.at(k).at(0));
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);

        for(int m=1; m<=3; m++){
          gain += (dLoudspMatrix(n,m))*(dSourcesMatrix.at(k).at(m));
        }
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);

        for(int m=4; m<=8; m++){
          gain += (dLoudspMatrix(n,m))*(dSourcesMatrix.at(k).at(m));
        }
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);

        for(int m=9; m<=15; m++){
          gain += (dLoudspMatrix(n,m))*(dSourcesMatrix.at(k).at(m));
        }
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);
      }

      dGainsMatrix.push_back(row);
      row.clear();
    }
  }
  else{
    for(int n=0; n<=(numLoudsp - 1); n++){
      vector <double> row;

      for(int k=0; k<=(sources - 1); k++){
        float gain = 0.0;

        for(int m=0; m<=(numSphHarm - 1); m++){
          gain += (dLoudspMatrix(n,m))*(dSourcesMatrix.at(k).at(m));
        }
        if(inverseFlag == 0)
          row.push_back(gain*normFactor);
        else
          row.push_back(gain);
      }

      dGainsMatrix.push_back(row);
      row.clear();
    }
  }
}
//------------------------------------------------------------------------------------



// NCF filtering methods
//*******************************************
//------------------------------------------------------------------------------------
void ambiSound::initNumCoeffMatrix(){

  for(int k=0; k<=(kNumInputs - 1); k++){
    vector <float> coefficients;
    vector < vector <float> > allCoefficients;
    coefficients.push_back(0.0); //setting default coefficients value to 0.0

    for(int i=0; i<=2; i++){
      coefficients.push_back(0.0);
      allCoefficients.push_back(coefficients);
    }

    numCoeff.push_back(allCoefficients);
    allCoefficients.clear();
    coefficients.clear();
  }
}
//------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------
void ambiSound::initdNumCoeffMatrix(){

  for(int k=0; k<=(kNumInputs - 1); k++){
    vector <double> coefficients;
    vector < vector <double> > allCoefficients;
    coefficients.push_back(0.0); //setting default coefficients value to 0.0
    for(int i=0; i<=2; i++){
      coefficients.push_back(0.0);
      allCoefficients.push_back(coefficients);
    }

    dNumCoeff.push_back(allCoefficients);
    allCoefficients.clear();
    coefficients.clear();
  }
}
//------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------
void ambiSound::setNfcDenFilterCoeff(float distance){

  // Denominator filter coefficients computation
  vector <float> zeroes;                // vector to store related zeroes
  vector <float> coefficients;          // vector to store coefficients
  float c = 340.f;                      // speed of sound
  float alpha = 4*sampleRate*dist/c;         // factor used in computation

  denCoeff.clear();
  coefficients.clear();

  float ar0 = 1.000 - (-2.000)/alpha;
  float ar1 = -(1.000 + (-2.000)/alpha);
  coefficients.push_back(ar1/ar0);      // 1 coefficient for H1(z) denominator
  denCoeff.push_back(coefficients);
  coefficients.clear();
  zeroes.clear();
  coefficients.clear();

  zeroes.push_back(-3.0000);            // f2(w) zeroes computation
  zeroes.push_back(pow(3.0000, 2) + pow(1.7321, 2));

  float ac0 = 1 - 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));
  float ac1 = -2 * (1 - zeroes.at(1)/(pow(alpha, 2)));
  float ac2 = 1 + 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));

  coefficients.push_back(ac1/ac0);      // 2 coefficients for H2(z) denominator
  coefficients.push_back(ac2/ac0);
  coefficients.push_back(ac0);          // actually doesn't take part on filtering
  denCoeff.push_back(coefficients);
  coefficients.clear();
  zeroes.clear();
  coefficients.clear();

  zeroes.push_back(-3.6778);            // f3(w) zeroes computation
  zeroes.push_back(pow(3.6778, 2) + pow(3.5088, 2));
  zeroes.push_back(-4.6444);

  ar0 = 1 - zeroes.at(2)/alpha;
  ar1 = -(1 + zeroes.at(2)/alpha);
  ac0 = 1 - 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));
  ac1 = -2 * (1 - zeroes.at(1)/(pow(alpha, 2)));
  ac2 = 1 + 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));

  float a_0 = ac0*ar0;
  float a_1 = ac1*ar0 + ac0*ar1;
  float a_2 = ac2*ar0 + ac1*ar1;
  float a_3 = ac2*ar1;

  coefficients.push_back(a_1/a_0);      // 3 coefficients for H3(z) denominator
  coefficients.push_back(a_2/a_0);
  coefficients.push_back(a_3/a_0);
  coefficients.push_back(a_0);          // actually doesn't take part on filtering
  denCoeff.push_back(coefficients);
  coefficients.clear();
}
//-----------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------
void ambiSound::setdNfcDenFilterCoeff(double distance){

  // Denominator filter coefficients computation
  vector <double> zeroes;              // vector to store related zeroes
  vector <double> coefficients;        // vector to store coefficients
  double c = 340.0000;                 // speed of sound
  double alpha = 4.0*sampleRate*dist/c;    // factor used in computation

  dDenCoeff.clear();
  coefficients.clear();

  double ar0 = 1.000 - (-2.000)/alpha;
  double ar1 = -(1.000 + (-2.000)/alpha);

  coefficients.push_back(ar1/ar0);      // 1 coefficient for H1(z) denominator
  dDenCoeff.push_back(coefficients);
  coefficients.clear();
  zeroes.clear();
  coefficients.clear();

  zeroes.push_back(-3.0000);            // f2(w) zeroes computation
  zeroes.push_back(pow(3.0000, 2) + pow(1.7321, 2));

  double ac0 = 1 - 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));
  double ac1 = -2 * (1 - zeroes.at(1)/(pow(alpha, 2)));
  double ac2 = 1 + 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));

  coefficients.push_back(ac1/ac0);      // 2 coefficients for H2(z) denominator
  coefficients.push_back(ac2/ac0);
  coefficients.push_back(ac0);          // actually doesn't take part on filtering
  dDenCoeff.push_back(coefficients);
  coefficients.clear();
  zeroes.clear();
  coefficients.clear();

  zeroes.push_back(-3.6778);            // f3(w) zeroes computation
  zeroes.push_back(pow(3.6778, 2) + pow(3.5088, 2));
  zeroes.push_back(-4.6444);

  ar0 = 1 - zeroes.at(2)/alpha;
  ar1 = -(1 + zeroes.at(2)/alpha);
  ac0 = 1 - 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));
  ac1 = -2 * (1 - zeroes.at(1)/(pow(alpha, 2)));
  ac2 = 1 + 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));

  double a_0 = ac0*ar0;
  double a_1 = ac1*ar0 + ac0*ar1;
  double a_2 = ac2*ar0 + ac1*ar1;
  double a_3 = ac2*ar1;

  coefficients.push_back(a_1/a_0);      // 3 coefficients for H3(z) denominator
  coefficients.push_back(a_2/a_0);
  coefficients.push_back(a_3/a_0);
  coefficients.push_back(a_0);          // actually doesn't take part on filtering
  dDenCoeff.push_back(coefficients);
  coefficients.clear();
}
//-----------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------
void ambiSound::setNfcNumFilterCoeff(int sourceIndex, float sourceDist){
  // Numerator filter coefficients coputation
  vector <float> zeroes;                // vector to store related zeroes
  vector <float> coefficients;          // vector to store coefficients
  float c = 340.f;                      // speed of sound
  float alpha = 4*sampleRate*sourceDist/c;   // factor used in computation

  float br0 = 1.000 - (-2.000)/alpha;
  float br1 = -(1.000 + (-2.000)/alpha);

  coefficients.push_back(br0/denCoeff.at(0).at(0));// 2 coefficients for H1(z)
  coefficients.push_back(br1/denCoeff.at(0).at(0));// numerator
  numCoeff.at(sourceIndex).at(0) = coefficients;   // numCoeff storing
  coefficients.clear();

  zeroes.push_back(-3.0000);                       // f2(w) zeroes computation
  zeroes.push_back(pow(3.0000, 2) + pow(1.7321, 2));

  float bc0 = 1 - 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));
  float bc1 = -2 * (1 - zeroes.at(1)/(pow(alpha, 2)));
  float bc2 = 1 + 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));

  coefficients.push_back(bc0/denCoeff.at(1).at(2));// 3 coefficients for H2(z)
  coefficients.push_back(bc1/denCoeff.at(1).at(2));// numerator
  coefficients.push_back(bc2/denCoeff.at(1).at(2));
  numCoeff.at(sourceIndex).at(1) = coefficients;   // numCoeff storing
  coefficients.clear();
  zeroes.clear();

  zeroes.push_back(-3.6778);                       // f3(w) zeroes computation
  zeroes.push_back(pow(3.6778, 2) + pow(3.5088, 2));
  zeroes.push_back(-4.6444);

  br0 = 1 - zeroes.at(2)/alpha;
  br1 = -(1 + zeroes.at(2)/alpha);
  bc0 = 1 - 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));
  bc1 = -2 * (1 - zeroes.at(1)/(pow(alpha, 2)));
  bc2 = 1 + 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));

  float b_0 = bc0*br0;
  float b_1 = bc1*br0 + bc0*br1;
  float b_2 = bc2*br0 + bc1*br1;
  float b_3 = bc2*br1;

  coefficients.push_back(b_0/denCoeff.at(2).at(3));// 4 coefficients for H3(z)
  coefficients.push_back(b_1/denCoeff.at(2).at(3));// numerator
  coefficients.push_back(b_2/denCoeff.at(2).at(3));
  coefficients.push_back(b_3/denCoeff.at(2).at(3));
  numCoeff.at(sourceIndex).at(2) = coefficients;   // numCoeff storing
  coefficients.clear();
  zeroes.clear();
}
//-------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------
void ambiSound::setdNfcNumFilterCoeff(int sourceIndex, double sourceDist){

  // Numerator filter coefficients coputation
  vector <double> zeroes;                 // vector to store related zeroes
  vector <double> coefficients;           // vector to store coefficients
  double c = 340.000;                     // speed of sound
  double alpha = 4.0*sampleRate*sourceDist/c;// factor used in computation

  double br0 = 1.000 - (-2.000)/alpha;
  double br1 = -(1.000 + (-2.000)/alpha);

  coefficients.push_back(br0/dDenCoeff.at(0).at(0));// 2 coefficients for H1(z)
  coefficients.push_back(br1/dDenCoeff.at(0).at(0));// numerator
  dNumCoeff.at(sourceIndex).at(0) = coefficients;   // dNumCoeff storing
  coefficients.clear();

  zeroes.push_back(-3.0000);                        // f2(w) zeroes computation
  zeroes.push_back(pow(3.0000, 2) + pow(1.7321, 2));

  double bc0 = 1 - 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));
  double bc1 = -2 * (1 - zeroes.at(1)/(pow(alpha, 2)));
  double bc2 = 1 + 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));

  coefficients.push_back(bc0/dDenCoeff.at(1).at(2));// 3 coefficients for H2(z)
  coefficients.push_back(bc1/dDenCoeff.at(1).at(2));// numerator
  coefficients.push_back(bc2/dDenCoeff.at(1).at(2));
  dNumCoeff.at(sourceIndex).at(1) = coefficients;   // dNumCoeff storing
  coefficients.clear();
  zeroes.clear();

  zeroes.push_back(-3.6778);                        // f3(w) zeroes computation
  zeroes.push_back(pow(3.6778, 2) + pow(3.5088, 2));
  zeroes.push_back(-4.6444);

  br0 = 1 - zeroes.at(2)/alpha;
  br1 = -(1 + zeroes.at(2)/alpha);
  bc0 = 1 - 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));
  bc1 = -2 * (1 - zeroes.at(1)/(pow(alpha, 2)));
  bc2 = 1 + 2*(zeroes.at(0)/alpha) + zeroes.at(1)/(pow(alpha, 2));

  double b_0 = bc0*br0;
  double b_1 = bc1*br0 + bc0*br1;
  double b_2 = bc2*br0 + bc1*br1;
  double b_3 = bc2*br1;

  coefficients.push_back(b_0/dDenCoeff.at(2).at(3));// 4 coefficients for H3(z)
  coefficients.push_back(b_1/dDenCoeff.at(2).at(3));// numerator
  coefficients.push_back(b_2/dDenCoeff.at(2).at(3));
  coefficients.push_back(b_3/dDenCoeff.at(2).at(3));
  dNumCoeff.at(sourceIndex).at(2) = coefficients;   // dNumCoeff storing
  coefficients.clear();
  allCoefficients.clear();
  zeroes.clear();
}
//------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------
float ambiSound::nfcFilter(int pastOutsIndex, float signal, int filtIndex){

  float filtSignal = 0.f;    // output initialization

  switch (filtIndex) {
    case 0:                  // 1st order harmonic components filtering
    {                        // convolution operation: signal(n)*h1(n)

      filtSignal = numCoeff.at(pastOutsIndex).at(filtIndex).at(0)*signal ...
                 ... + numCoeff.at(pastOutsIndex).at(filtIndex).at(1)* ...
                 ... pastInsMatrix.at(pastOutsIndex).at(0) ...
                 ... - denCoeff.at(filtIndex).at(0)* ...
                 ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(0);

      if(encodingOrder == 1){// past input samples update
        pastInsMatrix.at(pastOutsIndex).at(0) = signal;
      }
                             // past output samples update
      return pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(0) = filtSignal;
    }
      break;
    case 1:                  // 2nd order harmonic components filtering
    {                        // convolution operation: signal(n)*h2(n)

      filtSignal = numCoeff.at(pastOutsIndex).at(filtIndex).at(0)*signal ...
                 ... + numCoeff.at(pastOutsIndex).at(filtIndex).at(1)* ...
                 ... pastInsMatrix.at(pastOutsIndex).at(0) ...
                 ... + numCoeff.at(pastOutsIndex).at(filtIndex).at(2)* ...
                 ... pastInsMatrix.at(pastOutsIndex).at(1) ...
                 ... - denCoeff.at(filtIndex).at(0)* ...
                 ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(0) ...
                 ... - denCoeff.at(filtIndex).at(1)* ...
                 ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(1);

      if(encodingOrder == 2){// past input samples update
        pastInsMatrix.at(pastOutsIndex).at(1) = ...
                                     ... pastInsMatrix.at(pastOutsIndex).at(0);

        pastInsMatrix.at(pastOutsIndex).at(0) = signal;
      }
                             // past output samples update
      pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(1) = ...
                          ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(0);

      return pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(0) = filtSignal;
    }
      break;
    case 2:                  // 3rd order harmonic components filtering
    {                        // convolution operation: signal(n)*h3(n)

      filtSignal = numCoeff.at(pastOutsIndex).at(filtIndex).at(0)*signal ...
                 ... + numCoeff.at(pastOutsIndex).at(filtIndex).at(1)* ...
                 ... pastInsMatrix.at(pastOutsIndex).at(0) ...
                 ... + numCoeff.at(pastOutsIndex).at(filtIndex).at(2)* ...
                 ... pastInsMatrix.at(pastOutsIndex).at(1) ...
                 ... + numCoeff.at(pastOutsIndex).at(filtIndex).at(3)* ...
                 ... pastInsMatrix.at(pastOutsIndex).at(2) ...
                 ... - denCoeff.at(filtIndex).at(0)* ...
                 ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(0) ...
                 ... - denCoeff.at(filtIndex).at(1)* ...
                 ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(1) ...
                 ... - denCoeff.at(filtIndex).at(2)* ...
                 ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(2);

                             // past input samples update
      pastInsMatrix.at(pastOutsIndex).at(2) = ...
                                     ... pastInsMatrix.at(pastOutsIndex).at(1);

      pastInsMatrix.at(pastOutsIndex).at(1) = ...
                                     ... pastInsMatrix.at(pastOutsIndex).at(0);

      pastInsMatrix.at(pastOutsIndex).at(0) = signal;

                             // past output samples update
      pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(2) = ...
                          ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(1);

      pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(1) = ...
                          ... pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(0);

      return pastOutsMatrix.at(pastOutsIndex).at(filtIndex).at(0) = filtSignal;
    }
      break;
    default: return signal;
      break;
  }
}
//------------------------------------------------------------------------------------
```

```
//-------------------------------------------------------------------------------
double ambiSound::dNfcFilter(int dPastOutsIndex, double signal, int dFiltIndex){

  double filtSignal = 0.0;

  switch (dFiltIndex) {
    case 0:
    {
      filtSignal = dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(0)*signal ...
                 ... + dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(1)* ...
                 ... dPastInsMatrix.at(dPastOutsIndex).at(0) ...
                 ... - dDenCoeff.at(dFiltIndex).at(0)* ...
                 ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(0);

      if(encodingOrder == 1){
        dPastInsMatrix.at(dPastOutsIndex).at(0) = signal;
      }

      return dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(0) = filtSignal;
    }
      break;
    case 1:
    {
      filtSignal = dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(0)*signal ...
                 ... + dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(1)* ...
                 ... dPastInsMatrix.at(dPastOutsIndex).at(0) ...
                 ... + dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(2)* ...
                 ... dPastInsMatrix.at(dPastOutsIndex).at(1) ...
                 ... - dDenCoeff.at(dFiltIndex).at(0)* ...
                 ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(0) ...
                 ... - dDenCoeff.at(dFiltIndex).at(1)* ...
                 ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(1);

      if(encodingOrder == 2){
        dPastInsMatrix.at(dPastOutsIndex).at(1) = ...
                                 ...dPastInsMatrix.at(dPastOutsIndex).at(0);

        dPastInsMatrix.at(dPastOutsIndex).at(0) = signal;
      }

      dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(1) = ...
                  ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(0);

      return dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(0) = filtSignal;
    }
      break;
    case 2:
    {
      filtSignal = dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(0)*signal ...
                 ... + dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(1)* ...
                 ... dPastInsMatrix.at(dPastOutsIndex).at(0) ...
                 ... + dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(2)* ...
                 ... dPastInsMatrix.at(dPastOutsIndex).at(1) ...
                 ... + dNumCoeff.at(dPastOutsIndex).at(dFiltIndex).at(3)* ...
                 ... dPastInsMatrix.at(dPastOutsIndex).at(2) ...
                 ... - dDenCoeff.at(dFiltIndex).at(0)* ...
                 ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(0) ...
                 ... - dDenCoeff.at(dFiltIndex).at(1)* ...
                 ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(1) ...
                 ... - dDenCoeff.at(dFiltIndex).at(2)* ...
                 ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(2);

      dPastInsMatrix.at(dPastOutsIndex).at(2) = ...
                                ... dPastInsMatrix.at(dPastOutsIndex).at(1);

      dPastInsMatrix.at(dPastOutsIndex).at(1) = ...
                                ... dPastInsMatrix.at(dPastOutsIndex).at(0);
      dPastInsMatrix.at(dPastOutsIndex).at(0) = signal;

      dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(2) = ...
                        ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(1);

      dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(1) = ...
                        ... dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(0);

      return dPastOutsMatrix.at(dPastOutsIndex).at(dFiltIndex).at(0) = filtSignal;
    }
      break;
    default: return signal;
      break;
  }
}
//-----------------------------------------------------------------------------------
```

Bibliography

[1] Aurelio Uncini. *Audio Digitale*, McGraw-Hill, 2006, ch. 10.

[2] Jérôme Daniel, *Représentation des champs acoustiques, application à la trasmission et à la reproduction de scènes sonores complexes dans un contexte multmédia*, PhD Thesis, University of Paris 6, 2001. http://gyronymo.free.fr/audio3D/download_Thesis_PwPt.html

[3] Michael Gerzon, *Multidirectional sound reproduction systems*, 1976.

[4] Jérôme Daniel, Rozenn Nicol and Sébastian Moreau, AES, *Further Investigations of Higher Order Ambisonics and Wavefield Synthesis for Holophonic Sound Imaging*, in AES 114th Convention, 2003. http://gyronymo.free.fr/audio3D/publications/AES114-WFS_HOA.pdf

[5] Jérôme Daniel and Sébastian Moreau, AES, *Further Study of Sound Field Coding with Higher Order Ambisonics*, in AES 116th Convention, 2004. http://gyronymo.free.fr/audio3D/publications/AES116%20High-Passed%20HOA.pdf

[6] Jérôme Daniel, *Spatial Sound Encoding Including Near Field Effect: Introducing Distance Coding Filters and a Viable, New Ambisonics Format*, in AES 23° International Conference, 2003. http://gyronymo.free.fr/audio3D/publications/AES23%20NFC%20HOA.pdf

[7] David T. Blackstock. *Fundamentals of Physical Acoustics.* Wiley-Interscience, 2000.

[8] Salvo Daniele Valente. *Notes on Ambisonics Surround Sound.* (Unpublished) Politecnico di MIlano, 2010.

[9] Michael Kratschmer and Rudolf Rabenstein, *IMPLEMENTING AMBISONICS ON A 48 CHANNEL CIRCULAR LOUDSPEAKER ARRAY*, in Ambisonics Symposium, 2009. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.158.9285

[10] R. Rabenstein and S. Spors. *Sound Field Reproduction*. Springer Handbook of Speech Processing. Springer, 2008.

[11] Jan C. Schacher, *SEVEN YEARS OF ICST AMBISONICS TOOLS FOR MAXMSP - A BRIEF REPORT*,2010. http://ambisonics10.ircam.fr/drupal/files/proceedings/poster/P1_7.pdf

[12] York University Music Technology Group, *Home Page for Ambisonics and Related 3-D Audio Research*, 2005. http://www.york.ac.uk/inst/mustech/3d_audio/

[13] Dave Malham, *Higher order Ambisonic systems*, 2003. http://www.york.ac.uk/inst/mustech/3d_audio/higher_order_ambisonics.pdf

[14] Dave Malham, *VST Ambisonic Tools*, 2002. http://www.dmalham.freeserve.co.uk/vst_ambisonics.html

[15] Steinberg. *Virtual Studio Technology Plug-In Specification 2.0 Software Development Kit*. Steinberg, 1999.

[16] A&G Soluzioni Digitali, *Website Home Page*, 2010. http://www.aegweb.it/homex.html

[17] Daniel Courville, *Daniel Courville Ambisonic Studio Web Page*, 2010. http://www.radio.uqam.ca/ambisonic/b2x.html#b2verb

[18] Thomas Chen, *Thomas Chen's software Web Page*, 2007. http://www.ambisonic.net/tchen.html

[19] Wikipedia, *Ambisonics Wiki Page*, 2011. http://en.wikipedia.org/wiki/Ambisonics

[20] Wikipedia, *Virtual Studio Technology Wiki Page*, 2011. http://en.wikipedia.org/wiki/Virtual_Studio_Technology

[21] Steinberg Media Technologies GmbH, *Our Technologies Page*, 2011. http://www.steinberg.net/en/company/technologies.html

[22] Graham Wakefield, *Third-Order Ambisonic Extensions for MaxMSP With Musical Applications*, 2011. http://ucsb.academia.edu/grrrwaaa/Papers/249822/Third-Order_Ambisonic_Extensions_for_MaxMSP_With_Musical_Applications

[23] Geoffrey Francis, *Up and Running: A REAPER User Guide v 3.75*, 2011. http://www.cockos.com/reaper/resources.php

[24] Conrad Sanderson, NICTA, *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*, Technical Report, 2010.