UNIVERSITY OF DERBY

AN INVESTIGATION INTO THE REAL-TIME MANIPULATION AND CONTROL OF THREE-DIMENSIONAL SOUND FIELDS

Bruce Wiggins

Doctor of Philosophy 2004

Contents

List of Figures

List of Equations

List of Tables

Acknowledgements

Many thanks must go to my supervisors, Iain Paterson-Stephens and Richard Thorn for their greatly appreciated input throughout this research. I thank Stuart Berry and Val Lowndes for introducing me to the world of heuristic search methods and Peter Lennox, Peter Schillebeeckx and Howard Stratton who have been constant sources of opinion, knowledge and wisdom on various areas of my project. Finally, I must thank Rachel, for keeping my feet on the ground, keeping me sane, and putting up with the, seemingly, everlasting write-up period.

Abstract

This thesis describes a system that can be used for the decoding of a three dimensional audio recording over headphones or two, or more, speakers. A literature review of psychoacoustics and a review (both historical and current) of surround sound systems is carried out. The need for a system which is platform independent is discussed, and the proposal for a system based on an amalgamation of Ambisonics, binaural and transaural reproduction schemes is given. In order for this system to function optimally, each of the three systems rely on providing the listener with the relevant psychoacoustic cues. The conversion from a five speaker ITU array to binaural decode is well documented but pair-wise panning algorithms will not produce the correct lateralisation parameters at the ears of a centrally seated listener. Although Ambisonics has been well researched, no one has, as yet, produced a psychoacoustically optimised decoder for the standard irregular five speaker array as specified by the ITU as the original theory, as proposed by Gerzon and Barton (1992) was produced (known as a Vienna decoder), and example solutions given, before the standard had been decided on. In this work, the original work by Gerzon and Barton (1992) is analysed, and shown to be suboptimal, showing a high/low frequency decoder mismatch due to the method of solving the set of non-linear simultaneous equations. A method, based on the Tabu search algorithm, is applied to the Vienna decoder problem and is shown to provide superior results to those shown by Gerzon and Barton (1992) and is capable of producing multiple solutions to the Vienna decoder problem. During the write up of this report Craven (2003) has shown how 4th order circular harmonics (as used in Ambisonics) can be used to create a frequency independent panning law for the five speaker ITU array, and this report also shows how the Tabu search algorithm can be used to optimise these decoders further. A new method is then demonstrated using the Tabu search algorithm coupled with lateralisation parameters extracted from a binaural simulation of the Ambisonic system to be optimised (as these are the parameters that the Vienna system is approximating). This method can then be altered to take into account head rotations directly which have been shown as an important psychoacoustic parameter in the localisation of a

sound source (Spikofski *et al.,* 2001) and is also shown to be useful in differentiating between decoders optimised using the Tabu search form of the Vienna optimisations as no objective measure had been suggested. Optimisations for both Binaural and Transaural reproductions are then discussed so as to maximise the performance of generic HRTF data (i.e. not individualised) using inverse filtering methods, and a technique is shown that minimises the amount of frequency dependant regularisation needed when calculating cross-talk cancellation filters.

Chapter 1 - Introduction

1.1 Background

Surround sound has quickly become a consumer ‘must have’ in the audio world, due, in the main part, to the advent of the Digital Versatile Disk, Super Audio CD technology and the computer gaming industry. It is generally taken to mean a system that creates a sound field that surrounds the listener. Or, to be put another way, it is trying to recreate the illusion of the ‘you are there’ experience. This is in contrast to the stereophonic reproduction that has been the standard for many years, which creates a ‘they are here’ illusion (Glasgal, 2003c).

The direction that the surround sound industry has taken, when referring to format and speaker layout, has depended, to some extent, on which system the technology has been used for. As already mentioned, two main streams of surround sound development are taking place:

- The DVD Video/Audio industry can be broadly categorised as follows:
    - These systems are predicated around audio produced for a standard 5 speaker (plus sub-woofer, or low frequency effects channel) layout as described in the ITU standard ‘ITU-R BS.775-1’.
    - Few DVD titles deviate from this standard as most DVD players are hardware based and, therefore, of a fixed specification.
    - Some processors are available with virtual speaker surround (see crosstalk cancelled systems) and virtual headphone surround systems.
    - Recording/panning techniques are not fixed and many different systems are utilised including:
        - Coincident recording techniques
        - Spaced recording techniques
        - Pair-wise panned using amplitude or time or a combination of the two.

- The computer gaming industry can be broadly categorised as follows:
    - Number and layout of speakers are dictated by the soundcard installed in the computer. Typically:
        - Two speakers – variable angular spacing.
        - Four speakers – based on a Quadraphonic arrangement or the ITU five speaker layout without a centre speaker.
        - Five speakers – based on ITU-R BS.755-1 layout.
        - Six speakers – same as above but with a rear centre speaker.
        - Seven speakers – typically, same as five speakers with additional speakers at +/- 90^0.
    - Two channel systems rely on binaural synthesis (using head related transfer functions) and/or crosstalk cancellation principles using:
        - Binaural/Transaural simulation of a more than two speaker system.
        - HRTF simulation of sources.
    - More than two speaker systems generally use pair-wise panning algorithms in order to place sounds.

Both of the above viewpoints overlap, mainly due to the need for computers to be compatible with DVD audio/video. However, the computer gaming industry has started moving away from five speaker surround with 7.1 surround sound being the standard on most new PCs.

The systems described above all co-exist, often being driven by the same carrier signals. For example, all surround sound output on a DVD is derived from the 5.1 speaker feeds that are stored on the actual disk. So headphone surround processing can be carried out by simulating the 5.1 speaker array binaurally, and two speaker virtual surround systems can be constructed by playing a crosstalk cancelled version of the binaural simulation. In the same fashion many crosstalk cancelled and binaural decodes provided by the audio hardware in computers is driven by the signal that would normally be sent to the 4, 5, 6 or 7 speaker array with other cards choosing to process the sound effects and music directly with individual pairs of head related transfer functions (see CMedia, N.D. and Sibbald, A., 2000 for examples of these two systems).

The above situation sounds ideal from a consumer choice, point of view, but there are a number of issues with the systems, described above, as a whole. The conversion from multi-speaker to binaural/transaural (crosstalk cancelled) system assumes that a, normally pair-wise panned, speaker presentation will provide the ear/brain system with the correct cues needed for the listener to experience a truly immersive, psychoacoustically correct aural presentation. However, the five speaker layout, as specified by the ITU, was not meant to deliver this, and is predicated on a stable 60^0 frontal image, with the surround speakers used only for effects and ambience information. This is, of course, not a big issue for films, but as computer games and audio only presentations are based around the same, five speaker, layout, this is not ideal. Computer games often do not want to give a preference to any particular direction with the surround sound audio experience hopefully providing extra cues to the game player in order to give them a more accurate auditory ‘picture’ of the environment around them and music presentations often want to try and simulate the space that the music was recorded in as accurately as possible, which will include material from the rear and sides of the listener.

A less obvious problem with PC based audio systems is that although the final encoding and decoding of the material is handled by the audio hardware (as most sound sources for games are panned in real-time), and so it is the hardware that dictates what speaker/headphone setup to use, inserting pre-recorded surround sound music can be problematic as no speaker layout can be assumed. Conversely for the DVD systems, the playing of music is, obviously, well catered for but only as long as it is presented in the right format. Converting from a 5.1 to a 7.1 representation, for example, is not necessarily a trivial matter and so recordings designed for a 5.1 ITU setup cannot easily use extra speakers in order to improve the performance of the recording. This is especially true as no panning method can be assumed after the discrete speaker feeds have been derived and stored on the DVD.

The problems described above can be summarised as follows:

- 5.1 DVD recordings cannot be easily ‘upmixed’ as:
    - No panning/recording method can be assumed.
    - Pair-wise panned material cannot be upmixed to another pair-wise panned presentation (upmixing will always increase the number of speakers active when panning a single source).
- Computer gaming systems produce surround sound material ‘on-the-fly’ and so pre-recorded multi-channel music/material can be difficult to add as no presentation format can be assumed.
- Both systems, when using virtual speaker technology (i.e. headphone or cross talk cancelled simulation of a multi-speaker representation) are predicated on the original speaker presentation delivering the correct psychoacoustical cues to the listener. This is not the case for the standard, pair-wise panned method which relies on this crosstalk to present the listener with the correct psychoacoustic cues (see Blumlein’s Binaural Sound in chapter 3.2.2).

These problems stem, to some extent, from the lack of separation between the encoding and the decoding of the material, with the encode/decode process generally taken as a whole. That is the signals that are stored, used and listened to are always derived from speaker feeds. This then leads to the problem of pre-recorded pieces needing to either be re-mixed and/or re-recorded if the number or layout of the speakers is to be changed.

1.2 The Research Problem

How can the encoding be separated from the decoding in audio systems, and how can this system be decoded in a psychoacoustically aware manner for multiple speakers or headphone listening?

While the transfer from multiple speaker systems to binaural or crosstalk cancelled systems is well documented, the actual encoding of the material must be carried out in such a way so as to ensure:

- Synthesised or recorded material can be replayed over different speaker arrays.
- The decoded signal should be based on the psychoacoustical parameters with which humans hear sound thus allowing a more meaningful conversion from a multi-speaker signal to binaural or crosstalk cancelled decode.

The second point would be best catered for using a binaural recording or synthesis technique. However, upmixing from a two channel binaural recording to a multi-speaker presentation can not be carried out in a satisfactory way, with the decoder for such a system needing to mimic all of the localisation features of the ear/brain system in order to correctly separate and pan sounds into the correct position. For this reason, it is a carrier signal based on a multi-speaker presentation format that will be chosen for this system.

Many people sought to develop a multi-speaker sound reproduction system as early as the 1900s, with work by Bell Labs trying to create a truly 'they are here' experience using arrays of loudspeakers in front of the listener. Perhaps they were also striving for a true volume solution which, to a large extent, has still not been achieved (except in a system based on Bells' early work called wavefield synthesis, see Chapter 3). However, it was Alan Blumlein's system, binaural sound, that was to form the basis for the system we now know as stereo, although it was to be in a slightly simplified form than the system that Blumlein first proposed.

The first surround sound standard was the Quadraphonic format. This system was not successful due to the fact that it was based on the simplified stereo technique and so had some reproduction problems coupled with Quadraphonics having a number of competing standards. At around the same time a number of researchers, including Michael Gerzon, recognised these problems and proposed a system that took more from Blumlein's original idea. This new system was called Ambisonics, but due to the failings of the Quadraphonic system, interest in this new surround sound format was poor.

Some of the benefits of the Ambisonics system are now starting to be realised and it is this system that was used as the basis of this investigation.

1.3 Aims and Objectives of the Research

- Develop a flexible multi-channel sound listening room capable of the auditioning of several speaker positioning formats simultaneously.
- Using the Matlab/Simulink software combined with a PC and a multi-channel sound card, create a surround sound toolbox enabling a flexible and quick development environment used to encode/decode surround sound systems in real-time.
- Carry out an investigation into the Ambisonic surround sound system looking at the optimisation of the system for different speaker configurations, specifically concentrating on the ITU standard five speaker layout.
- Carry out an investigation into Binaural and Transaural sound reproduction and how the conversion from Ambisonics to these systems can be achieved.
- Propose a hybrid system consisting of a separate encode and decode process, making it possible to create a three-dimensional sound piece which can be reproduced over headphones or two or more speakers.
- Create a real-time implementation of this system.

At the beginning of this project, a multi-channel sound lab was setup so different speaker layouts and decoding schemes could be auditioned. The lab contained speakers placed in a number of configurations so that experiments and testing would be quick to set up, and flexible. It consisted of a total of fourteen speakers as shown in Figure 1.1.

Three main speaker system configurations have been incorporated into this array:

- A regularly spaced, eight speaker, array
- A standard ITU-R BS.755-1 five speaker array
- A closely spaced front pair of speakers

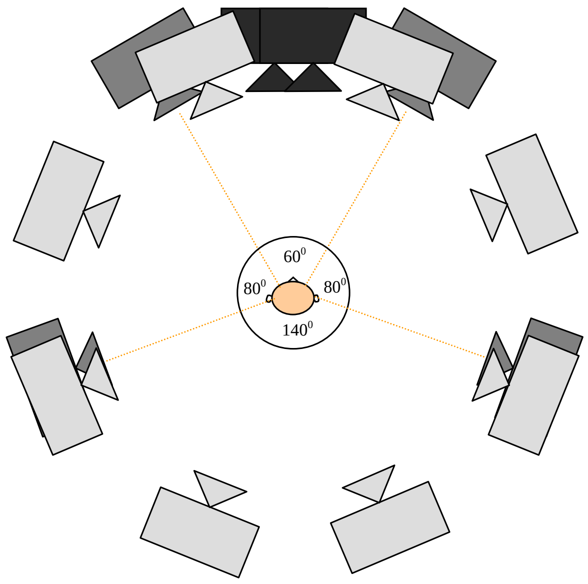


Figure 1.1 Speaker configuration developed in the multi-channel surround sound laboratory

The system, therefore, allows the main forms of multi-speaker surround formats to be accessed simultaneously. A standard Intel® Pentium® III (Intel Corporation, 2003) based PC was used in combination with a Soundscape® Mixtreme® (Sydec, 2003) sixteen channel sound card. This extremely versatile setup was originally used with the Matlab®/Simulink® program (The MathWorks, 2003), which was possible after rewriting Simulinks 'To' and 'From Wave Device' blocks to handle up to sixteen channels of audio simultaneously and in real-time (the blocks that ship with the product can handle a maximum of two channels of audio, see Chapter 5). This system was then superseded by custom C++ programs written for the Microsoft Windows operating system (Microsoft Corporation, 2003), as greater CPU efficiency could be utilised this way, which is an issue for filtering and other CPU intensive tasks.

Using both Matlab/Simulink and dedicated C++ coded software it was possible to both test, evaluate and apply optimisation techniques to the decoding of an Ambisonics based surround sound system and to this end the aim of this project was to develop a surround sound format, based on the hierarchical nature of B-format, the signal carrier of Ambisonics, that was able to be decoded to headphones and speakers, and investigate and optimise these systems using head related transfer functions.

1.4 Structure of this Report

This report is split into three main sections as listed below:

1. Literature review and discussion:
    a. Chapter 2 – Psychoacoustics and Spatial Sound Perception
    b. Chapter 3 – Surround Sound Systems
2. Surround sound format proposal and system development research
    a. Chapter 4 – Hierarchical Surround Sound Format
    b. Chapter 5 – Surround Sound Optimisation Techniques
3. System implementation and signal processing research
    a. Chapter 6 – Implementation of a Hierarchical Surround Sound System.

Sections two and three detail the actual research and development aspects of the project with section one giving a general background into surround sound and the psychoacoustic mechanisms that are used to analyse sounds heard in the real world (that is, detailing the systems that must be fooled in order to create a realistic, immersive surround sound experience).

Chapter 2 - Psychoacoustics and Spatial Sound Perception

2.1 Introduction

This Chapter contains a literature review and discussion of the current thinking and research in the area of psychoacoustics and spatial sound perception. This background research is important as it is impossible to investigate and evaluate surround systems objectively without first knowing how our brain processes sound, as it is this perceptual system that we are aiming to fool. This is particularly true when optimisations are to be sought after, as unless it is known what parameters we are optimising for, only subjective and empirically derived alterations can be used to improve a system's performance or, in the same way, help us explain why a system is not performing as we would have hoped.

2.2 Lateralisation

One of the most important physical rudiments of the human hearing system is that it possesses two separate data collection points, that is, we have two ears. Many experiments have been conducted throughout history (for a comprehensive reference on these experiments see Blauert (1997) and Gulick *et al.* (1989)) concluding that the fact that we hear through two audio receivers at different positions on the head is important in the localisation of the sounds (although our monaural hearing capabilities are not to be under-estimated).

If we observe the situation shown in Figure 2.1 where a sound source (speaker) is located in an off-centre position, then there are a number of differences between the signals arriving at the two ears, after travelling paths 'a' and 'b'. The two most obvious differences are:

- The distances travelled by the sounds arriving at each ear are different (as the source is closer to the left ear).
- The path to the further away of the two ears ('b') has the added obstacle of the head.

These two separate phenomena will manifest themselves at the ears of the listener in the form of time and level differences between the two incoming signals and, when simulated correctly over headphones, will result in an effect called lateralisation. Lateralisation is the sensation of a source being inside the listener's head. That is, the source has a direction, but the distance of the listener to the source is perceived as very small.

If we take the speed of sound as 342 ms^{-1} and the diameter of an average human head (based on a sphere, with the ears at 90^0 and 270^0 of that sphere) as 18 cm, then the maximum path difference between the left and right ears (d) is half the circumference of that sphere, given by equation (2.1).

$$d = \Pi r = \Pi \times 0.09 = 0.28274\text{m} \tag{2.1}$$

where d is half the circumference of a sphere

r is the radius of the sphere

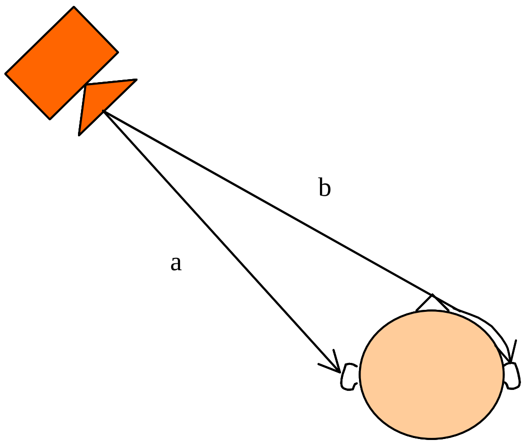


Figure 2.1 **The two paths, 'a' and 'b', that sound must travel from a source at 45^0 to the left of a listener, to arrive at the ears.**

Taking the maximum circumferential distance between the ears as 28 cm, as shown in equation (2.1), this translates into a maximum time difference between the sounds arriving at the two ears of 0.83 ms. This time difference is termed the Interaural Time Difference (I.T.D.) and is one of the cues used by the ear/brain system to calculate the position of sound sources.

The level difference between the ears, termed I.L.D. (Interaural Level Difference) is not, substantially, due to the extra distance travelled by the sound. The main difference here is obtained from the shadowing effect of the head. So, unlike I.T.D., which will be the same for all frequencies (although the phase difference is *not* constant), I.L.D. is frequency dependent due to diffraction. As a simple rule of thumb, any sound that has a wavelength larger than the diameter of the head will tend to be diffracted around and any sound with a wavelength shorter than the diameter of the head will tend to be attenuated causing a low pass filtering effect. The frequency corresponding to the wavelength equal to the diameter of the head is shown in equation (2.2).

$$f = {}^{1}\!/_{0.18} \times 342 = 1.89kHz \qquad \textbf{(2.2)}$$

where 0.18 is the diameter of the head.

There is, however, a smooth transition from low to high frequencies that means that the attenuation occurring at the opposite ear will increase with frequency. A graph showing an approximation of the I.L.D. of a sphere, up to 2 kHz, is shown in Figure 2.2 (equations taken from Duda (1993)). This figure shows the increasing I.L.D. with increasing frequency and angle of incidence.

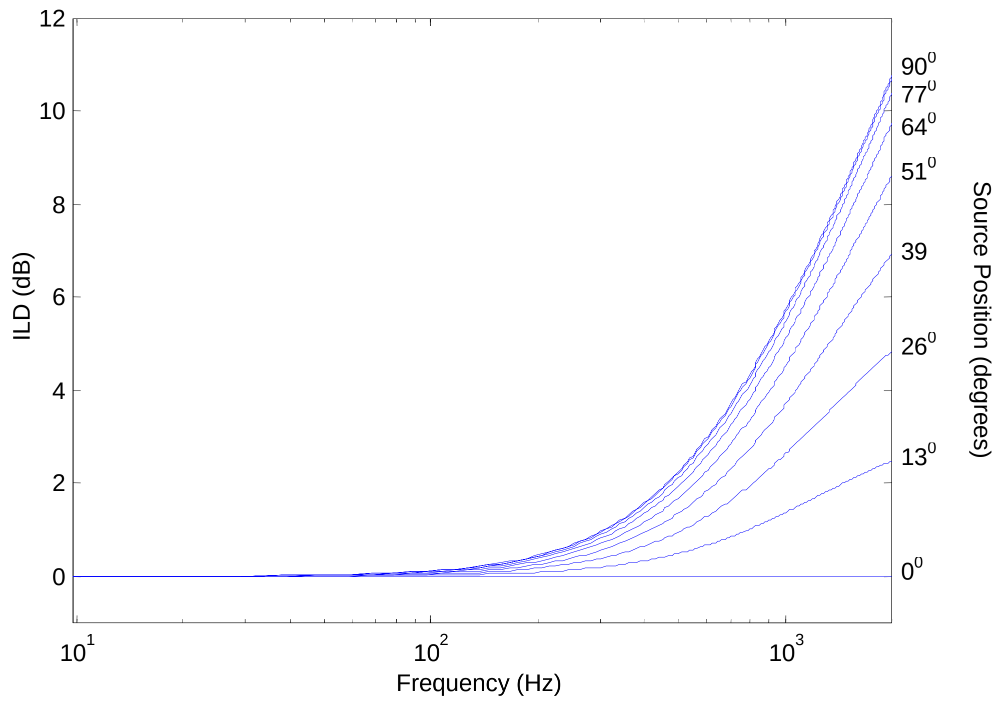


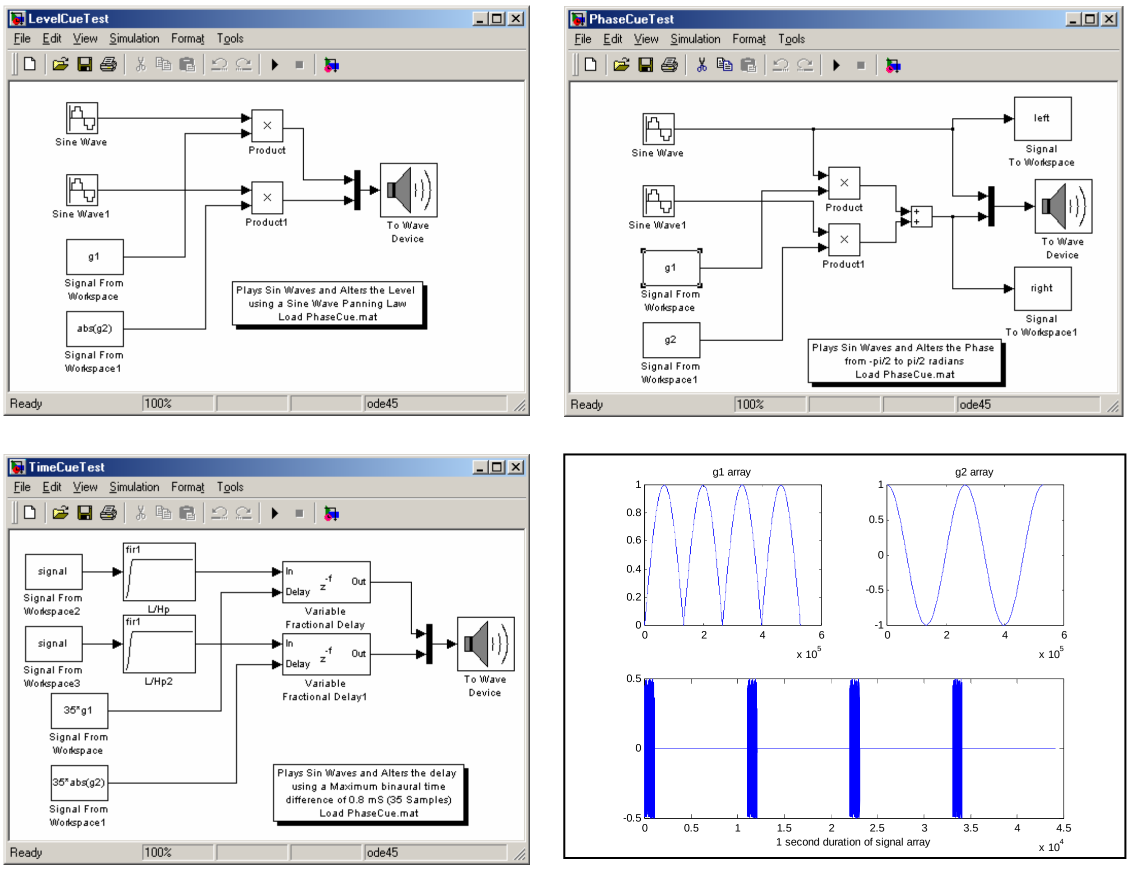
Figure 2.2 Increasing I.L.D. with frequency and angle of incidence.

2.2.1 Testing the Lateralisation Parameters.

A few simple experiments can be set up in order to test the working frequency ranges, and the effectiveness of the sound source position artefacts described above. The two cues presented, I.L.D. and I.T.D. actually result in three potential auditory cues. They are:

- An amplitude difference between the two ears (I.L.D).
- A time difference between the two ears (I.T.D).
- A phase difference between the sounds at the ears (I.T.D.).

Simulink models that can be used to test these three localisation parameters, under headphone listening conditions, are shown in Figure 2.3. Several data sources are utilised in these models (also shown in Figure 2.3) and are discussed below.



Plays Sin Waves and Alters the Level using a Sine Wave Panning Law Load PhaseCue.mat

Plays Sin Waves and Alters the Phase from -pi/2 to pi/2 radians Load PhaseCue.mat

Plays Sin Waves and Alters the delay using a Maximum binaural time difference of 0.8 mS (35 Samples) Load PhaseCue.mat

Figure 2.3 **Simulink models showing tests for the three localisation cues provided by I.L.D. and I.T.D..**

Arrays ‘g1’ and ‘g2’ are a rectified sine wave and a cosine wave, and are used to represent an amplitude gain, a phase change or a time delay. In order for the various lateralisation cues to be tested, the models must be configured as described below:

- Level Difference – If ‘g1’ is taken as the gain of the left channel, and a rectified version of ‘g2’ is used for the gain of the right channel, then the sound source is level panned smoothly between the two ears, and this is what the listener perceives, at any given frequency.
- Phase Difference – A sine wave of any phase can be constructed using a mixture of a sine wave at 0^0 and a sine wave at 90^0 (a cosine). So applying the gains ‘g1’ and ‘g2’ to a sine and a cosine wave which are then summed, will create a sine wave that changes phase from -$\Pi/2$ to $\Pi/2$. At low frequencies this test will tend to pan the sound between the two ears. However, as the frequency increases the phase difference between the signals has less effect. For example, at 500 Hz the sounds lateralises very noticeably. At 1000 Hz only a very slight source movement is

perceivable and at 1500 Hz, although a slight change in timbre can be noted, the source does not change position.

- Time Difference – For this test a broad band random noise source was used so that the sound contained many transients. The source was also pulsed on and off (see Figure 2.3) so that as the time delay between the two ears changed the pulsed source would not move significantly while it was sounding. The time delay was achieved using two fractional delay lines, using 'g1' and a rectified 'g2' scaled to give a delay between the ears varying from –0.8 ms to 0.8 ms (+/- 35 samples at 44.1 kHz), which roughly represents a source deflection of $–90^0$ to 90^0 from straight ahead. Slight localisation differences seem to be present up to a higher frequency than with phase differences, but most of this cue's usefulness seems to disappear after around 1000 Hz.

It is clear that the phase and time differences between the two ears of the listener are related, but they should be considered as two separate cues to the position of a sound source. For example if we take a 1 kHz sine wave, the period is equal to 0.001 seconds. If this sound is delayed by 0.00025 seconds, the resulting phase shift will be 90^0. However, if the sine wave is delayed by 0.00125 seconds the phase shift seen will be 450^0. As the ears are not able to detect absolute phase shift they must compare the two ears' signals, which will still give a phase shift of 90^0 as shown in Figure 2.4. It is also apparent from Figure 2.4 that if a sound of a different frequency is used, the same time delay will give a different phase difference between the ears. As frequency increases the phase change due to path differences between the ears becomes greater, but once the phase difference between the two ears is more than 180^0 then the brain can no longer decide which signal is lagging and the cue becomes ambiguous (Gulick, 1989).

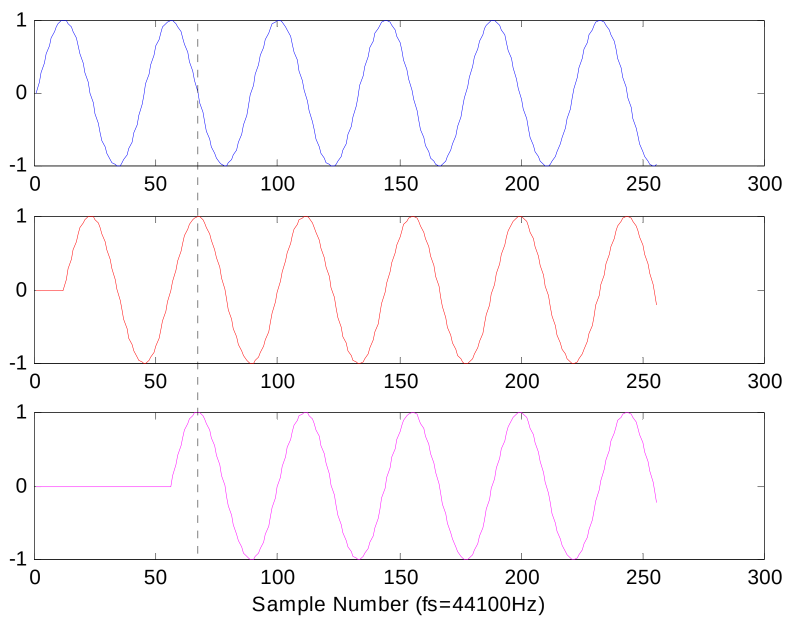


Figure 2.4 **Relative phase shift for a 1 kHz sine wave delayed by 0.00025 and 0.00125 seconds**

The difference between time and phase cues is significant, as they will need to be utilised by the ear/brain system for different localisation situations. If we take the situation where the listener is trying to localise a continuous sine wave tone, the time of arrival cues seen in Figure 2.4 will be not be present and only phase and amplitude cues can be used (it should also be noted that a pure sine wave tone can be a difficult source to locate anyway). Alternatively, if the listener is trying to localise a repeating ‘clicking’ sound, then the time of arrival cues due to source position will be present. Also, it has been found that, even for higher frequency sounds, time/phase cues can still be utilised with regards to the envelope of the sound arriving at the head, as shown in Figure 2.5.

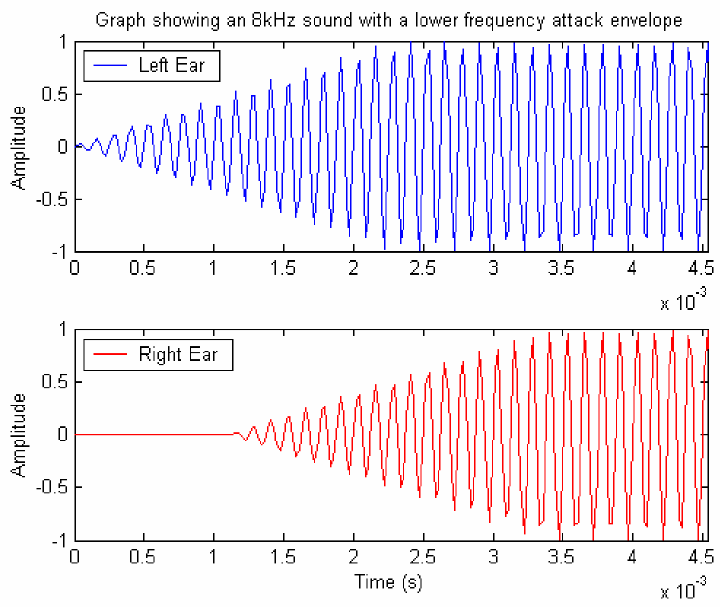


Figure 2.5 An 8 kHz tone with a low frequency attack envelope

Using a combination of the cues described above, a good indication of the angle of incidence of an incoming sound can be constructed, but the sound will be perceived as inside the head with the illusion of sounds coming from behind the listener being more difficult to achieve. The reason for this is the so-called 'Cone of Confusion' (Begault, 2000). Any sound that is coming from a cone of directions (shown as grey circles in Figure 2.6) will have the same level, phase and time differences associated with it making the actual position of the source potentially ambiguous.

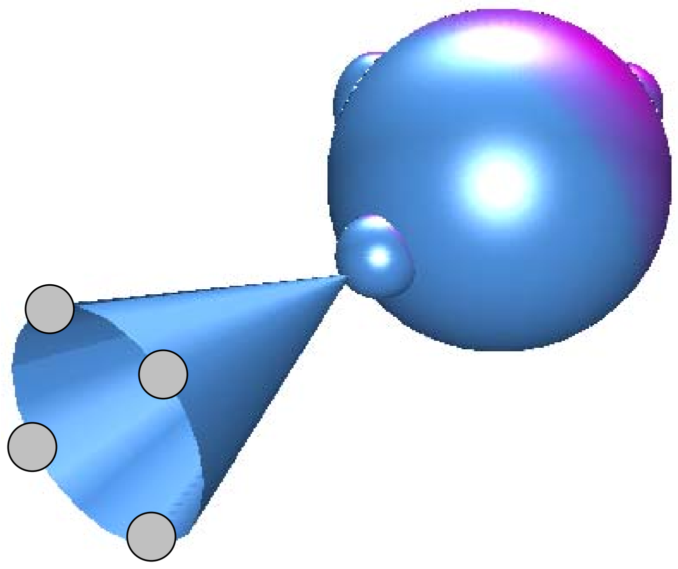


Figure 2.6 Cone of Confusion – Sources with same I.L.D. and I.T.D. are shown as grey circles.

So how does the ear/brain system cope with this problem? There are two other mechanisms that help to resolve the position of a sound source. They are:

- Head movement.
- Angular dependent filtering.

Head movement can be utilised by the ear/brain system to help strengthen auditory cues. For example if a source is at 45^0 to the left (where 0^0 represents straight ahead), then turning the head towards the left would decrease the I.L.D. and I.T.D. between the ears and turning the head to the right would increase the I.L.D. and I.T.D. between the ears. If the source were located behind the listener the opposite would be true, giving the ear/brain system an indication of whether the source is in the front or the back hemi-sphere. In a similar fashion, up/down differentiation can also be resolved with a tilting movement of the head. This is a very important cue in the resolution of front/back reversals perfectly demonstrated by an experiment carried out by Spikofski *et al.* (2001). In this experiment a subject listens to sounds recorded using a fixed dummy head with small microphones placed in its ears. Although reported lateralisation was generally good, many front back reversals are present for some listeners. The same experiment is then conducted with a head tracker placed on the listeners head which controls the angle that the dummy head is facing (that is, the recording dummy head mirrors the movements of the listener in real-time). In this situation virtually no front/back reversals are perceived by the listener. Optimising binaural presentations by utilising the head turning parameter is well documented, however, its consideration in the optimisation of speaker based systems has not been attempted, but will be investigated in this project.

Angular dependant filtering is another cue used by the ear/brain system, and is the only angular direction cue that can be utilised monaurally, that is, sound localisation can be achieved by using just one ear (Gulick, 1989). The filtering results from the body and features of the listener, the most prominent of which is the effect of the pinnae, the cartilage and skin surrounding the opening to the ear canal, as shown in Figure 2.7.

Figure 2.7 The Pinna

The pinna acts as a very complex filtering device, imprinting a unique phase and frequency response onto pressure waves impinging on the head, depending on the angular direction of this pressure wave. This implies that sound sources made up of certain bands are more likely to be heard as emanating from a particular location due to the natural peaks and troughs that are apparent in the HRTF data due to pinna filtering, and this has been shown in experiments using narrow-band sound sources. For example, Zwicker & Fastl (1999) found that narrow band sources of certain frequencies are located at certain positions on the median plane, irrespective of the position of the sound source as indicated in Table 2.1.

Narrow band source centre frequency	Perceived position (in the median plane)
300Hz, 3kHz	Front
8kHz	Above
1kHz, 10kHz	Behind

Table 2.1 Table indicating a narrow band source's perceived position in the median plane, irrespective of actual source position.

The example filters shown in Figure 2.8 (taken from HRTF data measured at the MIT media lab by Gardner & Martin (1994)) shows the phase/magnitude response at the right ear due to a source at 0^0,45^0 and 90^0 to the right of the listener. Interestingly, if the complex filtering from a moving source is heard from a stationary sound source using both ears (e.g. if an in-ear recording is replayed over speakers), the listener will perceive timbral changes in the heard material.

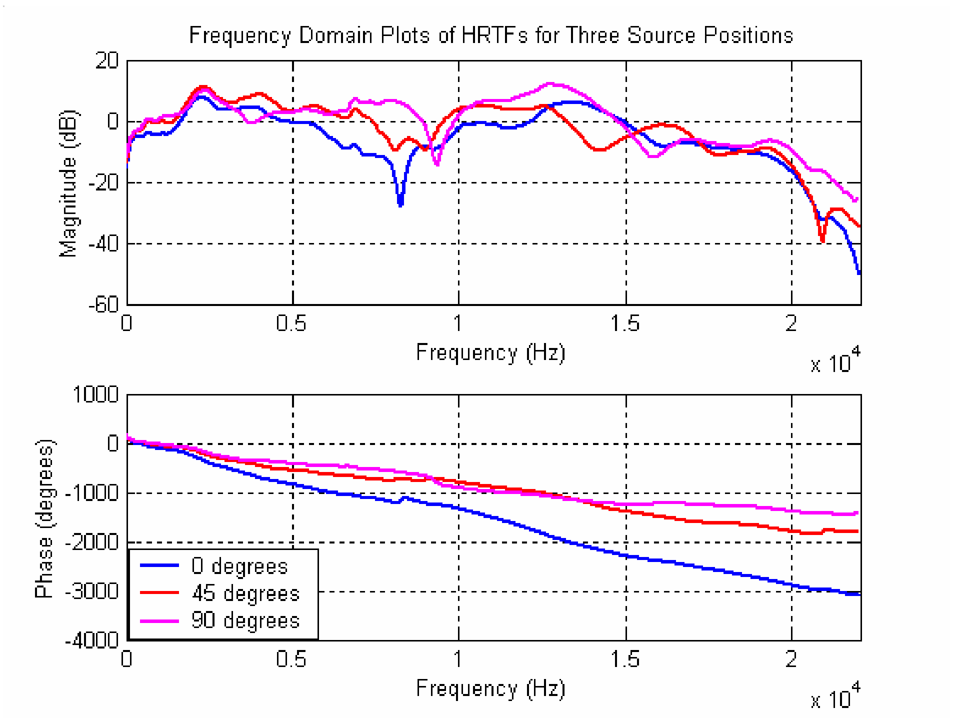


Figure 2.8 **Frequency and phase response at the right ear when subjected to an impulse at 0^0,45^0 and 90^0 to the right of the listener.**

Using the points discussed above, a number of simple assumptions can be made about the human auditory system.

- Amplitude differences between the ears will only be present, and therefore can only be utilised, in sounds greater than some frequency (that is, when the sound no longer diffracts around the head).
- Phase cues can only be totally unambiguous if the sound is delayed by less than half the corresponding wavelength of the sound's frequency (i.e. low frequencies), but may still be utilised together with other cues (such as I.L.D.) up to a delay corresponding to a full wavelength (a phase change of 360^0) (Gulick, W.L. *et al*., 1989).
- Time cues can only be useful when transients are apparent in the sound source, e.g. at the beginning of a sound.

2.2.2 Analysis of the Lateralisation Parameters

In order to quantify what frequency ranges the lateralisation parameters are valid for, an example 'head' is now used. This head was measured at the M.I.T. media lab in the U.S.A. and the impulse response measurements for a great many source positions were taken in an anechoic room. The resulting impulse responses are measures of the Head Related Transfer Function (which result in Head Related Impulse Responses, but are still generally known as HRTFs) due to the dummy head. As the tests were carried out in an anechoic chamber, they are a very good measure of how we lateralise sound sources, that is, the minimum of auditory cues are present as no information regarding the space in which the recordings are made is apparent. Figure 2.9 shows a plot representing the amplitude difference (z-axis) measured between the two ears for frequencies between 0 Hz and 20 kHz (x-axis) and source angles between 0 and 180^0 (y-axis). The red colouring indicates that there is no amplitude difference between the ears, and is most apparent at low frequencies, which is expected as the head does not obstruct the sound wave for these, longer, wavelengths. The amplitude differences in the signals arriving at the ears can be seen to occur at around 700 Hz and then can be seen to increase after this point. This graph shows a significant difference between modelling the head as a sphere (as in Figure 2.2) and measuring the non-spherical dummy head with amplitude peaks and troughs becoming very evident.

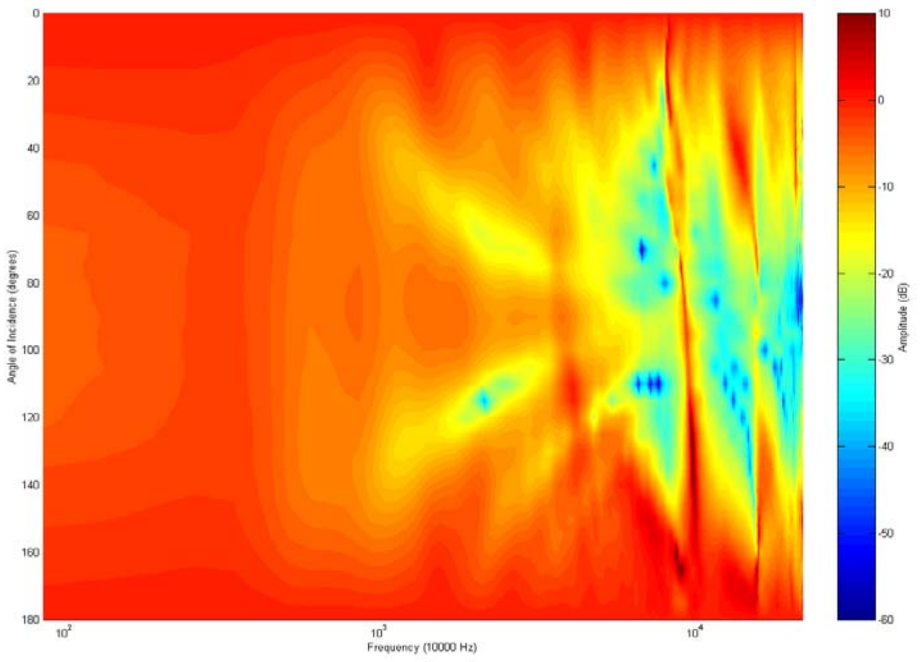


Figure 2.9 **The relationship between source incidence angle, frequency and amplitude difference between the two ears.**

Figure 2.10 shows a very similar graph, but this time, representing the phase difference between the two ears. The colour scaling now goes from –180^0 to 180^0 (although the scale on this graph is in radians, from -3.142 to 3.142). A clear pattern can be observed with the limit of unambiguous phase differences between the ears following a crescent pattern with no phase differences occurring when sounds are directly in front of or behind the listener. The largest phase difference between the ears is to be found from a source at an angle of 90^0 to the listener where unambiguous phase differences occur up to approximately 800 Hz. The anomalies apparent in this figure (negative phase difference) could be due to one of two effects:

- Pinna, head and torso filtering.
- Errors in the measured HRTF data.

Of the two possible effects, the second is most likely, as the compact set of HRTFs were used (see Gardner & Martin (1994)). The compact set of HRTFs has been processed in such a way as to cut down their size and inverse filtered in a crude manner. Given these limitations, a good trend in terms of the phase difference between the two ears is still evident.

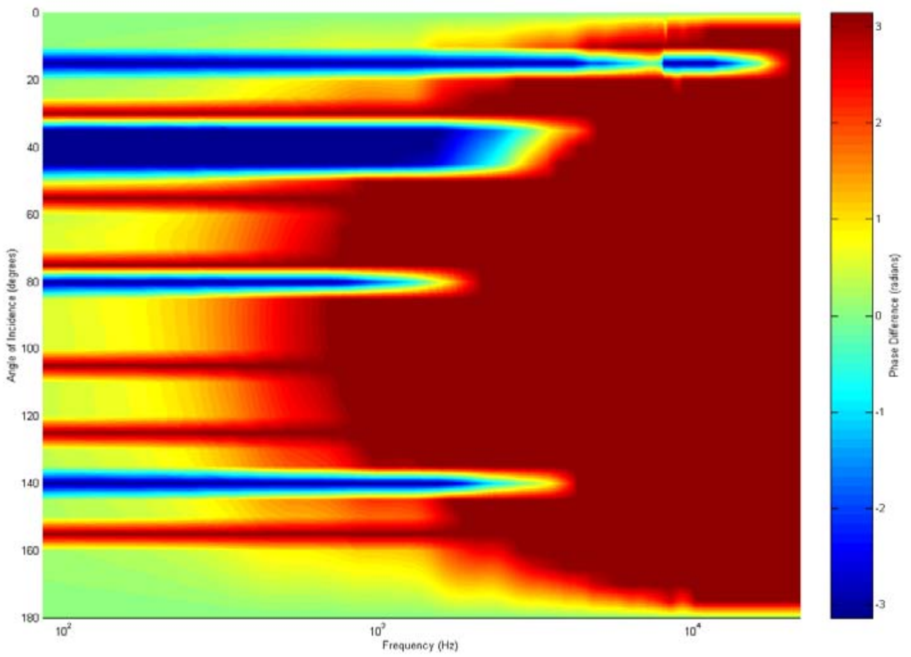


Figure 2.10 **Relationship between source incidence angle, frequency and the phase difference between the two ears.**

Figure 2.11 shows the time of arrival difference between the two ears, and also indicates why interaural time difference and interaural phase difference should be considered as two separate auditory cues. Usable time differences are apparent for every frequency of sound as long as the source is at an off-centre position, and this is the only lateralisation cue for which this is the case. This graph also shows that filtering due to the pinna, head and torso create differing time delays which are dependent upon the frequency of the incoming sound. If some form of time delay filtering were not present (i.e. no head/torso or pinna filtering), the time difference for each source angle of incidence would be constant across the audio spectrum.

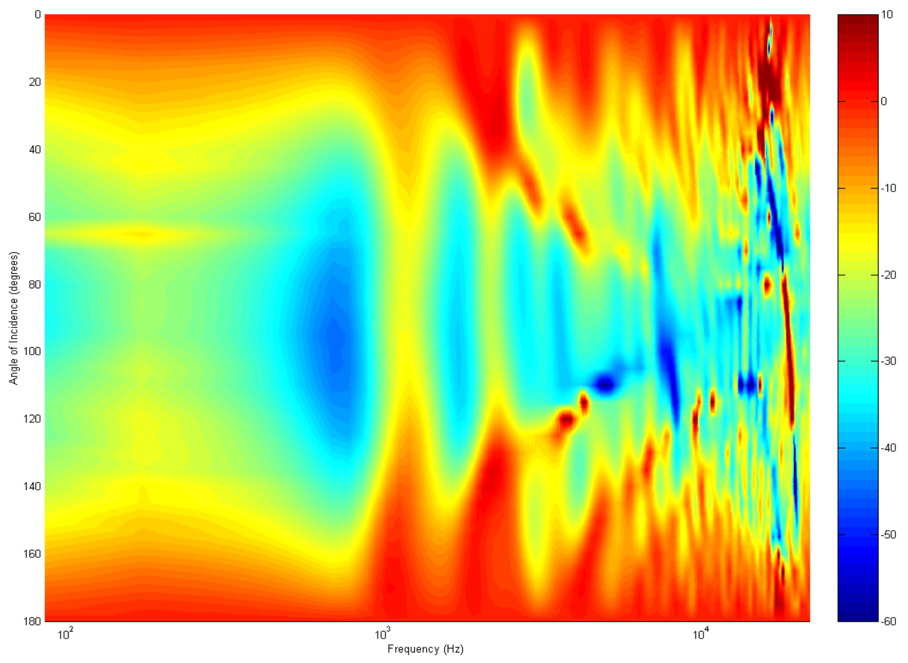


Figure 2.11 Relationship between source incidence angle, frequency and the time difference (in samples) between the two ears.

The three graphs shown in Figure 2.9, Figure 2.10 and Figure 2.11 usefully provide an insight into possible reasons for a number of psychoacoustic phenomena. If we consider the minimum audible angle (M.A.A.) for sounds of differing frequencies, and source azimuths (where the M.A.A. is taken as the angle a source has to be displaced by, until a perceived change in location is noted), it can be seen that the source's M.A.A. gets larger the more off-centre the source's original position (see Figure 2.12 and Gulick (1989)). This is coupled with the M.A.A. increasing for all source positions between the frequencies of 1 kHz and 3 kHz.

The question arises; can the M.A.A. effect be explained using the three H.R.T.F. analysis figures given above? Firstly, why would the minimum audible angle be greater the more off-centre the sound source for low frequencies? If the phase difference graph is observed, then it can be seen that the gradient of the change of phase difference with respect to head movement is greatest when a source is directly behind or directly in front of the listener. That is, if the head is rotated 1^0, then a source directly in front of the listener will create a greater phase change between the two listening conditions when compared to a source that is at an azimuth of 90^0 implying an increased resolution to the front (and rear) of the listener.

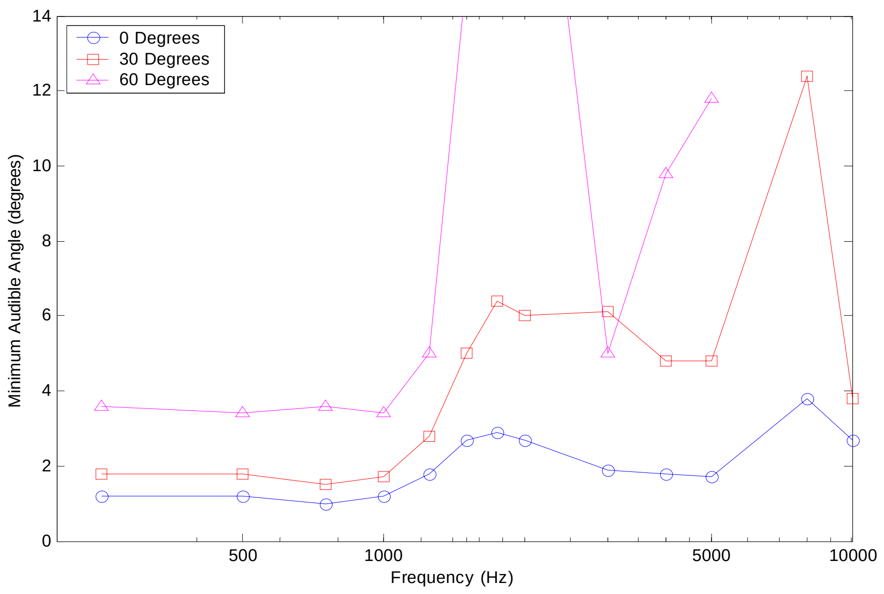


Figure 2.12 **Minimum audible angle between successive tones as a function of frequency and position of source (data taken from Gulick (1989)).**

It should also be noted that the M.A.A. worsens between 1 kHz and 3 kHz. If the interaural amplitude is studied, it can be seen that the difference between the ears starts to become pronounced after approximately 1 kHz and does not become more obvious until higher frequencies. Also, 1 kHz is around the frequency where unambiguous phase cues start to disappear (and more so as the angle of incidence of the source increases). It is this cross-over period between the brain using level and phase cues where the M.A.A. is at its largest. Another interesting result, that can also be seen from Figure 2.12, is that phase cues (used primarily at low frequencies) perform better, on average, than higher frequency cues (pinna filtering and level differences) and it is often mentioned that low frequency, temporal, cues are the more robust cues (for example, Wightman, F.L. and Kistler, D.J., 1992 and Huopaniemi, J. et al, 1999).

2.3 Sound Localisation

The term localisation differs from lateralisation in that not only is source direction angle arrived at, but a listener can gain information on the type of location a sound is emanating from and the distance from the source to the listener. Also, information on the size of a sound source as well as which way it may be facing can be gleaned just by listening for a short time.

2.3.1 Room Localisation

When walking into an acoustic space for the first time, the brain quickly makes a number of assumptions about the listening environment. It does this using the sound of the room (using any sounds present) and the reaction of the listener inside this room. One example of this is when walking into a cathedral. In this situation one of the first sounds possibly heard will be your own footsteps, and this will soon give the impression that the listener is in a large, enclosed space. This is also the reason that people susceptible to claustrophobia are ill advised to enter an anechoic chamber, as the lack of any reverberation in the room can be very disconcerting, and bring on a claustrophobic reaction. Interestingly, listening to sound sources in an anechoic chamber will often give the impression that the sound source is almost ‘inside the head’ (much like listening to conventional sound sources through headphones). The human brain is not used to listening to sounds without a corresponding location (even large open expanses have sound reflections from the floor), and the only time this will happen is if the source is very close to the head, somebody whispering in your ear, for example, and so the brain decides that any sound without a location is likely to be very close.

If we are listening to a sound source in a real location, a large number of reflections may also reach the ears. The first sound that is heard will be the direct sound, as this has the shortest path length (assuming nothing obstructs the source). Then, the first order reflections will be heard. Figure 2.13 shows a simplified example of this (in two dimensions). Here it can clearly be seen that the direct sound has the shortest path length, which implies that this signal has the properties listed below:

- The direct sound will be the loudest signal from the source to reach the listener (both due to the extra path length and the fact that some of the reflected source's energy will be absorbed by the reflective surface).
- The direct sound will be the first signal to reach the ears of the listener.
- The direct sound may be the only signal that will be encoded (by the head of the listener) in the correct direction.

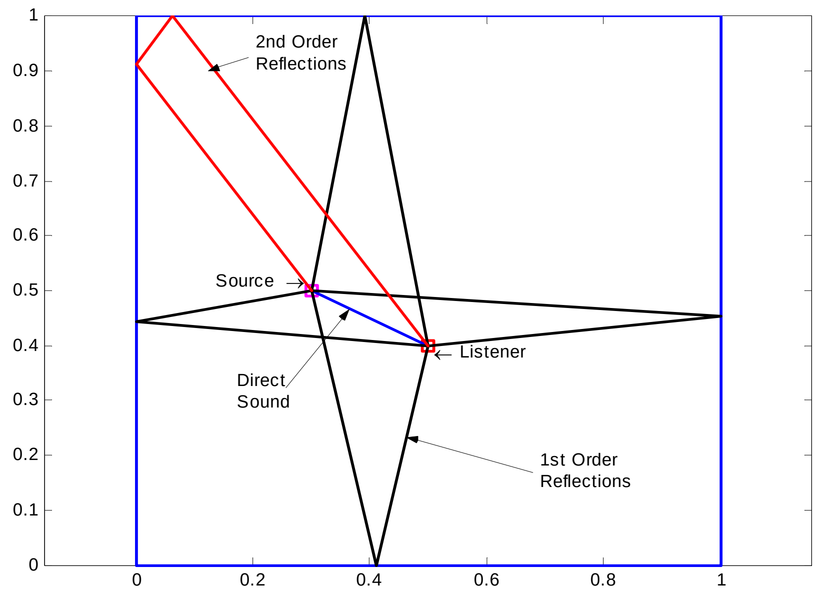


Figure 2.13 Simple example of a source listened to in a room. Direct, four 1st order reflections and one 2nd order reflection shown (horizontal only).

In the example shown above (Figure 2.13) a simple square room is shown along with four of the 1st order sound reflections (there are two missing, one from the floor and one from the ceiling) and one 2nd order reflection. These signal paths will also be attenuated due to absorption associated with the wall and the air. Surfaces in a room, and the air itself, possess an absorption coefficient, a numerical grade of acoustic absorption, although a more accurate measure is the frequency dependent absorption coefficient. As reflections in the room build up to higher and higher orders, a diffuse sound field is created, where the individual echoes are more difficult to analyse. Figure 2.14 shows an example impulse response of an actual room. The room has a reasonably short reverberation time as the walls are acoustically treated with foam panels. The graph shows ¼ of a second in time (11025 samples at 44.1 kHz sampling rate).

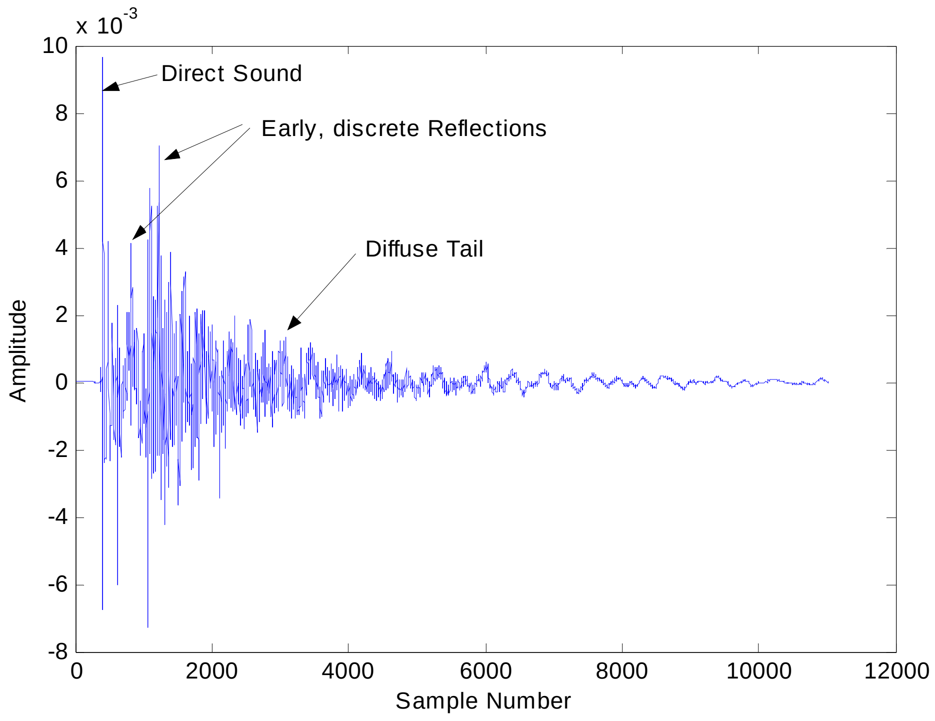


Figure 2.14 Impulse response of an acoustically treated listening room.

As mentioned at the beginning of this section, the response of a room gives listeners significant insight into the type of environment that they are in. However, Figure 2.14 shows a very complicated response. So how does the brain process this? An extremely important psychoacoustic phenomenon and one that the ear/brain system uses in this type of situation has been termed the precedence effect (Begault, 2000). The precedence effect is where the brain gives precedence to the sound arriving at the listener first, with the direction of this first sound taken as the angular direction indicator. This sounds very simple, but as we have two ears, the initial sound arrives at the ears twice and, therefore, has two arrival times associated with it. Figure 2.15 shows the equivalent reverberation impulse responses that arrive at both ears. The source used in this graph is at 30^0 to the left of the listener very close to the rear wall, and about 1 metre away from the left wall. It can clearly be seen that the source's direct sound arrives at the left ear first, followed, around 11 samples later (0.25 ms at 44.1 kHz), by the right ear. As the ear/brain system uses this time difference to help lateralise the incoming sound, the precedence effect does not function for such short time differences. Under laboratory tests it has been noted that if the same signal is played into each ear of a pair of headphones, but one channel is delayed slightly (Begault, 2000):

- For a delay between 0 and 0.6mS the source will move from the centre towards the undelayed side of the listeners head.
- Between approximately 0.7 and 35mS the source will remain at the undelayed side of the listeners head, that is, the precedence effect employs the first source to determine the lateralisation. However, although the source position will not change, the perceived tone, and width of the source will tend to alter as the delay between the left and right ears is increased (note that this implies an effect analogous to comb filtering which occurs during the processing of the sounds arriving at the two ears by the brain of the listener).
- Finally, increasing the time delay still further will create the illusion of two separate sources one to the left of the listener and one to the right. The delayed source is perceived as an echo.

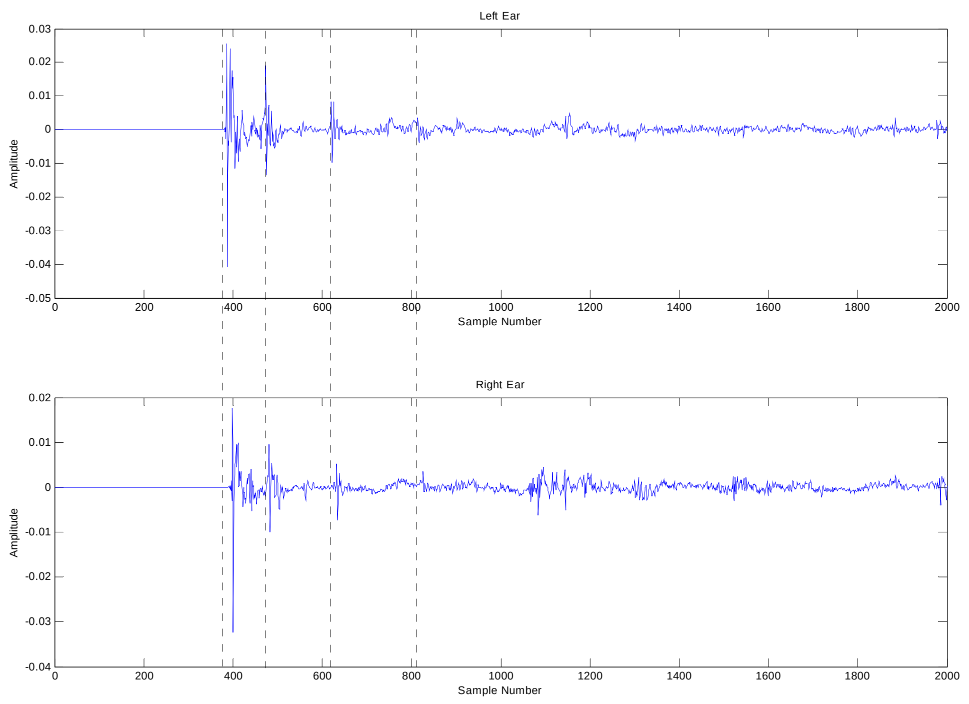


Figure 2.15 **Binaural impulse response from a source at 30^0 to the left of the listener. Dotted lines indicate some discrete reflections arriving at left ear.**

The above points help to explain why the ear/brain system uses the precedence effect. If a source has many early reflections (i.e. the source is in a reverberant room) the ear/brain system needs a way of discriminating between the direct sound and the room's response to that sound (reflections and diffuse field). The precedence effect is the result of this phenomenon. If we take a source in a room (as given in figure 2.13), assuming the room is a 4 m by 4 m, square room and the initial source is 0.45m away from the listener (that is, the listener and source positions are as in Figure 2.13). The direct sound will take 1.3ms to reach the listener (taking the speed of sound in air as 342 ms^{-1}). The source is at an azimuth of approximately 63^0 from straight ahead which will lead to a time difference between the ears of around 0.5 ms (using the approximate binaural distance equation from Gulick (1989)). The nearest reflection has a path length of around 3.2 m from the source to the listener which equates to a delay time of 9.4 ms. Because of the precedence effect, the first time delay between the ears will be utilised in the lateralisation of the sound source, and the first discrete echo will not be heard as an echo, but it will not change the perceived position of the sound source either, and will just change the width or timbre of the source. It is this type of processing in the ear/brain system that gives us vital information about the type of space we are situated in. However, as the above points suggest, it may be at the expense of localisation accuracy, with the precedence effect breaking down if the echo is louder than the direct sound, which normally only occurs if the source is out of sight, but a reflection path off a wall is the loudest sound to reach the listener.

2.3.2 Height and Distance Perception

Although lateralisation has been discussed, no explanation has yet been given to resolution of sources that appear above or below the listener. As the ears of the listener are both on the same plane, horizontally, the sound reaching each ear will not contain any path differences due to elevation (although, obviously, if a sound is elevated and off-centre, the path differences for the lateral position of the sound will be present), and as there are no path differences the only static cue that can be utilised for an elevated cue is the comb filtering introduced by head and pinna. Figure 2.16 shows a 3-axis graph representing a source straight in front of the listener changing elevation angle from -40^0 to 90^0. Perhaps the most notable feature of this plot is the pronounced trough that originates at around 7 kHz for an elevation of -40^0, which goes through a smooth transition to around 11 kHz at an elevation of 60^0. It is most probably these pinna filtering cues (combined with head movements) that are used to resolve sources that are above and below the listener (Zwicker & Fastl, 1999). Interestingly, it has also been shown in Zwicker & Fastl (1999) that narrow, band-limited sources heard by a listener can have a ‘natural’ direction. For example, an 8 kHz centre frequency is perceived as coming from a location above the head of the subject, whereas a 1 kHz centre frequency is perceived as coming from a location behind the listener.

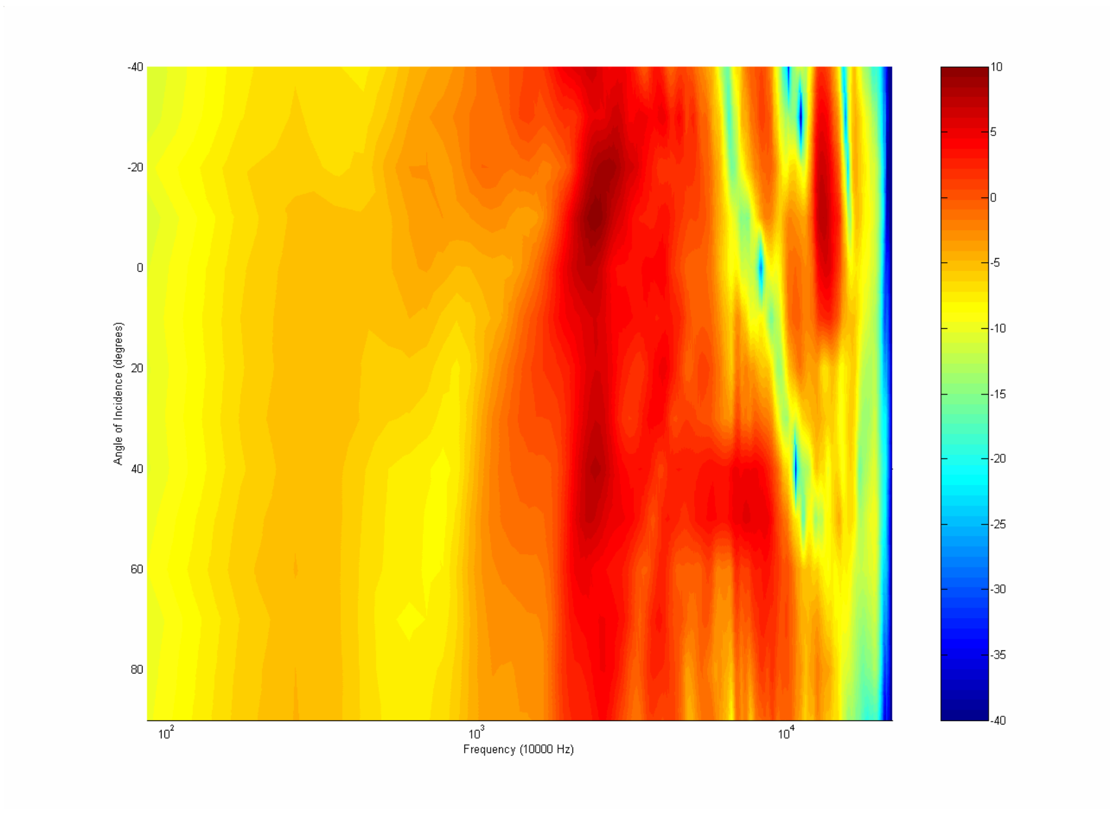


Figure 2.16 Relationship between source elevation angle, frequency and the amplitude at an ear of a listener (source is at an azimuth of 0^0).

In order to assess the apparent distance of a source to the listener, a number of auditory cues are used. The first and most obvious cue is that of amplitude. That is, a source that is near by will be louder than a source that is further away. The relationship between a point source's amplitude and distance, in the free field, is known as the inverse square law as for each doubling of distance, the amplitude of the source will reduce by a quarter ($1/[2^2]$). This is, of course, the simplest case, only holding true for a point source in the free field. Sources are rarely a perfect point source, and rarely heard in the perfect free field (i.e. anechoic circumstances) so, in reality, the amplitude reduction is normally less than the inverse square law suggests. In addition to the pure amplitude changes, distance dependant filtering can be observed, due to air absorption (Savioja, 1999). This will result in a more low-pass filtered signal, the further away the source. The direct to reverberant ratio of the sound will change depending on the source's distance to the listener with a source close to the listener exhibiting a large amount of direct sound when compared to the reverberation, but a sound further away will have a similar amount of reverberation, but a lower level of direct sound

(Begault, 2000). There are two reasons for this. Firstly, the diffuse part of the room's response (i.e. the part not made up of direct sound or first order reflections) is made up from the sound bouncing off many surfaces, and as such, will be present all through the room. This means that the level of this part of the reverberation is reasonably constant throughout the room. Also, as the source moves away from the listener, the distance ratio between the path length of the direct sound and the early reflections becomes closer to one. This means that the first reflections will arrive closer (in time), and have an amplitude that is more similar to the level of the direct sound. This is shown in Figure 2.17.

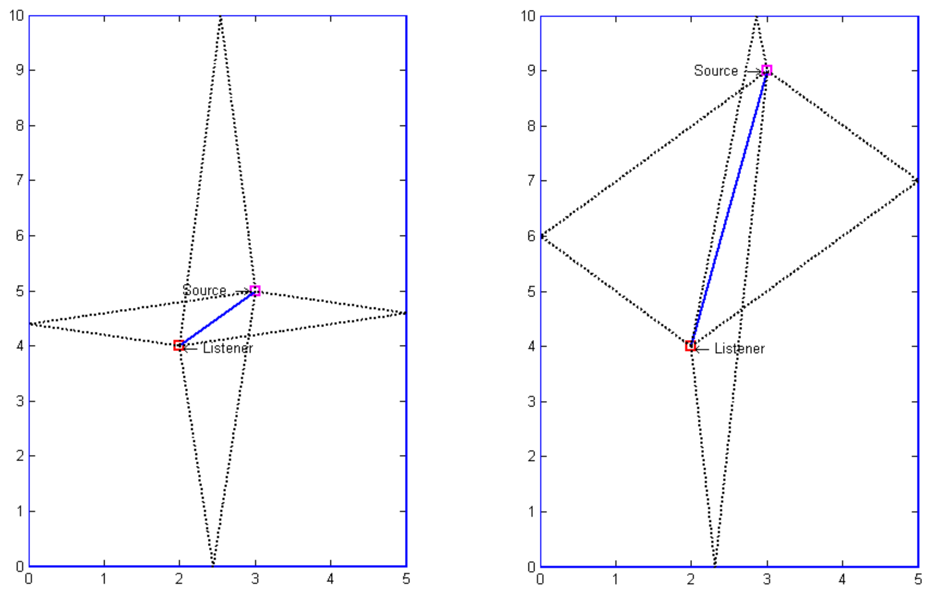


Figure 2.17 A graph showing the direct sound and early reflections of two sources in a room.

Evidence suggests that the reverberation cue is one of the more robust cues in the simulation of distance and has been shown to create the illusion of a sound source outside of the head under headphone listening conditions (McKeag & McGrath, 1997).

Of all the cues available to differentiate source distances, the least apparent is that the source's incidence angle from the listener's ears will change as the source is moved away from a listener (Gulick, 1989). Figure 2.18 shows two source examples, one source very close to the listener, and one source at infinity. The close source has a greater binaural distance associated with it when compared to the far source. This means that as sources move off-centre, the binaural distance for a far source will not increase as quickly as the binaural distance for a near source (that is, the maximum binaural time difference is less for a far source).

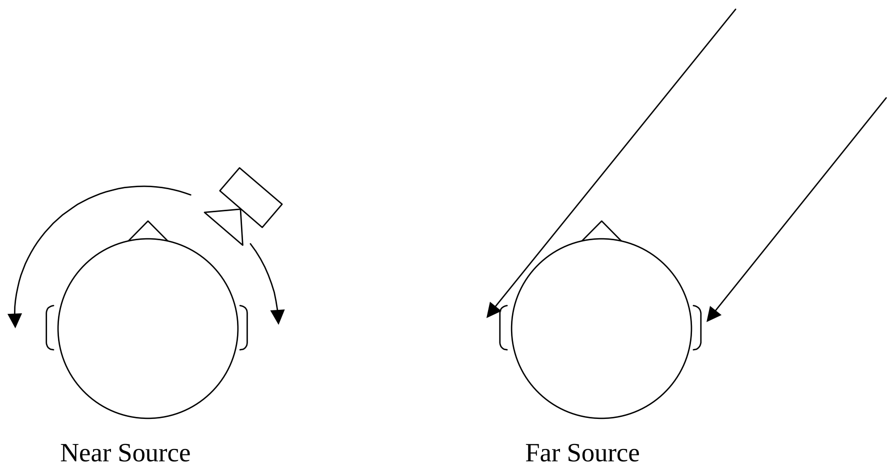


Figure 2.18 A near and far source impinging on the head.

2.4 Summary

In summary, the ear/brain system uses a number of different cues when trying to make sense of the sounds that we hear. These consist of the low level cues that are a result of the position and shape of the ears, such as:

- Interaural level differences.
- Interaural phase and time differences.
- Head/torso and pinna filtering.

These cues are used by the ear/brain system to help determine the angular direction of a sound, but are also combined and processed using higher order cognitive functions in order to help make sense of such things as the environment that the sounds have occurred in. It is these higher order functions that give us the sense of the environment that we are in, assigning more information to the object than a directional characteristic alone. Such attributes as distance perception are formed in this way, but other attributes can also be attached in a similar manner, such as the size of an object, or an estimation as to whether the sounding object is facing us, or not (in the case of a person talking, for example).

If a successful surround sound system is to be developed then it is apparent that not only should the low-level cues be satisfied, but they should also be as coherent with one another as possible so that the higher order cognitive functions of the ear/brain system can also be satisfied in a useful and meaningful way.

Chapter 3 - Surround Sound Systems

3.1 Introduction

In this chapter past and current surround sound algorithms and techniques will be discussed starting with a historical account of the first systems, proposed by Bell Labs and Alan Blumlein, how Blumlein's early system was used as a loose basis for stereo, and then on to the theory and rationale behind the systems that are used presently.

The early systems are of importance as most surround sound systems in use today base themselves on the techniques and principles of this early work. In the context of this research, one main system will be decided upon as warranting further research in order to fulfil the research problem detailed in Chapter 1, with the following criteria needing to be met:

- A hierarchical carrier format must be decided upon.
- This carrier must be able to be decoded for multi-speaker systems with different speaker arrangements.
- This decode must be able to provide the listener with the relevant auditory cues which will translate well into a binaural representation.

As the above system is to be converted into a binaural and transaural representation, these systems will also be discussed.

3.2 Historic Review of Surround Sound Techniques and Theory

Although standard stereo equipment works with two channels, early work was not necessarily fixed to that number, with the stereo arrangement familiar to us today not becoming a standard until the 1950s. Bell labs original work was predicated on many more speakers than this initially (Rumsey & McCormick, 1994) and is the first system described in this section.

3.2.1 Bell Labs' Early Spaced Microphone Technique

The early aims of the first directional sound reproduction techniques tried at Bell Labs was that of trying to reproduce the sound wave front from a source on a stage (Rumsey & McCormick, 1994). A sound source was placed on a stage in a room; this was then picked up by a large number of closely spaced microphones in a row, in front of the source. These signals were then transmitted to an equal number of similarly spaced loudspeakers (as shown in Figure 3.1).

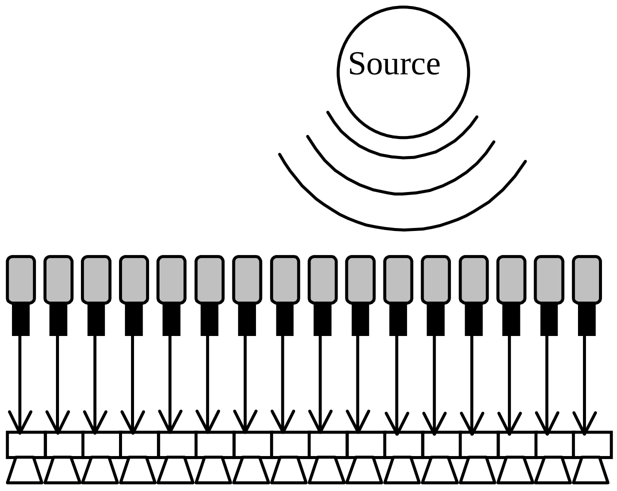


Figure 3.1 **Graphical depiction of early Bell Labs experiments. Infinite number of microphones and speakers model.**

The result was an accurate virtual image that did not depend on the position of the listener (within limits) as the wave front approaching the speakers is reproduced well, much like wave-field synthesis (to be discussed later in this chapter). Bell Labs then tried to see if they could recreate the same idea using a smaller number of speakers (Figure 3.2), but this did not perform as accurately (Steinberg, J. & Snow, W., 1934). The main problem with such a setup is that once the many speakers are removed, the three sources (as in the example shown in Figure 3.2) do not reconstruct the wave front correctly.

Let us consider the three speaker example shown in Figure 3.2. If the source is recorded by three microphones, as shown, the middle microphone will receive the signal first, followed then by the microphone on the right, and lastly captured by the microphone on the left. These three signals are reproduced by the three loudspeakers. If the listener is placed directly in front of the middle loudspeaker, then the signal from the middle speaker will reach them first, followed by the right and left loudspeakers together. However, as the signal from the source was delayed in reaching the left and right microphones, the delay from each of the left and right speakers is increased even more. Now, if the combined spacing between the microphones and speakers equates to a spacing greater than the diameter of the head, then the time delays reproduced at the ears of the listener will be greater than the maximum interaural time difference of a real source. This will then result in either the precedence effect taking over (i.e. the source will emanate from the centre loudspeaker) or, worse still, echoes will be perceived. This is due to a phenomenon known as ‘spatial aliasing’ and will be described in more detail in section 3.3.2. The spacing of the microphones was necessary as directional microphones had not been invented at this point in time, and only pressure sensitive, omnidirectional microphones were available.

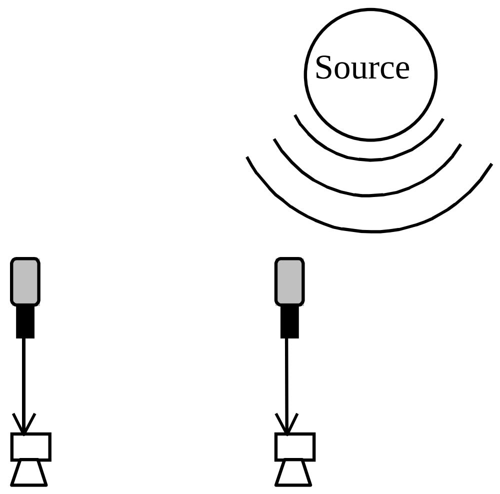


Figure 3.2 Early Bell Labs experiment. Limited number of microphones and speakers model.

3.2.2 Blumlein’s Binaural Reproduction System

While carrying out research into the work of Alan Blumlein, it was soon discovered that there seems to be some confusion, in the audio industry, about certain aspects of his inventions. This seems mainly due to the fact that the names of the various techniques he pioneered have been changed, or misquoted, from the names that he originally gave. Alan Blumlein delivered a patent specification in 1931 (Blumlein, 1931) that both recognised the problems with the Bell Labs approach and defined a method for converting spaced microphone feeds to a signal suitable for loudspeaker reproduction. Blumlein called his invention Binaural Reproduction. This recording technique comprised of two omni-directional microphones spaced at a distance similar to that found between the ears, with a round panel baffle in between them. This technique was known to work well for headphone listening, but did not perform as accurately when replayed on loudspeakers. Blumlein realised that for loudspeaker reproduction, phase differences at the speakers (i.e. in the spaced microphone recording) did not reproduce phase differences at the listener's ears. This was due to the unavoidable crosstalk between the two speakers and the two ears of the listener, as shown in Figure 3.3.

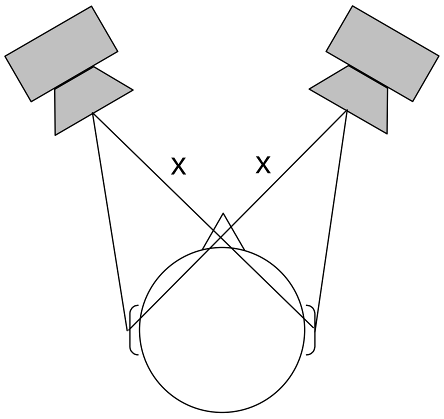


Figure 3.3 **Standard "stereo triangle" with the speakers at +/-30^0 to the listener (x denotes the crosstalk path).**

Blumlein had discovered that in order to reproduce phase differences at the ears of a listener, level differences needed to be presented by the speakers. His invention included the description of a 'Shuffling' circuit, which is a device that converts the phase differences, present in spaced microphone recordings, to amplitude differences at low frequencies (as at higher frequencies the amplitude differences would already be present due to the sound shadow presented by the disk between the two microphones).

If we consider the stereo pair of loudspeakers shown in Figure 3.3, it can be seen that there are two paths from each speaker to each ear of the listener. If the sound that is recorded from the Blumlein stereo pair of microphones is to the left of centre, then the left channel's signal will be greater in amplitude than the right channel's signal. Four signals will then be transmitted to the ears:

1. The left speaker to the left ear.

2. The left speaker to the right ear.
3. The right speaker to the right ear.
4. The right speaker to the left ear.

If we take the case of a low frequency sound (where the interaural phase difference is the major cue), as the paths from the speaker to the contralateral ear is longer than from the speaker to the ipsilateral ear, the signal will appear delayed in time (but not changed in amplitude, due to the wave diffracting around the head, see Chapter 2). The resulting signals that arrive at each ear are shown in Figure 3.4.

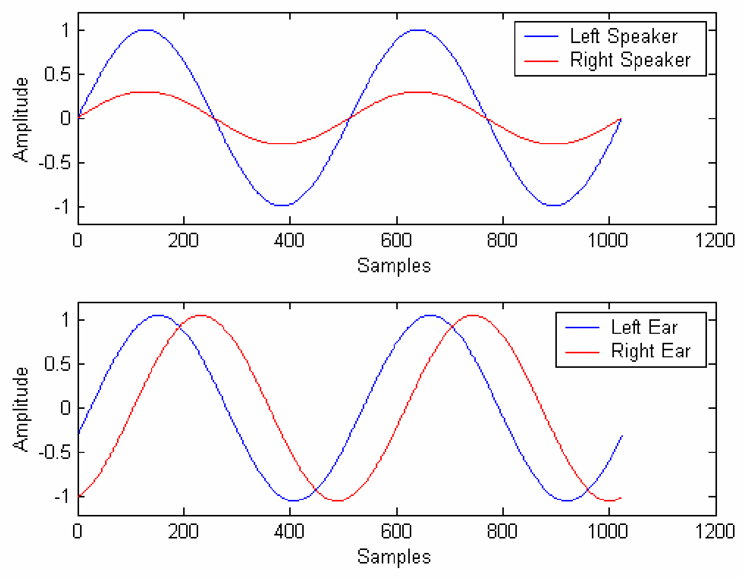


Figure 3.4 Low frequency simulation of a source recorded in Blumlein Stereo and replayed over a pair of loudspeakers. The source is to the left of centre.

It can be clearly seen that low frequency phase cues can be encoded into a stereo signal using just amplitude differences and once the head starts to become a physical obstacle for the reproduced signals (at higher frequencies), a level difference between the ears will also become apparent.

It may seem strange that Blumlein used a spaced microphone array to model what seems to be a coincident, amplitude weighted, microphone technique, but only omnidirectional microphones were available at this time. However, less than a year later a directional, ribbon microphone appeared that had a figure of eight polar response. This microphone was better suited to Blumlein’s Binaural Reproduction technique.

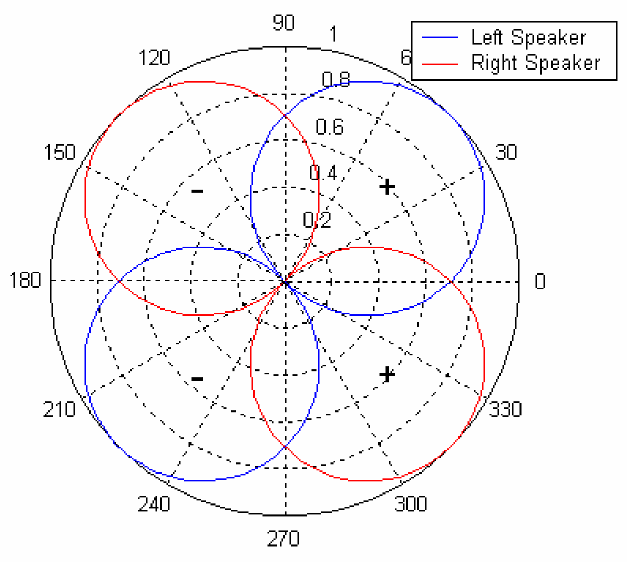


Figure 3.5 Polar pickup patterns for Blumlein Stereo technique

Blumlein’s coincident microphone technique involved the use of two coincident microphones with figure of eight pickup patterns (Blumlein, 1931) (as shown in Figure 3.5) and has a number of advantages over the spaced microphone set-up shown in Figure 3.2. Firstly, this system is mono compatible, whereas spaced microphone techniques are generally not (if not shuffled). If we again consider the microphone arrangement given in Figure 3.2 then each of the microphones receives the same signal, but changed in delay and amplitude. As there are delays involved, adding up the different channels will produce comb-filtering effects (as different frequencies will cancel out and reinforce each other depending on their wavelengths). However, this will not be the case using Blumlein’s binaural sound as the two microphones will pick up the same signal, differing only in amplitude. A mono signal can be constructed by adding the left and right signals together resulting in a forward facing figure of eight response. The Blumlein approach also has the added advantage that the actual signals that are presented from each loudspeaker can be altered *after* the recording process. For example, the apparent width of the sound stage can be altered using various mixtures of the sum and difference signals (see spatial equalisation, later in this section). Also, Blumlein based his work on what the ear would hear, and described how a stereo image, made up of amplitude differences alone, could create low frequency phase cues at the ears of a listener (Blumlein, 1931).

Blumlein did foresee one problem with his two microphone arrangement, however. This was that the amplitude and phase cues for mid and low frequencies, respectively, would not be in agreement (Blumlein, 1931; Glasgal, 2003a). It was possible to solve this problem using the fact that the signals fed to each speaker could be altered after recording using the sum and difference signals. This technique is now known as spatial equalisation (Gerzon, 1994), and consisted of changing the low frequency signals that fed the left and right speaker by boosting the difference signal and cutting the sum signal by the same amount (usually around 4dB). This has the effect of altering the pickup pattern for the recorded material in a manner shown in Figure 3.6. This technique is still used today, and is a basis for parts of the Lexicon Logic 7™ (Surround Sound Mailing List Archive, 2001) and Ambisonic systems (Gerzon, 1974), the principles of which will be discussed in detail later in this chapter.

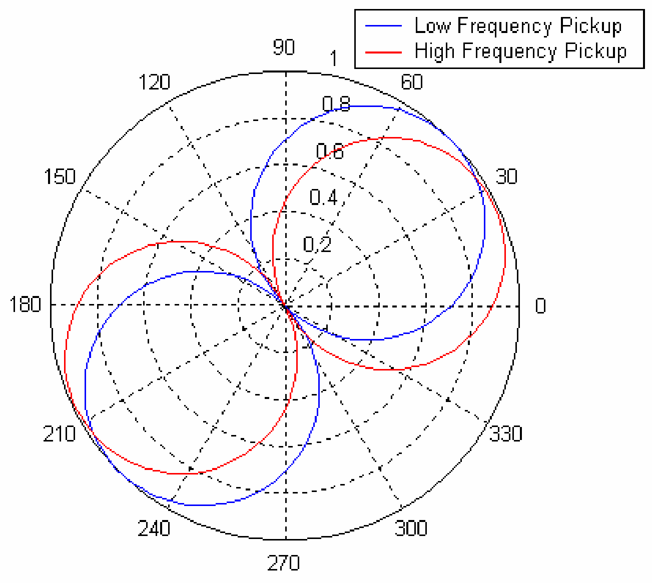


Figure 3.6 **Graph showing the pick up patterns of the left speaker's feed after spatial equalisation.**

Blumlein's binaural reproduction technique is one of the few that truly separates the encoding of the signal from the decoding, which allows for the various post recording steps that can be carried out in a clearly defined, mathematically elegant way. Blumlein was soon employed by the military to work on radar, amongst other things. It may be because of this that Blumlein's work was not openly recognised for a number of years (Alexander, 1997), but his principles were later used in the formulation of a three dimensional sound system (see Ambisonics, later in this chapter).

3.2.3 Stereo Spaced Microphone Techniques

Although the Blumlein Stereo technique has many advantages as a recording format when used for reproduction over loudspeakers, there is another school of thought on this matter. This is that such 'summation localisation theories' cannot hope to accurately reproduce recorded material as no onset time delay is introduced into the equation, and if this is the case, then although steady state (continuous) signals can be reproduced faithfully, the onset of sounds cannot be reproduced with strong enough cues present to successfully fool the ear/brain system. To this end, a number of spaced microphone techniques were developed that circumvented some of the problems associated with Bell Labs wave front reconstruction technique described above. It must be noted, however, that Blumlein did use spaced microphone techniques to record sound as he was well aware that, for headphone listening, this produced the best results. However, in order to replay these recordings over speakers, to achieve externalisation, a Blumlein shuffler was used, that converted the signals, at low frequencies, to consist of only amplitude differences.

If we recall from the Bell Labs system, anomalies occurred because of the potentially large spacing between the microphones that were picking up the sound sources. A more logical approach is a near-coincident microphone technique that will limit the time of arrival errors so that the maximum time difference experienced by a listener will not be perceived as an echo. The ORTF method uses a pair of spaced directional microphones usually spaced by around 17 cm (roughly equal to the diameter of a human head) and at an angle of separation of 110^0 (as shown in Figure 3.7). This means that the largest possible time difference between the two channels is comparable with the largest time of arrival difference experienced by a real listener. Directional microphones are used to simulate the shadowing effect of the head. This arrangement is a trade off between spaced and coincident microphone techniques as it has the increased spaciousness of spaced microphones (due to the increased de-correlation of the two signals) but also has reasonably good mono compatibility due to the close proximity of the microphone capsules.

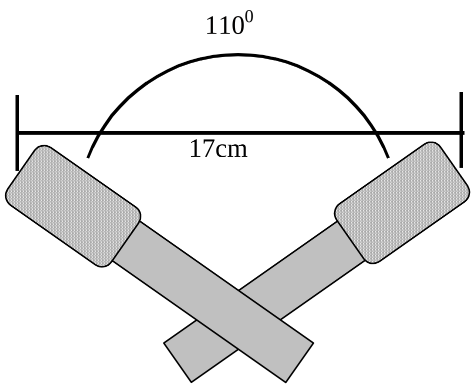


Figure 3.7 ORTF near-coincident microphone technique.

Another widely used technique is the Decca Tree (Rumsey and McCormick, 1994). This is a group of three microphones matrixed together to create two loudspeaker feeds. An example of the Decca Tree arrangement is shown in Figure 3.8. In this arrangement the centre microphone feed is sent to both channels, the left microphone feed is sent to the left channel and the right microphone is sent to the right channel. In this way, the differences between the two channels outputs are lessened, giving a more stable central image, and alleviating the ‘hole in the middle’ type effect of a spaced omni technique (the sound always seeming to originate from a specific speaker, as in the Bell Labs set-up).

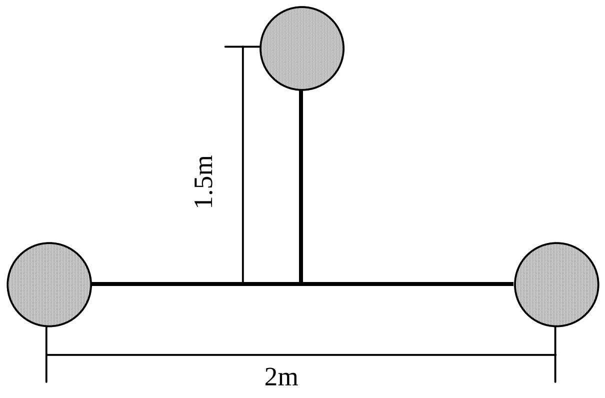


Figure 3.8 **Typical Decca Tree microphone arrangement (using omni-directional capsules).**

3.2.4 Pan-potted Stereo

The systems that have been discussed thus far have been able to record events for multiple speaker playback, but a system was needed that could be used to artificially place sources in the desired location to create the illusion of a recorded situation. Due to the simplicity of Blumlein stereo, as opposed to spaced microphone techniques, creating a system where individual sources could be artificially positioned was based on amplitude panning (Rumsey and McCormick, 1994). So, a simulation of the Blumlein coincident microphone system was needed. As the coincident microphones were figure of eight responses the gains needed to artificially pan a sound from the left speaker to the right speaker are given in equation (3.1). The SPos offset parameter is basically to 'steer' the virtual figure-of-eight responses so that a signal at one speaker position will have no gain at the opposite speaker, i.e. a virtual source at the speaker position is an *actual* source at the speaker position.

$$\begin{aligned} LeftGain &= \sin(\theta + SPos) \\ RightGain &= \cos(\theta + SPos) \end{aligned} \tag{3.1}$$

where: SPos is the absolute angular position of the speaker.

θ is the desired source position (from SPos^0 to –SPos^0).

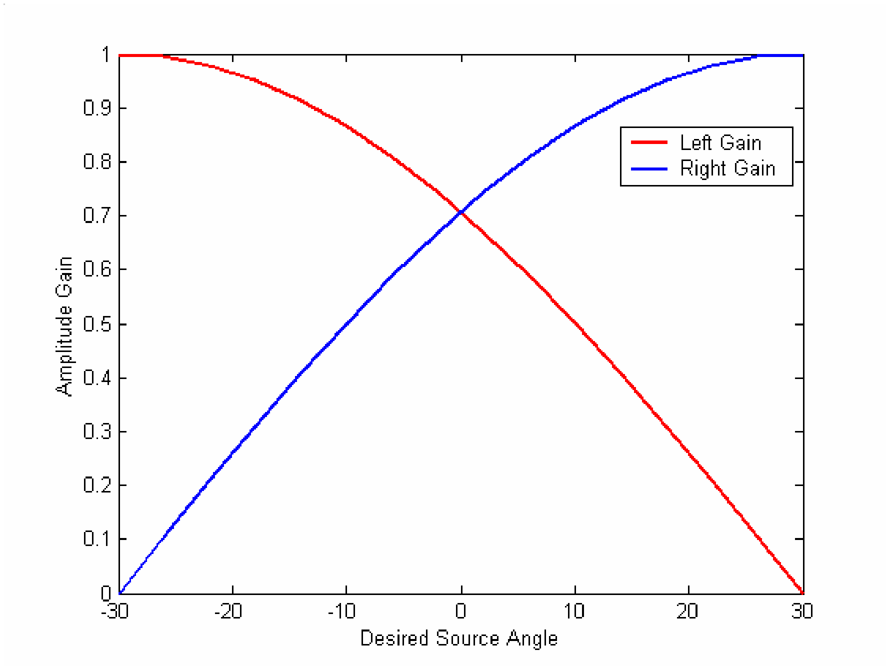


Figure 3.9 **A stereo panning law based on Blumlein stereo.**

This is, however, really a simplification of Blumlein's stereo technique as his spatial equalisation circuit is generally not used in amplitude stereo panning techniques.

Simple amplitude (or pair-wise panning) has now been used for many years, but does suffer from a few problems. It has been shown that the maximum speaker separation that can be successfully utilised is +/- 30^0 and that side-imaging is very hard to achieve using this method (Glasgal, 2003b). Both of these facts are not necessarily detrimental to simple two-speaker stereo reproduction, but will present a larger problem with surround sound techniques as this would mean a minimum of six equally spaced speakers placed around the speaker would need to be used (based on only the angular spacing assumption).

In summary, there are basically two schools of thought when it comes to the recording of live situations for replay over a stereo speaker array (pan-potted, stereo, material is almost always amplitude panned, although artificial reverberation devices often mimic a spaced microphone array rather than a coincident setup). There are those that abide by spaced microphone techniques, reasoning that the time onset cues are very important to the ear/brain system (i.e. the precedence effect) and these are impossible to recreate using a coincident microphone arrangement. On the other side there are those who prefer the mathematical simplicity of coincident microphone arrangements, believing that the potential phase/time misalignment of the signals originating from the speakers in spaced microphone techniques to be detrimental to both the timbre and accuracy of the recorded material.   Of course, both are correct to a certain degree and both coincident and spaced techniques can produce very pleasing results. However, the main problem with spaced microphone techniques is that, because potentially unknown time differences will be present between the two channels, the practical reprocessing of new signal feeds becomes much more difficult, while not an issue for two-speaker stereo, will become an issue for larger arrays of speakers.

3.2.5 Enhanced Stereo

As can be deduced from both Blumlein and Bell Labs early work, stereo sound (which, incidentally, neither Blumlein or Bell Labs referred to their work as ‘Stereo’ sound) was never limited, theoretically, to just two speakers, as their work was mainly geared towards film sound reproduction that needed to encompass large audiences. Three speakers was a good minimum for such a situation as it was soon found that angular distortion was not too detrimental to the experience, except when it came to dialogue (Blumlein’s original idea of the dialogue following the actors was not widely taken up). Dialogue needed to always sound as if it was coming from the screen and not the nearest speaker to the listener, which could happen due to the precedence effect. To this end the centre speaker was useful for both fixing dialogue to the centre of the sound stage, and also for increasing the useful listening area of the room. If a source is panned between two speakers, then a mixture of the time difference and the level difference between the ears will be used to calculate where the sound source is originating from. So, if the listener is in the centre of the two speakers the time (phase) cues will be constructed from the level differences between the speakers. However, as the listener moves off-centre the time delay from the two speakers will change the perceived direction of the sound source. This time difference can be counteracted by the amplitude differences between the two speakers, but angular distortion will always occur, and once the listener is much closer to one speaker than the other, all but the hardest panned material will tend to emanate from the closer of the two loudspeakers. Hence, having a centre speaker not only fixed dialogue to the screen, but also lessened the maximum time difference that could be experienced between two speakers at any one time.

3.2.6 Dolby Stereo

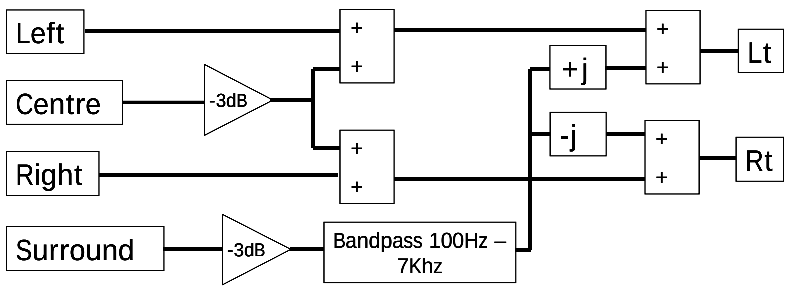
Much of the motivation for early surround sound implementations was the cinema, and early multi-channel playback was attempted as early as 1939 in the Disney film, Fantasia (Kay *et al*. 1998). However, although a magnetic multi-channel standard had been available since the 1950's (Dolby Labs, 2002), it was not as robust or long lasting as the mono optical track that was used at this time. Dolby was to change this in 1975 mainly due to the use of their noise reduction techniques that had revolutionised the professional recording industry since the 1960's. The optical system in use at that time had a number of problems associated with it. The standard for the mono track's frequency response was developed in the 1930's which, although making the soundtrack replayable in almost any cinema in the world, reduced the bandwidth to that of a telephone. This response, called the Academy characteristic (Dolby Labs, 2002), also meant that the soundtracks were recorded with so much high frequency pre-emphasis that considerable distortion was also present in the audio. Dolby's research found that most of these problems were because of the low signal to noise ratio of the optical transmission medium, and in the late 1960's looked at using their type A noise reduction systems in order to improve the response of the sound. Although this worked very well, the noise reduction was not embraced as enthusiastically as for the professional audio industry and Dolby decided that if it was to make serious ground in the film industry it was the number of channels available, and not solely the sound quality that would gain success. In 1975 Dolby made public their film sound breakthrough. Using the same optical technology as was already in place, a new four-channel stereo system was introduced (Dolby Labs, 2002). It worked by storing just two channels of audio which represented the left and right speaker feeds. Then, the sum of these two channels represented the centre channel, and the difference between these two signals represented the surround feed. These principles were updated slightly due to the nature of the storage mechanism and replay situations.

1. Due to the potential phase misalignment and other analogue imperfections in the replay medium, high frequency sounds intended for the centre front speaker could leak back into the surround speakers. For this reason, the surround channels were band limited to around 7 kHz.
2. The surround speakers found in cinemas were often closer to the listener than the front speakers were. To make sure that the precedence effect didn't pull much of the imaging to the back and sides, the surround feeds were delayed.
3. The surround feed was phase shifted by +/- 90^0 prior to being added to the left and right channels. This meant that any material added to the surround channel would be summed, equally out of phase, with the left and right channels (as opposed to one in phase, one out of phase).

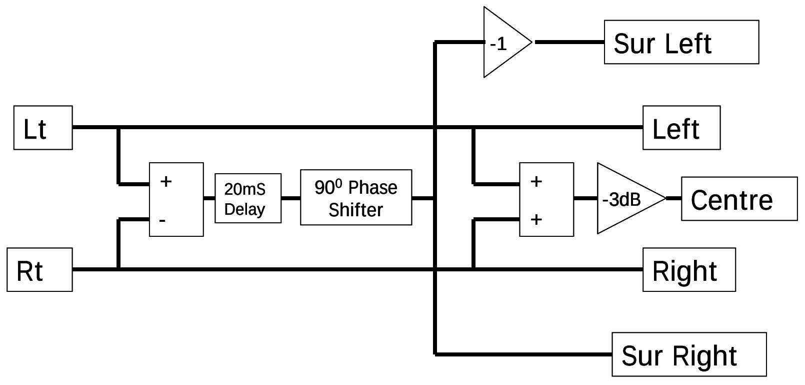
A simplified block diagram of the Dolby encode/decode process is shown in Figure 3.10. This, matrix, surround sound technique had a number of points in its favour:

1. It could be distributed using just two channels of audio
2. It was still an optical, and therefore cheap and robust, recording method.
3. The stereo track was mono compatible.
4. A new curve characteristic was used which, when coupled with Dolby noise reduction, greatly improved the fidelity of cinema sound.

For these reasons, the film industry took to the new Dolby Stereo format.



Dolby Stereo Encoding Process



Dolby Stereo Decoding Process

Figure 3.10 Simplified block diagram of the Dolby Stereo encode/decode process

3.2.7 Quadraphonics

While Dolby was concentrating on film sound reproduction, surround sound techniques were being developed for a wider audience (in the home) and the first of these systems was termed Quadraphonics. Quadraphonics worked on the principle that if the listener wanted to be surrounded by sound then all that would be needed was an extension of the stereo panning law described above, but moving between four loudspeakers. The loudspeakers were setup in a square (usually) and sounds could theoretically be pair-wise panned to any azimuth around the listener. However, it was soon shown that +/- 45^0 was too wide a panning angle at the front and back, and side images could not be formed satisfactorily using pair-wise panning techniques (Gerzon, 1974b & 1985). This, coupled with a number of incompatible formats, the extra expense needed for more speakers/amplifiers and the poor performance of early Quadraphonic matrix decoders meant that Quadraphonics was not a commercial success.

3.3 Review of Present Surround Sound Techniques

This section describes systems that are now still generating work and interest within the surround sound community (not necessarily any newer than some systems mentioned in section 3.2).

Systems in use today can be separated into two distinct categories:

1. Systems that define a speaker layout and/or carrier medium but with no reference to how signals are captured and/recorded for the system. Examples include
   - Dolby Digital - Ac-3 (Dolby Labs, 2004)
   - DTS (Kramer, N.D.)
   - Meridian Lossless packaging (De Lancie, 1998)
2. Systems that define how material is captured and/or panned for replay over a specified speaker layout. Examples include
   - Ambisonics
   - Wavefield Synthesis
   - Ambiophonics

This thesis will concentrate on the systems in the 2nd of these categories, that define *how* material is captured and replayed over a system as the 1st type of system is just defining a standard for which the 2nd category of system could be applied to (for example, both DTS and Dolby Digital are both lossy, perceptual codecs used to efficiently store 6 discrete channels to be played over a standard, ITU, 5.1 speaker array)

3.3.1 Ambisonics

3.3.1.1 Theory

Ambisonics was a system pioneered mainly by Michael Gerzon and is based on the spherical harmonic decomposition of a sound field (Gerzon, 1974). In order to understand this last statement the fundamentals of Ambisonics are reviewed.

A definition for what makes a decoder *Ambisonic* can be found in Gerzon & Barton (1992) and their equivalent U.S. patent regarding Ambisonic decoders for irregular arrays (Gerzon & Barton, 1998), and states (slightly adapted to remove equations):

A decoder or reproduction system is defined to be Ambisonic if, for a centrally seated listening position, it is designed such that:

- The decoded velocity and energy vector angles agree and are substantially unchanged with frequency.
- At low frequencies (below around 400 Hz) the low frequency velocity vector magnitude is equal to 1 for all reproduced azimuths.
- At mid/high frequencies (between around 700 Hz and 4 kHz) the energy vector magnitude is substantially maximised across as large a part of the 360^0 sound stage as possible.

To understand these statements, the underlying concepts of Ambisonics will be explained, leading into a description of the velocity and energy vectors and their relevance to multi-speaker surround sound systems.

Ambisonics is a logical extension of Blumlein's binaural reproduction system (at least, after it's conception). Probably one of the most forward looking features of the Blumlein technique is that when using the two figure of eight capsules positioned perpendicular to each other, any other figure of eight response could be created (it was this fact that was utilised in Blumlein's spatial equalisation technique). For example, if we take the two figure of eight microphones shown in Figure 3.5, then any figure of eight microphone response can be constructed using the equations shown in Equation (3.2). Some example microphone responses have been plotted in Figure 3.11.

$$\begin{aligned}
Sum &= (L+R)/\sqrt{2} \\
Dif &= (L-R)/\sqrt{2} \\
Figure8 &= (\cos(\theta)\times Sum)+(\sin(\theta)\times Dif)
\end{aligned}$$

(3.2)

where: θ is the desired response angle.

L is the left facing figure of eight microphone.

R is the right facing figure of eight microphone.

Figure8 is the reconstructed figure of eight microphone.

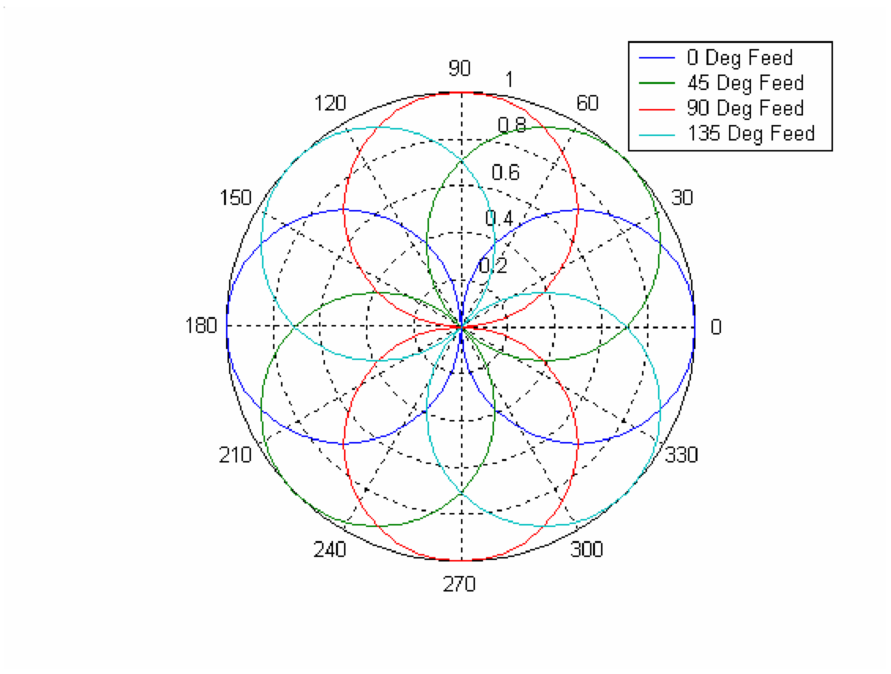


Figure 3.11 Plot of microphone responses derived from two figure of eight microphones.

This approach is very similar to Gerzon's in that the encoding (recording) side is independent from the decoding (reproduction) process. That is, Blumlein stereo could be replayed over 1, 2 or more speakers. Where Gerzon's Ambisonics improves upon this idea is as follows:

- Ambisonics can be used to recreate a full three dimensional sound field (i.e. height information can also be extracted from the Ambisonics system).
- The decoded polar pattern can be changed, that is, you are not fixed to using a figure of eight response.

As an example, 1st order Ambisonics can represent a sound field using four signals (collectively known as B-Format). The W signal is an omni-directional pressure signal that represents the zeroth order component of the sound field and X, Y and Z are figure of eight microphones used to record the particle velocity in any one of the three dimensions. Graphical representations of these four B-Format microphone signal responses are given in Figure 3.12.

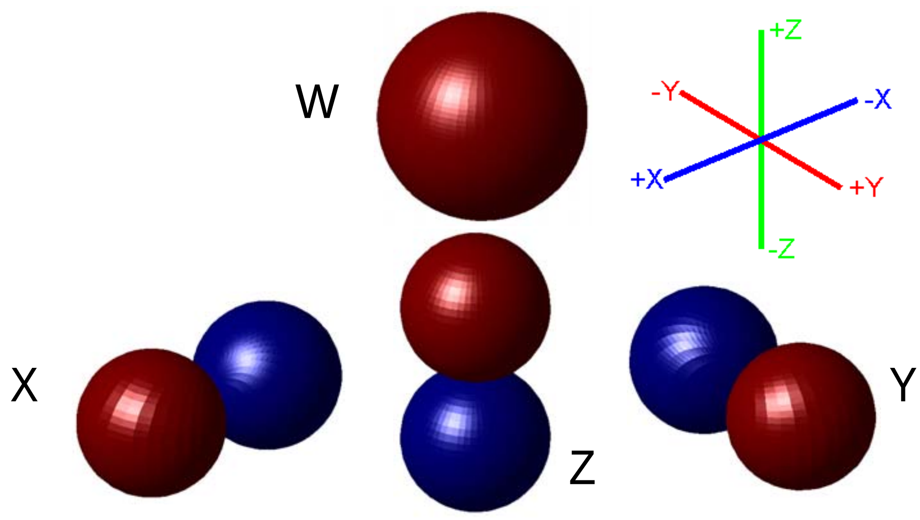


Figure 3.12 **The four microphone pickup patterns needed to record first order Ambisonics (note, red represents in-phase, and blue represents out-of-phase pickup).**

Ambisonics is a hierarchical format so that although four channels are needed for full three-dimensional reproduction, only three channels are needed if the final replay system is a horizontal only system. The mathematical equations representing the four microphone responses shown in Figure 3.12 are shown in equation (3.3). These equations can also be used to encode a sound source and represent the gains applied to the sound for each channel of the B-format signal.

$$\begin{aligned} W &= 1/\sqrt{2} \\ X &= \cos(\theta)\times\cos(\alpha) \\ Y &= \sin(\theta)\times\cos(\alpha) \\ Z &= \sin(\alpha) \end{aligned} \tag{3.3}$$

where: α = elevation angle of the source.

θ = azimuth angle of the source.

In order to replay a B-Format signal, virtual microphone responses are calculated and fed to each speaker. That is, using the B-format signals, any 1st order microphone response can be obtained pointing in any direction. As mentioned before, this is very much like the theory behind Blumlein Stereo, except that you can choose the virtual microphone response from any first order pattern (and not just a figure of eight), from omni to figure of eight. This is possible using the simple equation shown in equation (3.4) (Farina *et al.*, 2001)

$$
\begin{aligned}
&g_w = \sqrt{2}\\
&g_x = \cos(\theta)\cos(\alpha)\\
&g_y = \sin(\theta)\cos(\alpha)\\
&g_z = \sin(\alpha)\\
&S = 0.5 \times \left[(2-d)g_w W + d\left(g_x X + g_y Y + g_z Z\right)\right]
\end{aligned}
$$

(3.4)

where:	W,X,Y & Z are the B-format signals given in equation (3.3)
	S = speaker output
	θ = speaker azimuth
	α = speaker elevation
	d = directivity factor (0 to 2)

This gives us the flexibility to alter the polar pattern for each speaker in a decoder. Example patterns are shown in Figure 3.13.

To clarify the Ambisonic encode/decode process, let us encode a mono source at an azimuth of 35^0 and an elevation of 0^0 and replay this over a six speaker, hexagonal rig.

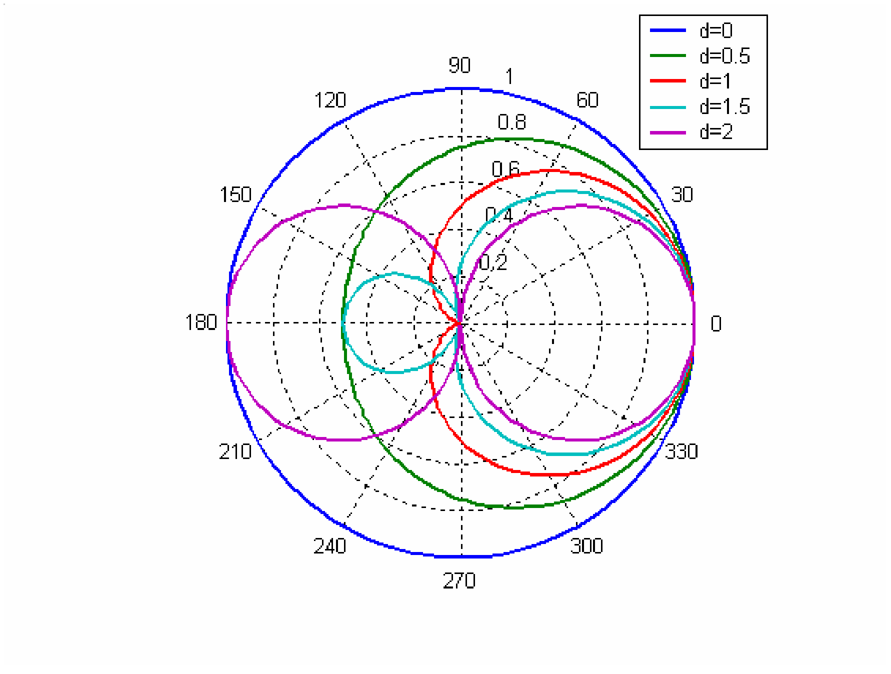


Figure 3.13 Graphical representation of the variable polar patterns available using first order Ambisonics (in 2 dimensions, in this case).

From equation (3.3) the B-format (W, X, Y and Z) signals will consist of the amplitude weighted signals shown in equation (3.5).

W = 0.7071 x mono

X = cos(35)cos(0) x mono = 0.8192 x mono

Y = sin(35)cos(0) x mono = 0.5736 x mono

Z = sin(0) x mono = 0 x mono

(3.5)

Where: mono is the sound source to be panned

W, X, Y & Z are the resulting B-Format signals after mono has had the directionally dependant amplitude weightings applied.

Equation (3.4) can now be used to decode this B-format signal. In this case a cardioid response will be used for each speaker's decoded feed, which equates to a directivity factor of 1 (see Figure 3.13). Equation (3.6) shows an example speaker feed for a speaker located at 150^0 azimuth and 0^0 elevation.

S = 0.5 x [(1.414 x W) + (-0.866 x X) + (0.5 x Y) + (0 x Z)]

(3.6)

where: W, X & Y are the encoded B-Format signals.
S = resulting speaker feed

The polar pattern used for the decoder can be decided either by personal preference, that is, by some form of empirically derived setting, or by a theoretical calculation which obtains the optimum decoding scheme.

This leads us back to the original statement of what makes a system Ambisonic. Although the B-format input signal is the simplest to use for the Ambisonic system, the term Ambisonics is actually more associated with how a multi-channel decode can be obtained that maximises the accuracy of the reproduced sound field. The three statements given at the beginning of this section mention the energy and velocity vectors associated with a multi-speaker presentation, and it is using these that an Ambisonic decoder can be designed.

3.3.1.2 Psychoacoustic Decoder Design Using the Energy and Velocity Vectors.

Although Gerzon defined what makes a system Ambisonic, a number of different decoding types have been suggested both by Gerzon himself and by others (see Malham, 1998 and Farino & Uglotti, 1998). However, the theory behind Ambisonics is, as already mentioned, similar to Blumlein's original idea that in order to design a psychoacoustically correct reproduction system the two lateralisation parameters must be optimised with respect to a centrally seated listener (Gerzon, 1974).

Originally, Gerzon's work concentrated on regularly spaced arrays in two and three dimensions (such as square and cuboid arrays) where the virtual microphone responses chosen for the decoders were based on the system being quantified using the principles of energy and velocity vectors calculated at the centre of the array to be designed. These two vectors have been shown to estimate the perceived localisation and quality of a virtual source when reproduced using multiple speakers (Gerzon, 1992c). The equations used to calculate the energy and velocity vectors are shown in Equation (3.7) with the vector lengths representing a measure of the 'quality' of localisation, and the vector angle representing the direction that the sound is perceived to originate from, with a vector length of one indicating a good localisation effect.

$$
\begin{aligned}
P &= \sum_{i=1}^{n} g_i & E &= \sum_{i=1}^{n} g_i^2 \\
Vx &= \sum_{i=0}^{n} g_i \cos(\theta_i)/P & Ex &= \sum_{i=0}^{n} g_i^2 \cos(\theta_i)/E \\
Vy &= \sum_{i=0}^{n} g_i \sin(\theta_i)/P & Ey &= \sum_{i=0}^{n} g_i^2 \sin(\theta_i)/E
\end{aligned}
$$

(3.7)

Where:
g_i represents the gain of the ith speaker (assumed real for simplicity).
n is the number of speakers.
θ_i is the angular position of the ith speaker.

These equations use the gain of the speakers in the array, when decoding a virtual source from many directions around the unit circle (each speaker's gain can be calculated using the B-Format encoding equations given in Equation (3.3) combined with the decoding equation given in Equation (3.4)).

For regular arrays, as long as the virtual microphone responses used to feed the speakers were the same for all, the following points can be observed:

- The reproduced angle would *always* be the same as the source's encoded angle.
- The energy (E) and pressure (P) values (which indicate the perceived volume of a reproduced source) would always be the same for any reproduced angle.

This meant that when optimising a decoder designed to feed a regular array of speakers:

- Only the length of the velocity and energy vectors had to be optimised (made as close to 1 as possible).

- This could be achieved by simply changing the pattern control (d) in equation (3.4) differently for low (<700Hz) and high (>700Hz) frequencies.

As an example Figure 3.14 shows the velocity and energy vector plots of an eight speaker horizontal Ambisonic array using virtual cardioid responses for each speaker feed.

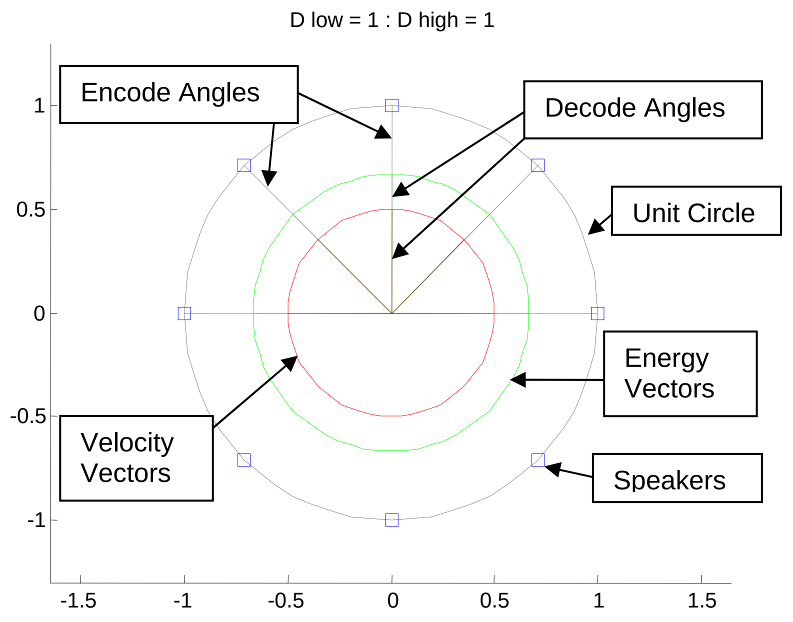


Figure 3.14 **Velocity and Energy Vector plot of an eight-speaker array using virtual cardioids (low and high frequency directivity of d=1).**

In order to maximise the performance of this decoder according to Gerzon's methods, the low frequency (velocity) vector length should be 1, and the high frequency (energy) vector length should be as close to 1 as possible (it is impossible to realise a virtual source with a energy vector of one, as more than one source is reproducing it). This can be achieved by using a low frequency directivity pattern of d=1.33 and a high frequency directivity pattern of d=1.15. This produces the virtual microphone patterns as shown in Figure 3.15 (showing the low frequency pattern for a speaker at 0^0 and a high frequency pattern for a speaker at 180^0 in order to make each pattern easier to observe) and has a corresponding velocity and energy vector plot as shown in Figure 3.16.

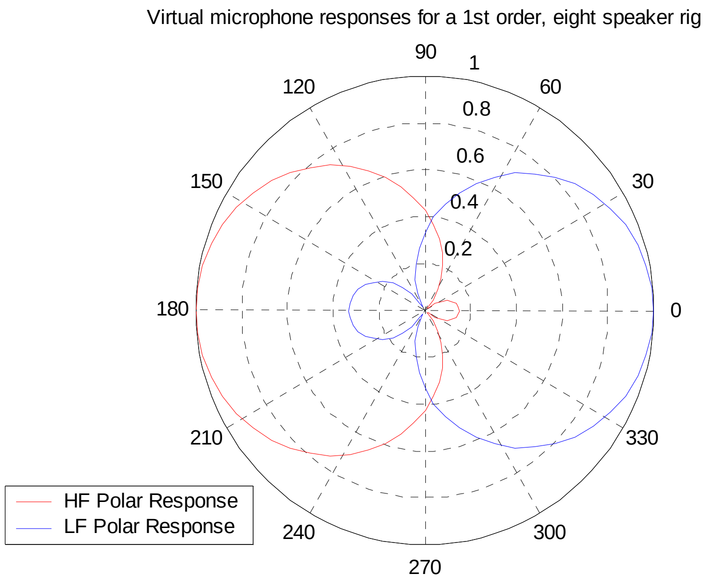


Figure 3.15 **Virtual microphone responses that maximise the energy and velocity vector responses for an eight speaker rig (shown at 0^0 and 180^0 for clarity).**

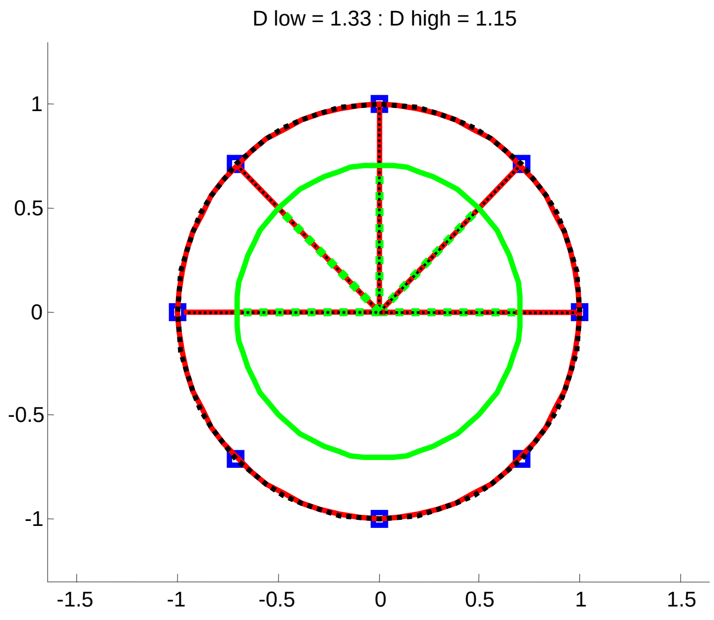


Figure 3.16 **Velocity and Energy Vector plot of an eight speaker Ambisonic decode using the low and high frequency polar patterns shown in Figure 3.16.**

As can be seen in Equation (3.4), a change of polar pattern in the decoding equation will result in two gain offsets; one applied to the W signal, and another applied to the X, Y and Z signals. This could be realised, algorithmically, by the use of shelving filters boosting and cutting the W, X, Y and Z signals by the desired amount prior to decoding, which simplified the design of, what was at the time, an analogue decoder.

It soon became apparent that, due to both the cinema and proposals for high definition television, the standard speaker layout for use in the home was not going to be a regular array. Gerzon had always had difficulty in solving the velocity and energy vector equations for irregular arrays because irregular arrays would generally need optimising, not only for the vector lengths, but also for the decoded source angles and the perceived volume of the decoder (for example, more speakers in the front hemisphere, when compared to the rear, would cause sources to be louder when in that hemisphere). This meant that a set of non-linear simultaneous equations needed to be solved. Also, the shelving filter technique used for regular decoders could not be used for irregular decoders as it was not just the polar pattern of the virtual microphones that needed to be altered. To this end a paper was published in 1992 (Gerzon & Barton, 1992) describing how a cross-over filter technique could be used along with two decoder designs, one for the low frequency and one for the high frequencies, in order to solve the irregular speaker problem.

In the Gerzon & Barton (1992) paper, a number of irregular Ambisonic decoders were designed, however, although many five speaker decoder examples were given, none were as irregular as the ITU finally specified. For example, the front and rear spacing of the ITU layout are +/- 30^0 from straight ahead and +/- 70^0 from directly behind the listener, respectively, but the decoders Gerzon designed always had a front and rear spacing that were similar to each other (e.g. +/-35^0 front and +/- 45^0 rear) and although much work has been carried out on Ambisonics, a psychoacoustically correct 'Vienna style' decoder (named after the AES conference in Vienna where the Gerzon & Barton paper was presented) has not yet been calculated. It must also be noted that Gerzon's method for solving these equations was, by his own admission, are "*very tedious and messy*" (Gerzon & Barton, 1992) and it can be observed, by visualising the velocity and vector responses, in a similar manner to Figure 3.16, that this paper does not solve the equations optimally.

This is due to the splitting of the encoding and the decoding by Gerzon. An example of a decoder optimised by Gerzon & Barton is shown in Figure 3.17

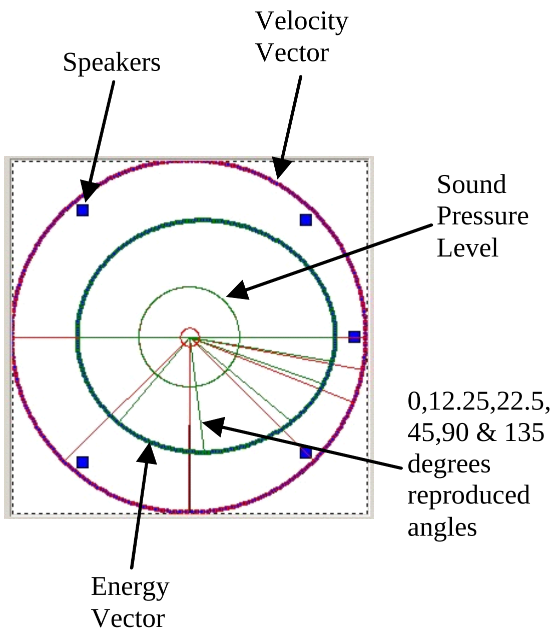


Figure 3.17 Energy and velocity vector analysis of an irregular speaker decode optimised by Gerzon & Barton (1992).

It can be clearly seen, in Figure 3.17, that the high frequency decode (green line representing the energy vector) has reproduced angles that do not match up with the low frequency velocity vector response. This is due to the fact that the Gerzon & Barton paper suggests that although the vector length and reproduced angle parameters should be optimised simultaneously for the high frequency energy vector, a forward dominance adjustment (transformation of the B-format input signal) should then be carried out to ensure that perceived volume of the high frequency decoder is not biased towards the back of the speaker array. This, inevitably, causes the reproduced angles to be shifted forward.

3.3.1.3 B-Format Encoding

The encoding equations (3.3) are basically a simulation of a B-format microphone (such as the SoundField Microphone, SoundField Ltd., n.d.) which has a four-channel response as shown in Figure 3.12. However, recording coincidentally in three dimensions proves to be extremely difficult. Coincident microphone techniques in two dimensions (see 3.2.2, Blumlein's , page 36) are possible where the microphones can be made coincident in the

X – Y axis but not in the Z axis (although this still causes some mis-alignment problems); however, in three dimensions this is not desirable as recording needs to be equally accurate in all three dimensions. This problem was solved by Gerzon and Craven (Craven & Gerzon, 1977) by the use of four sub cardioid microphone capsules mounted in a tetrahedral arrangement. This arrangement is shown in Figure 3.18.

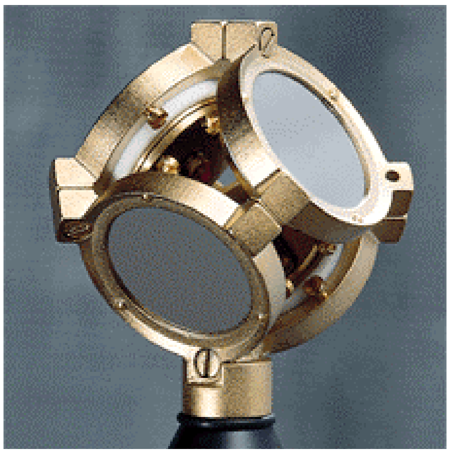


Figure 3.18 Four microphone capsules in a tetrahedral arrangement.

The capsules are not exactly coincident, but they are equally non-coincident in each axis' direction, which is important as this will simplify the correction of the non-coincident response. However, to aid in the explanation of the principles of operation of this microphone the capsule responses will, for now, be assumed to be exactly coincident and of cardioid response. As shown in Figure 3.18, each of the four microphone capsules faces in a different direction:

Capsule	Azimuth	Elevation
A	45^0	35.3^0
B	135^0	-35.3^0
C	-45^0	-35.3^0
D	-135^0	35.3^0

Table 3.1 SoundField Microphone Capsule Orientation

As each of the capsules has a cardioid pattern (in this example) all sound that the capsules pick up will be in phase. Simple manipulations can be performed on these four capsules (know collectively as A-format) so as to construct the four pick-up patterns of B-format as shown in equation (3.8). A graphical representation of the four cardioid capsule responses and the four first order components derived from these are shown in Figure 3.19.

$$
\begin{aligned}
W &= 0.5 \times (A + B + C + D) \\
X &= (A + C) - (B + D) \\
Y &= (A + B) - (C + D) \\
Z &= (A + D) - (B + C)
\end{aligned}
\tag{3.8}
$$

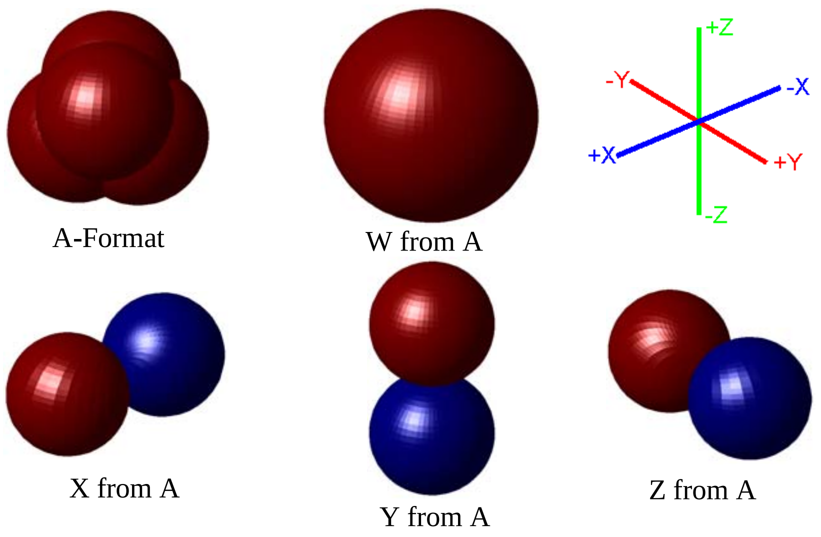


Figure 3.19 **B-Format spherical harmonics derived from the four cardioid capsules of an A-format microphone (assuming perfect coincidence). Red represents in-phase and blue represents out-of-phase pickup.**

As is evident from Figure 3.19, four perfectly coincident cardioid microphone capsules arranged as described above can perfectly recreate a first order, B-format, signal. However, as mentioned earlier, the four capsules providing the A-format signals are not perfectly coincident. This has the effect of misaligning the capsules in time/phase (they are so close that they do not significantly affect the amplitude response of the capsules), which results in colouration (filtering) of the resulting B-format signals. As all of the capsules are equally non-coincident then any colouration will be the same for each order, i.e. the 0th order component will be filtered in one way, and the 1st order components will be filtered in another way. However, using cardioid microphone pickup patterns causes the frequency response of the B-format signals to fluctuate too much, and so for the actual implementation of the microphone, sub-cardioid polar patterns were used (as shown in Figure 3.20).

To illustrate the frequency response characteristics of an Ambisonic microphone, it is simpler to assume that the microphone only works horizontally. Each of the four sub-cardioid capsules has no elevation angle, only an azimuth as described earlier. The equations that construct W, X, and Y will still be the same (3.8), but the Z component will not be constructed. Figure 3.20 shows a number of representations of a sound being recorded from four different directions, 0^0, 15^0, 30^0 and 45^0 and indicates what amplitude each capsule will record, what timing mismatches will be present (although, note that the sample scaling of this figure is over-sampled many times), and finally a frequency response for the W and X signals. It can be seen that the two channels not only have different frequency responses, but also these responses change as the source moves around the microphone. It must be remembered that the overall amplitude of the X channel will change due to the fact that the X channel has a figure of eight response. Looking at Figure 3.20 shows a clear problem with having the capsules spaced in this way, and that is the fact that the frequency response of the B-format signals changes as the source moves around the microphone. The smaller the spacing, the less of a problem it becomes (as the changes move up in frequency due to the shortening of the wavelengths when compared to the spacing of the capsules), and Figure 3.20 is based on the approximate spacing that it part of the SoundField MKV microphone (Farrah, 1979a).

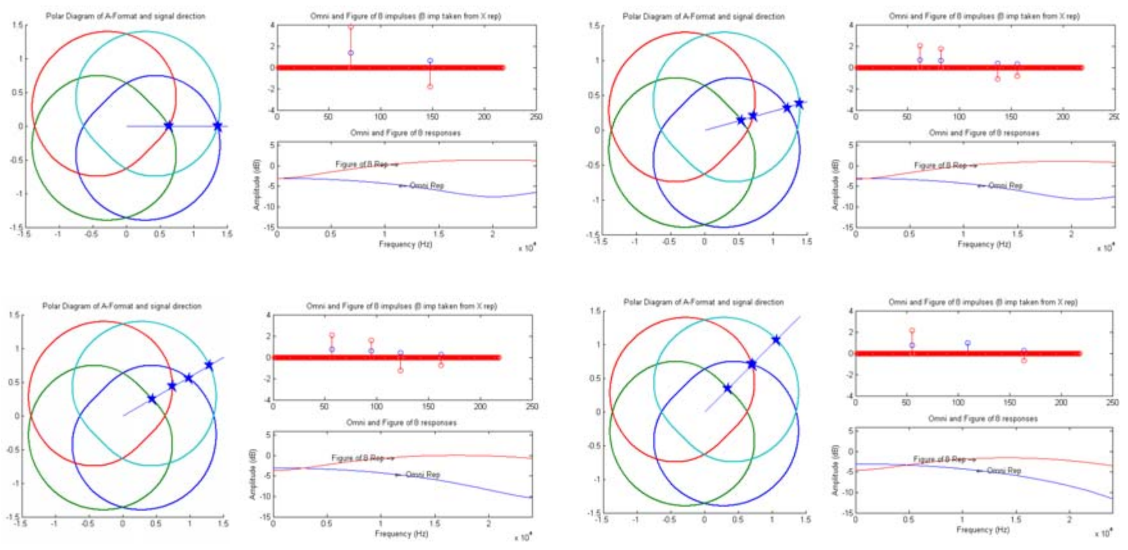


Figure 3.20 **Simulated frequency responses of a two-dimensional, multi-capsule A-format to B-format processing using a capsule spacing radius of 1.2cm.**

These responses can be corrected using filtering techniques, but only the average response will be correct, with the sound changing timbrally as it is moved around the microphone.

Although the frequency response deviations sound like a large problem, they are not noticed and are combined with other errors in the signal chain such as microphone capsule imperfections and loudspeaker responses. Also Farrah (1979b) claims that similar coincident stereo techniques have a far greater error than the SoundField microphone anyway – "*Closeness of the array allows compensations to be applied to produce B-format signal components effectively coincident up to about 10 kHz. This contrasts vividly with conventional stereo microphones where capsule spacing restricts coincident signals up to about 1.5 kHz*". What is being referred to here is the frequency at which the filtering becomes non-constant. If the graphs in the omni-directional signal response are observed, it can be seen that its frequency response remains constant up to around 15 kHz, and it is the spacing of the capsules that defines this frequency. The closer the capsules, the higher the frequency until non-uniformity is observed.

The SoundField microphone has many advantages over other multi-channel microphone techniques, with the main advantage being the obvious one in that it is just one microphone, and therefore needs no lining up with other microphones. Also, *any* combination of coincident first order microphones can be extracted from the B-format signals, which implies that the B-format signal itself can be manipulated, and this is indeed true. Manipulations including rotation, tumble and tilt are possible (Malham, 1998) along with being able to zoom (Malham 1998) into a part of the sound field, which alters the balance along any axis. Equations for these manipulations are given in (3.9).

X – Zoom	Rotation about Z	Rotation about X
$W' = W + \frac{1}{\sqrt{2}} \cdot d \cdot X$	$W' = W$	$W' = W$
$X' = X + \sqrt{2} \cdot d \cdot W$	$X' = X \cdot \cos(\theta) + Y \cdot \sin(\theta)$	$X' = X$
$Y' = \sqrt{1-d^2} \cdot Y$	$Y' = Y \cdot \cos(\theta) - X \cdot \sin(\theta)$	$Y' = Y \cdot \cos(\theta) - Z \cdot \sin(\theta)$
$Z' = \sqrt{1-d^2} Z$	$Z' = Z$	$Z' = Z \cdot \cos(\theta) + Y \cdot \sin(\theta)$

(3.9)

where d is the dominance parameter (from –1 to 1).

θ is the angle of rotation.

A graphical representation of the effect that the zoom, or dominance, control has on the horizontal B-format polar patterns is shown in Figure 3.21.

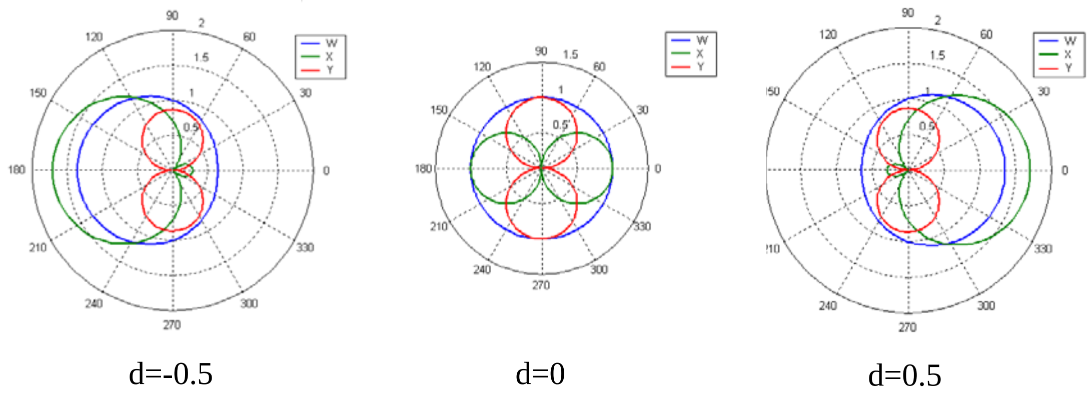


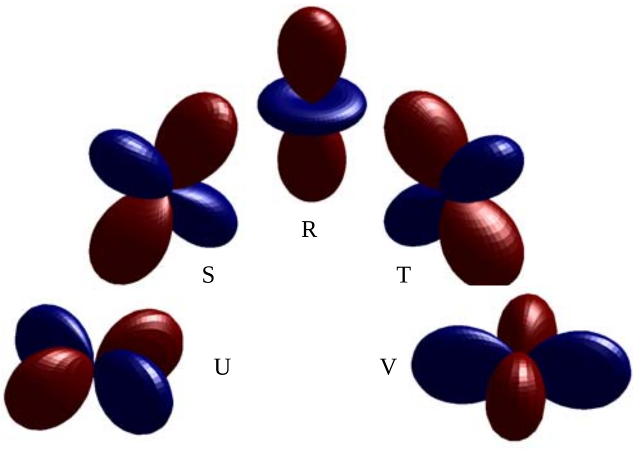
Figure 3.21 Effect of B-format zoom parameter on W, X, and Y signals.

As is evident from Figure 3.21 and Equation (3.9), the dominance parameter works by contaminating the W signal with the X signal and visa versa, which means that any speaker feeds taking in X and W will have these signals exaggerated if both are in phase, or cancelled out, if both are out of phase with each other. This coupled with the attenuation of the Y and Z channels means that any derived speaker feeds/virtual microphone patterns will be biased towards the X axis. Dominance in the Y and Z directions can also be achieved in the same way.

3.3.1.4 Higher Order Ambisonics

Ambisonics is a very flexible system with its only main drawback being that only a first order microphone system is commercially available (however, it must be noted that *all* commercially available microphones have a first order polar pattern at present). However, as the name first order suggests, higher order signals can be used in the Ambisonics system, and the theory needed to record higher order circular harmonics has been discussed in a paper by Mark Poletti (Poletti, 2000). A 2nd order system has nine channels for full periphony (as opposed to the four channels of 1st order) and five channels for horizontal only recording and reproduction (as opposed to three channels for 1st order). The equations for the nine 2nd order channels are given in (3.10) (Furse, n.d.).

$$
\begin{aligned}
W &= 1/\sqrt{2} \\
X &= \cos(\theta)\times\cos(\alpha) \\
Y &= \sin(\theta)\times\cos(\alpha) \\
Z &= \sin(\alpha) \\
R &= 1.5\times\sin^2(\alpha)-0.5 \\
S &= \cos(\theta)\times\sin(2\alpha) \\
T &= \sin(\theta)\times\sin(2\alpha) \\
U &= \cos(2\theta)\times\cos^2(\alpha) \\
V &= \sin(2\theta)\times\cos^2(\alpha)
\end{aligned}
$$



where: α = elevation angle of the source.

θ = azimuth angle of the source.

(3.10)

For horizontal only work α is fixed at zero which makes the Z, R, S, & T channels hold at zero, meaning that only W, X, Y, U & V are used. To demonstrate the difference in polar patterns (horizontally) between 1st, 2nd, 3rd and 4th order polar patterns (using equal weightings of each order), see Figure 3.22.

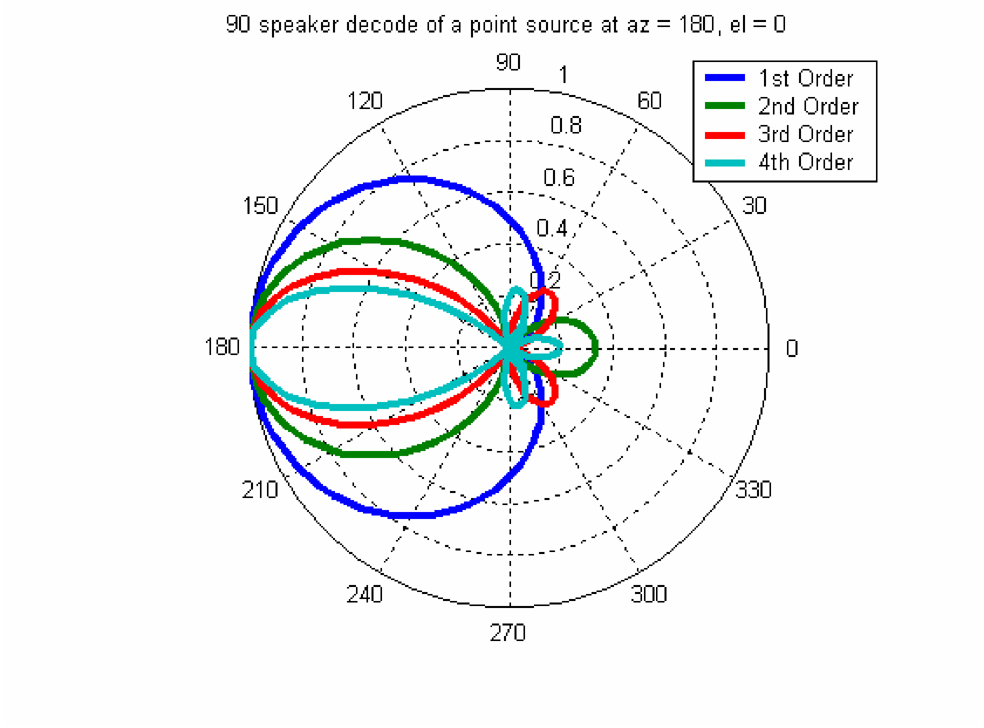


Figure 3.22 Four different decodes of a point source polar patterns of 1st, 2nd, 3rd & 4th order systems (using virtual cardioid pattern as a 1st order reference and equal weightings of each order). Calculated using formula based on equation (3.4), using an azimuth of 180^0 and an elevation of 0^0 and a directivity factor (d) of 1.

Higher order polar patterns, when decoded, do not imply that fewer speakers are working at the same time; they are just working in a different way to reconstruct the original sound field. Figure 3.23 shows the decoded levels for an infinite number of speakers placed on the unit circle. The virtual source is placed at 180^0 and the virtual decoder polar pattern is set to that shown in Figure 3.22. The multiple lobes can clearly be seen at 180^0 for the second order decode and at approximately 130^0 and 250^0 for the third order decode. Note that the peak at the source position is not necessarily the same for each Ambisonic order (the responses were scaled in Figure 3.22, but this is a decoder issue), but the sum of all the decoder feeds (divided by the number of speakers) is equal to 1 for each order. This means that the measured pressure value at the middle of the speaker array will be consistent.

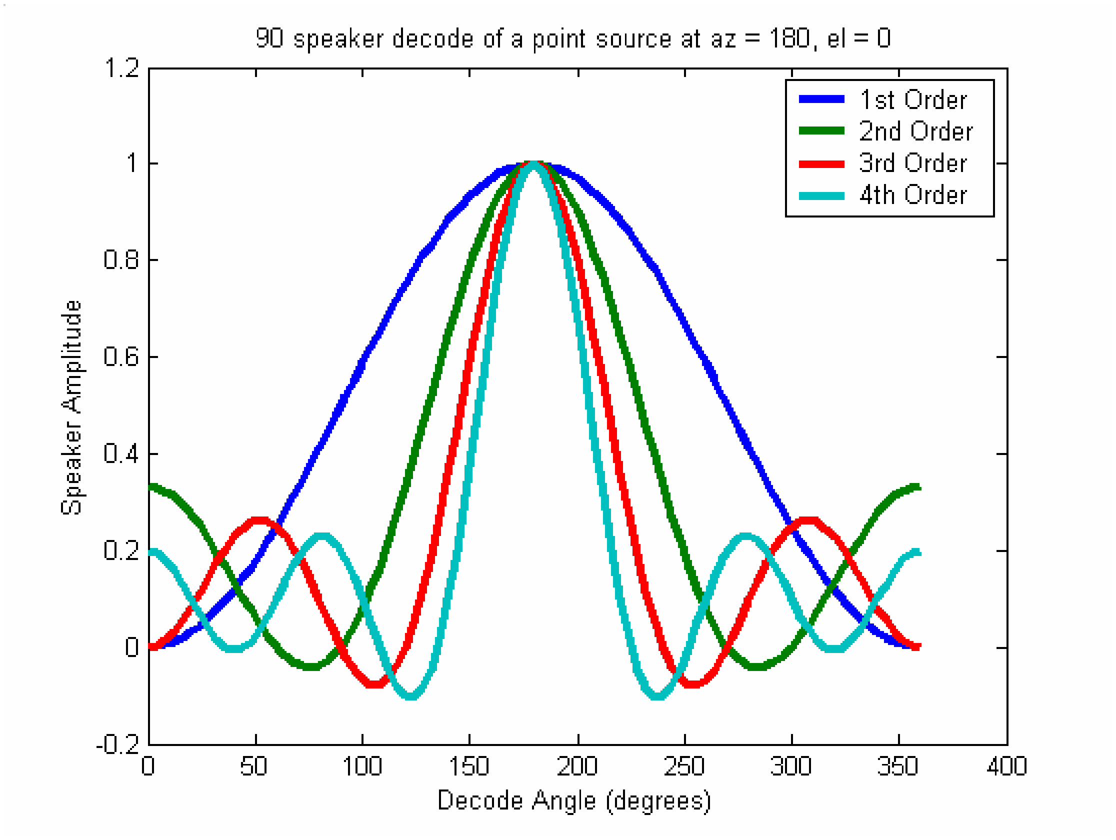


Figure 3.23 **An infinite speaker decoding of a 1st, 2nd, 3rd & 4th order Ambisonic source at 180^0. The decoder's virtual microphone pattern for each order is shown in Figure 3.22.**

One point not mentioned so far is that there are a minimum number of speakers needed to successfully reproduce each Ambisonic order, which is always greater than the number of transmission channels available for the decoder (Gerzon, 1985). This problem can be compared with the aliasing problem in digital audio, that is, enough 'samples' must be used in the reproduction array in order to reproduce the curves shown in Figure 3.23. For example, if we take a 1st and a 2nd order signal and reproduce this over four speakers (knowing that a 2nd order signal will need at least six speakers to be reproduced correctly) then the amplitude of the signals at the four speakers is shown in Figure 3.24. It can clearly be seen that speakers two and four (at 90^0 and 270^0 respectively) have no output, whereas speaker 3 (positioned at 180^0) has an amplitude of 1, coupled with the opposite speaker (at 0^0) having an output amplitude of 1/3.

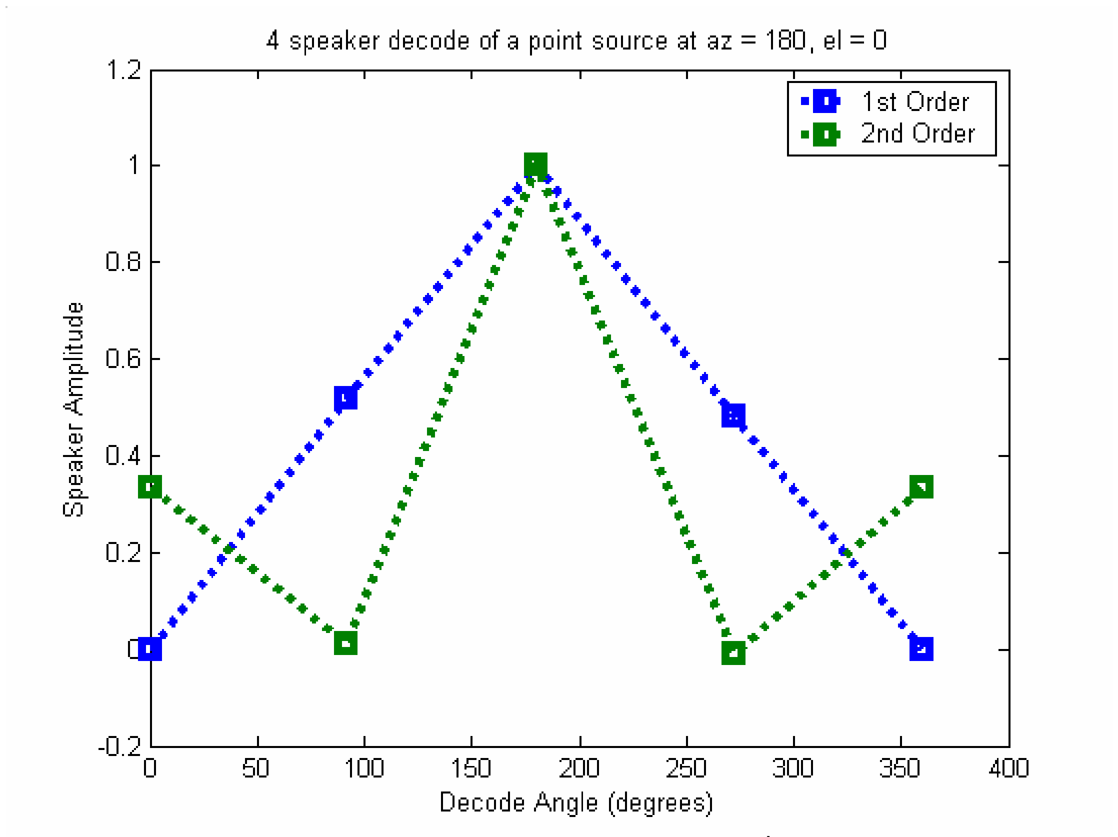


Figure 3.24 **Graph of the speaker outputs for a 1st and 2nd order signal, using four speakers (last point is a repeat of the first, i.e. $0^0/360^0$) and a source position of 180^0.**

This will result in the image pulling towards one speaker when the source position is near that direction. This is also shown in the research by Gerzon (1985) and will cause the decoding to favour the directions at the speaker locations. This is detrimental to the reproduced sound field as one of the resounding features of Ambisonics is that all directions are given a constant error, making the speakers 'disappear', which is one reason as to why Ambisonics can give such a natural sounding reproduction.

Recent work by Craven (2003) has now described a panning law (as described in the paper, which is analogous to an Ambisonic decoder) for irregular speaker arrays using 4th order circular harmonics. This uses the velocity and energy vector theories mentioned above to optimise the decoder for the ITU irregular 5-speaker array. What is interesting about this decoder is that although 4th order circular harmonics are used, the polar patterns used for the virtual microphone signals are not strictly 4th order (as shown in Figure 3.22) but are 'contaminated' with 2nd, 3rd and 4th order components in order to steer the virtual microphone polar patterns so that the performance of the decoder is maximised (which means having a high order front and low order rear decode, dependant on speaker density). The velocity and energy vector analysis of the 4th order decoder used by Craven (2003) can be found in Figure 3.25 and the corresponding virtual microphone patterns can be seen in Figure 3.26.

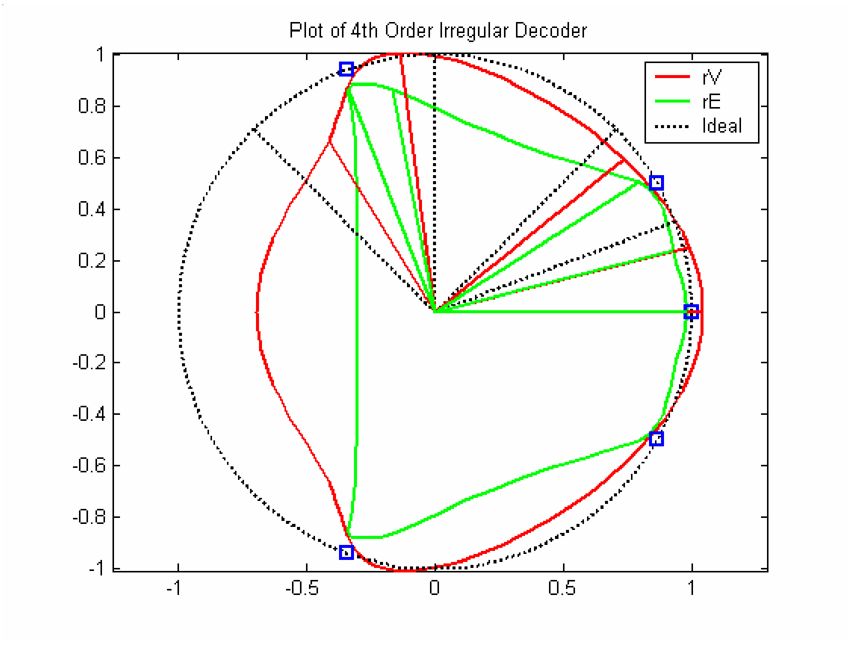


Figure 3.25 **Energy and Velocity Vector Analysis of a 4th Order Ambisonic decoder for use with the ITU irregular speaker array, as proposed by Craven (2003).**

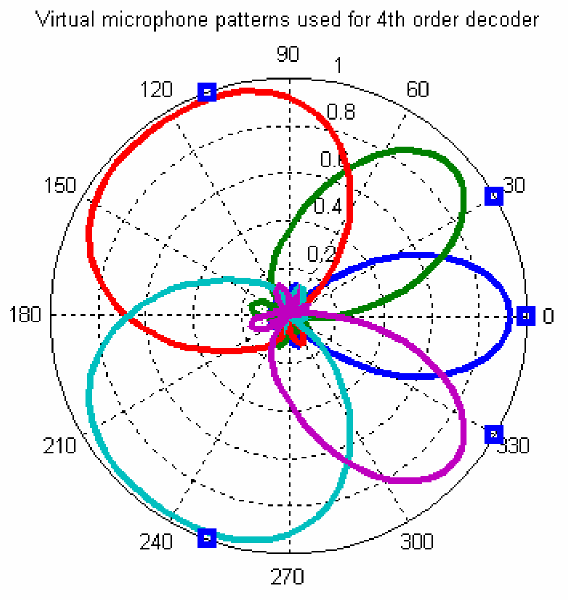


Figure 3.26 **Virtual microphone patterns used for the irregular Ambisonic decoder as shown in Figure 3.25.**

It must also be noted that a number of researchers have now started to work on much higher orders of Ambisonics (for example, 18th order) and it is at these orders that Ambisonics does, indeed, tend towards a system similar to wavefield synthesis (see Sontacchi & Holdrich, 2003 and Daniel *et al.*, 2003) and although these, much higher order systems, will not be utilised in this report, the underlying principles remain the same.

3.3.1.5 Summary

Ambisonics is an ideal system to work with for a number of reasons:

- It has both a well defined storage format and simple synthesis equations, making it useful for both recording/mixing and real-time synthesis.
- The encoding is separated from the decoding resulting in a system where decoders can be designed for different speaker arrays.
- The design of a decoder is based on approximations to what a centrally seated listener will receive, in terms of phase and level differences between the ears at low and high frequencies. This makes it an ideal choice for a system that can be converted to binaural and transaural reproduction.

However, a number of issues are apparent:

- The optimisation of a frequency dependant 1st order decoder for use with the ITU 5 speaker array has not been achieved with the technique of solving the non-linear simultaneous equations representing the velocity and energy vectors being both laborious and leading to non-ideal results.
- This process will only become more complicated when more speakers are added (Gerzon & Barton, 1992 and Gerzon & Barton, 1998).
- The energy and velocity vectors are low order approximations to the actual head related signals arriving at the ear of the listener. The analysis and design of Ambisonic decoders could, potentially, be improved through the use of head related data directly.

3.3.2 Wavefield Synthesis

3.3.2.1 Theory

Although this research concentrates on the Ambisonic form of speaker surround sound, it is not necessarily because it is the most realistic in its listening experience. One of the most accurate forms of surround sound (from a multiple listener point-of-view) is termed Wavefield Synthesis. In its simplest form Wavefield Synthesis is the system first tried by Bell Labs mentioned at the beginning of this chapter (Rumsey and McCormick, 1994); however, the theory and underlying principles of Wavefield Synthesis have been studied, the mathematical transfer functions calculated and a theoretical understanding of the necessary signal processing involved in such a system have been developed. The result is that individual sources can be synthesised, simulating both angular placement and distance (with distance being the cue that is, perhaps, hardest to recreate using other multi-speaker reproduction systems).

Wavefield synthesis is different from most other multi-speaker surround sound systems in a number of ways:

- It is a volume solution, that is, there is no ‘sweet spot’, with an equal reproduction quality experienced over a wide listening area.
- Distance simulation is very well suited to Wavefield Synthesis. This is a difficult cue to simulate using other forms of multi-channel sound.
- The resulting acoustic waves, rather than the source itself, are synthesised.

Wavefield Synthesis (and the Bell Labs version before it) is based on Huygen’s principle[1]. Put simply this states that any wave front can be recreated by using any number of point sources that lie on the original wave. This implies that to recreate a plane wave (i.e. a source at an infinite distance from the listener) a line-array of speakers must be used, but to create a

[1] The principle that any point on a wave front of light may be regarded as the source of secondary waves and that the surface that is tangent to the secondary waves can be used to determine the future position of the wave front.

spherical wave (more like the waves heard in real life) an arc of speakers must be used. However, where Wavefield Synthesis' innovation lies is that the necessary transfer functions have been calculated, and a line array of speakers can synthesise both of these situations using a mixture of time delays and amplitude scaling (a transfer function). It is often thought that Ambisonics is spherical Wavefield Synthesis on a lesser scale, and Bamford (1995) has analysed it in this way (that is, as a volume solution, looking at how the well the sound waves are reconstructed); however, this is not strictly the case as no time differences are recorded (assuming perfectly coincident microphone capsules), or necessarily needed, and so it is more accurate to think of Ambisonics as more of an amplitude panning scheme (albeit, one based on more solid foundations than simple pair-wise schemes). This also suggests that the results from Bamford (1995) that state that first order Ambisonics is only 'correct' up to 216Hz (in a sweet spot 25cm wide) may be a simplification (and under-estimation) of the system's performance. In other words, this is a measure of an Ambisonics wavefield synthesis performance. Clearly, if Ambisonics only had a useable (spatially speaking) frequency of up to 216Hz, and a sweet spot 25cm wide, it would not be very useful for surround sound.

So what is the limiting factor for Wavefield Synthesis? Due to the finite number of points used to recreate a sound wave, this system is limited by its 'Spatial Aliasing Frequency' (Berkhout *et al.,* 1992). The equation for this (although note that this is for a plane wave) is given in Equation (3.11) (Verheijen *et al.*, 1995).

$$f_{Nyq} = \frac{c}{2\Delta x \sin(\theta)}$$

(3.11)

where:	f_{Nyq}	=	Limiting Nyquist Frequency.
	Δx	=	Speaker spacing.
	θ	=	Angle of radiation.
	c	=	Speed of sound in air ($\approx 342\text{ms}^{-1}$)

It must be noted that although Wavefield Synthesis has a limiting frequency, this is its Spatial Aliasing limit. That is, the system can reproduce sounds of full bandwidth, however, accurate reproduction can only be correctly achieved (theoretically) below this frequency (which is, incidentally, the reason Bell Labs early simplification of their original multi-mike, multi-speaker array did not work as hoped when the number of speakers was reduced). It can also be seen that the limiting frequency is inversely proportional to the angle of radiation. To understand the reasons behind this, an example is shown in Figure 3.27.

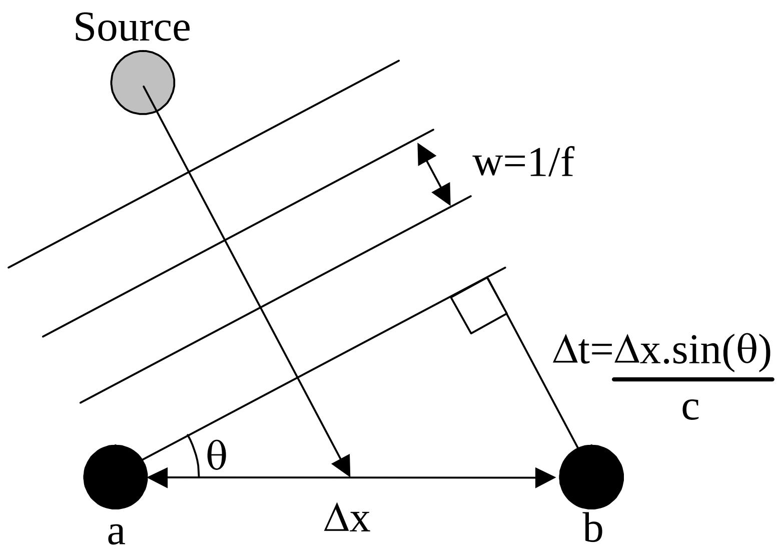


Figure 3.27 The effect that the angle of radiation has on the synthesis of a plane wave using Wavefield Synthesis.

Once the angle of radiation is changed to an off-centre value (i.e. non-zero) then the amount of time delay that is needed to correctly simulate the plane wave is increased, proportional to the distance between the speakers multiplied by the sine of the angle, θ. Once this time delay becomes more than half the wavelength of the source the superposition of the wave fronts creates artefacts that manifest themselves as interference patterns (Verheijen *et al.*, 1995). Filtering the transfer functions used to recreate the wave front (or using more directional loudspeakers (Verheijen *et al.*, 1995)) counteracts this.

3.3.2.2 Summary

Wavefield Synthesis is reported as being one of the most accurate forms of multi-channel sound available, but it does have some problems that make it an undesirable solution for this project:

- Huge amount of transducers needed to recreate horizontal surround sound (for example, the University of Erlangen-Nuremberg's experimental setup uses 24 speakers (University of Erlangen-Nuremberg, N.D) arranged as three sides of a square).
- The reproduction of three-dimensional sound is not yet possible using Wavefield Synthesis.
- Recording a sound field for reproduction using Wavefield Synthesis is difficult due to the high rejection needed for each direction. Synthesised material works much better (Verheijen *et al.*, 1995).
- Large amount of storage channels and processing power needed to provide loudspeakers with appropriate signals.

Also, there is not, as yet, a standard protocol for the storage and distribution of such material; although this is being worked on as part of the MPEG Carusso Project (Ircam, 2002). This lack of storage standard is not an issue, of course, for applications that calculate their acoustical source information on the fly, such as virtual reality systems.

3.3.3 Vector Based Amplitude Panning

3.3.3.1 Theory

Vector based amplitude panning (or V.B.A.P.) is an amplitude panning law for two or three dimensional speaker rigs, and was developed by Ville Pulkki. Once the speaker positions are known, the V.B.A.P. algorithm can then be used to decode the speaker rig using pair-wise (two dimensions) or triple-wise (three dimensions) panning techniques. An example of the two dimensional algorithm is shown in Figure 3.28 (Pulkki, 1997).

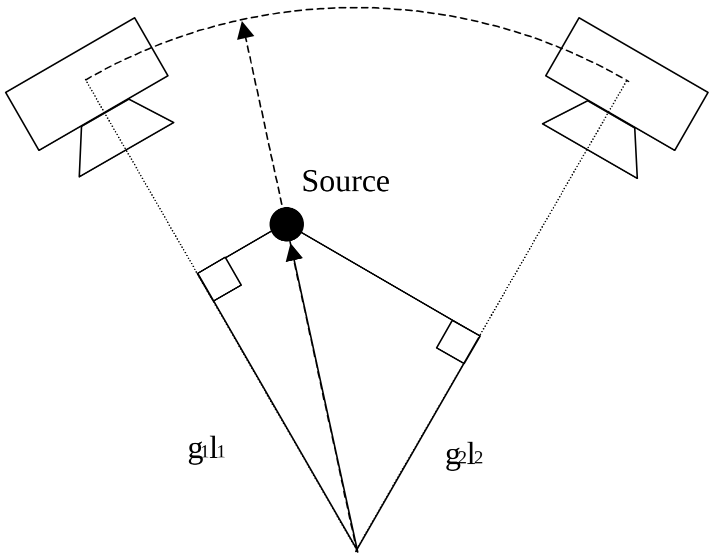
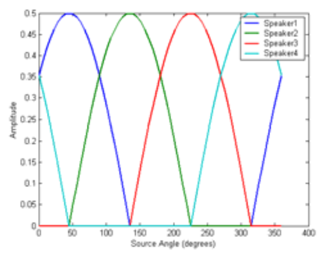
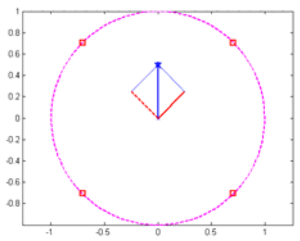


Figure 3.28 Graphical representation of the V.B.A.P. algorithm.

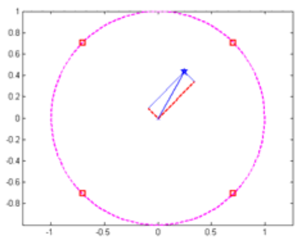
As can be seen in Figure 3.28, horizontal V.B.A.P. divides the source into its two component gains, in the direction of the loudspeakers, which are then used as the gains for the amount of the source that it supplied to each of the speakers. It must be noted, however, that the sources are limited to existing on the path between speakers by normalising the gain coefficients g_1 and g_2. To extend the system to three dimensions, triple-wise panning is used. An example decode of a source travelling from an angle of 0^0 to an angle of 120^0 is shown in Figure 3.29, along with the four un-normalised speaker gains. This system can work very well, mainly because the largest possible localisation error cannot be any more than one speaker away from where the source should be. However, as can be observed from Figure 3.29, a speaker detent effect will be noticed when a source position is in the same direction as a speaker as only that speaker will be replaying sound. This will create a more stable, and psychoacoustically correct virtual source (as it is now a *real* source) which will mean that the individual speakers will be heard with the sources potentially jumping from speaker to speaker if the spacing between the speakers it too great.



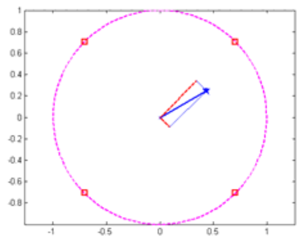
Speaker Amplitude



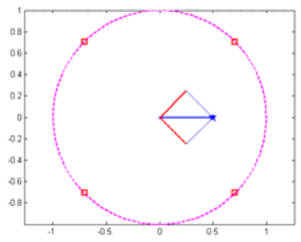
Source at 0^0



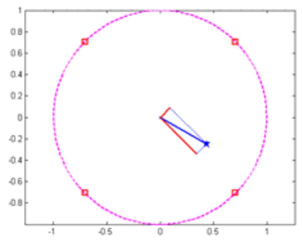
Source at 30^0



Source at 60^0



Source at 90^0



Source at 120^0

Figure 3.29 Simulation of a V.B.A.P. decode. Red squares – speakers, Blue pentagram – Source, Red lines – speaker gains.

3.3.3.2 Summary

VBAP is based around the simple pair-wise panning of standard stereo, although using the VBAP technique it can be easily used as a triple-wise, with-height system. To this end, a VBAP system comprising of a low number of speakers will suffer the same problems as other pair-wise panned systems (see Quadraphonics, section 3.2.7). However, as the number of speakers are increased, the accuracy of the system will improve, although side images will always suffer when compared to frontal images due to pair-wise panning techniques failing for speakers placed to the side of a listener (although the error will, again, lessen with increased speaker density).

For this project, however, VBAP is unsuitable as:

- VBAP has no storage format – all panning information is calculated when the material is replayed, as information regarding the speaker layout must be known.
- Any pre-decoded material can not have additional speaker feeds calculated according to the rules of VBAP.
- The decoded material is not optimised for a centrally seated listener, making the system sub-optimal if conversion to headphone or transaural systems is required.

3.3.4 Two Channel, Binaural, Surround Sound

Although all of the surround sound systems discussed so far have used more than two channels (many more, in some cases), it is possible to use only two channels. Such a system is termed binaural reproduction. As we only have two ears, then it seems reasonable that only two channels of audio are necessary to successfully fool the ear/brain system into thinking that it is experiencing a realistic, immersive, three dimensional sound experience. All of the speaker reproduction systems discussed so far have a number of marked limitations:

- System performance is normally proportional to the number of speakers used. The more speakers, the better the result.
- The sound from each speaker will reach both ears, making it a more involved task to control exactly what is being perceived by the listener.
- The final system is usually a compromise due to the above limitations.

Binaural sound circumvents these limitations with the use of headphones. As there is a one to one mapping of the ears to the transducers it is very easy to provide the ears with the signals necessary to provide convincing surround sound. Binaural sound reproduction works on the simple principle that if the ears are supplied with the same acoustical pressure that would have been present in real-life due to a real source, then the ear/brain system will be fooled into perceiving that a real source is actually there. As discussed in chapter 2, there are a number of auditory cues that the ear/brain system uses to localise a sound source, a number of which can be simulated using a head related transfer function (HRTF). An example pair of HRTFs are shown in Figure 3.30, and are taken from a KEMAR dummy head in an anechoic chamber by Gardner & Martin (1994). The source was at an angle of 45^0 from the centre of the head, and at a distance of 1 m.

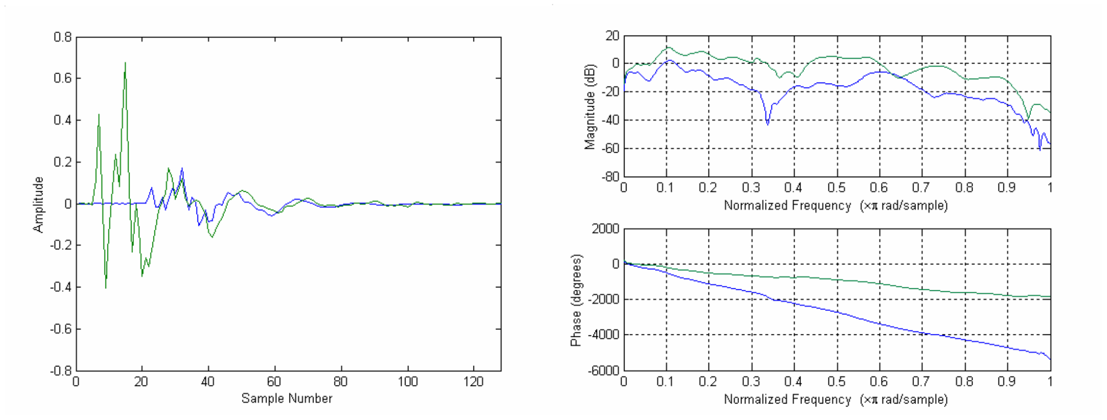


Figure 3.30 **Pair of HRTFs taken from a KEMAR dummy head from an angle of 45^0 to the left and a distance of 1 metre from the centre of the head. Green – Left Ear, Blue – Right Ear.**

The three lateralisation cues can be clearly seen in this figure. These are:

- Amplitude differences – amplitude is highest at the nearer ear.
- Time differences – farther ear signal being delayed compared to the closer ear (seen in both the time domain plot, and the phase response plot, by observing the larger [negative] gradient).
- Pinna and head filtering – as the sound has two different physical paths to travel to the ears, due to the pinna and the head, resulting in frequency dependent filtering (seen in the frequency response plot).

It is the head related transfer function that forms the basis on which binaural sound reproduction is founded, although through the use of anechoic HRTF data alone, only simple lateralisation is possible. This will be discussed shortly.

There are two ways in which to create a binaural reproduction, it can be recorded using in-ear microphones, or it can be synthesised using HRTF data. As far as the recording side of binaural sound is concerned, the theory is as simple as placing a pair of microphones into the ear of the recordist (or dummy head). The parts of the outer ear that filter the incoming sound wave are the pinna and the ear canal. If the recorded material is taken from a subject with an open ear canal (i.e. microphones placed in the ear of the subject) then the recording will possess the ear canal resonance, which lies at about 3 kHz (a 3 cm closed pipe has a fundamental resonant frequency of

2850 Hz). Then, when the listener replays the recording over headphones, the recording will be subjected to another ear canal resonance, meaning that the musical content will be perceived as having a large resonance at around 3 kHz. This, therefore, must be corrected with the use of equalisation; although the blocking of the ear canal of the recordist prior to recording is another solution (Kleiner, 1978). The actual positioning of the microphones within the outer ear of the subject has an effect on the system where the most robust positioning of the microphone is usually found to be inside the ear canal (Ryan & Furlong, 1995), (although the blocking of the ear canal is not really a desirable solution to the last problem). There are two other difficulties in using recorded binaural material and they are pinna individualism and head movements. As discussed in Chapter 2, everyone's pinnae are different, which in turn means that the complex filtering patterns that the pinnae apply to the incoming sound waves are also different. The binaural recording process means that the listener will be experiencing the sound field by listening through somebody else's ears. The results of this will be discussed later in this section.

When it comes to synthesising a binaural sound field, HRTF data is used. As the HRTF is a measure of the response of the ear due to a source, then it will suffer the same difficulties mentioned for the recorded material. However, some differences are apparent. The HRTF data used to synthesise sources is normally recorded in an anechoic chamber (Gardner and Martin, 1994) as this gives the greatest flexibility in source position synthesis as it is possible to add reverberation, but very difficult to take it away again. Also, HRTFs are usually recorded in pairs at a set distance from the centre of the head (say, one metre), but this is not necessarily the most versatile solution. As a demonstration of this, consider the situation shown in Figure 3.31.

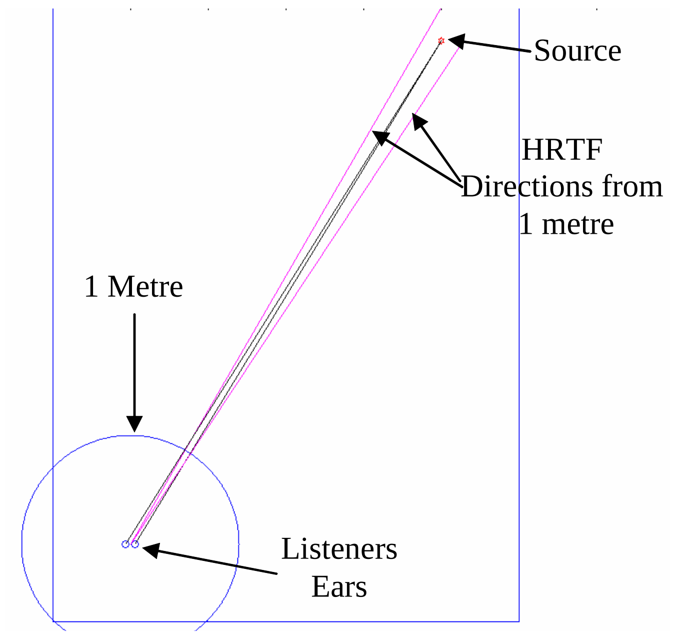


Figure 3.31 Example of a binaural synthesis problem.

If distance is to be simulated correctly, then recording and storing the HRTFs in pairs centred on the head actually complicates the situation. This is because the pair of HRTFs will have an amplitude difference, time difference, and pinna filtering that is not only due to the angle of incidence of the source, but also its distance, as discussed in Chapter 2. This means that if a source is to be synthesised at a distance that is different than the one that was measured then the point at which the source intersects the measured distance needs to be obtained. Extra delay also needs to be added to the HRTF filters, with a different value added to the left and right HRTFs. This adds extra, avoidable, calculations to the synthesis model, and is undesirable in real-time applications. To combat this problem it is far better that the HRTFs be recorded taking each ear as the centre point for the measurements as this means that the angle from source to each of the listener's ears needs to be calculated, which is simpler than the scheme detailed above (although extra delay does still need to be added for each response separately).

Once the problem of angle of incidence has been resolved (with one of the two methods suggested above) then one of the main advantages of binaural theory can come into play, and that is the simulation of distance cues. However, obtaining sources that are localisable outside of the head (i.e. not just resulting in source lateralisation) is not usually possible using anechoic simulation of the source (McKeag & McGrath, 1997). This, in some respects, is to be expected, as one of the psychological effects of being in an anechoic chamber is that sources tend to be perceived much closer than they actually are. One of the mechanisms that the brain utilises in the perception of source distance is in the direct to reverberant ratio of sounds (see Chapter 2). Sounds that are very close to the head have a very low (if any) reverberation perceived with them, so if a sound is heard in an anechoic chamber then the brain may assume that this source is close to us because of this. However, when listening to synthesised binaural sources it is unlikely that true, or even any, distance information will be perceived. This is due, mainly, to the reasons given below:

- In nearly all listening situations the ear/brain system uses small head rotations to resolve the position of a source within the cone of confusion.
- The shape and, therefore, filtering of the sound due to the pinna of the recording subject will be different than that of the listener.

A number of people (including Moller *et al.*, 1996) suggest that individualised HRTFs are needed for the accurate reproduction of binaural sound, while others suggest that head tracking is the most important aspect of the localisation process (Inanaga *et al*., 1995). However, it can be seen that neither or these are necessarily needed, and depth perception can be achieved by creating multiple, coherent auditory cues for the listener (McKeag & McGrath, 1997). Again, depending on the application, there are two methods of achieving this. Firstly, for the simulation of sources that are in a fixed position, the HRTFs can be measured in a real room, thereby recording the room's actual response to a source, in this position, at the two ears of a subject. This, when convolved with the source material, will create the illusion of a source outside the head of the listener (McKeag & McGrath, 1997). Secondly, if dynamic source movement is needed, such as in 3D gaming, and virtual reality applications, then a model of the room in which the source is placed must be realised separately from the source, and then all of the images synthesised using anechoic HRTF data. The binaural synthesis of material in this way can lead to a very convincing surround sound experience using a limited number of channels, which is probably why all 3D computer gaming cards use this form of modelling.

As mentioned in Chapter 1, it is now widely recognised that binaural headphone reproduction techniques can be used as a method of auralising multi-speaker arrays. This technique was pioneered by Lake DSP (for example, see McKeag & McGrath (1997) and McKeag & McGrath (1996) as an example of their later work), and more recently has been used by others (for example, see Leitner *et al.*, 2000 and Noisternig *et al*, 2003) as a method of simulating both discrete speaker feeds and, in the case of Ambisonics, realising an Ambisonic decoder efficiently as three or four HRTF filters (see Chapters 4 and 5 for more details on this).

Interestingly, although three of the four papers mentioned above discuss Ambisonics to binaural conversion, none use psychoacoustically optimised decoders as discussed in section 3.3.1.2. This will result in sub-optimal lateralisation parameters being reproduced at the listeners ears, as shown in the non-optimised decoders discussed in section 5.2.

3.3.5 Transaural Surround Sound

Transaural surround sound techniques were first proposed in the 1960's by Atal, Hill and Schroeder (Atal, 1966) and, although based on a relatively simple and understandable principle, were difficult to realise at this time. Transaural sound is a process by which Binaural reproduction can be realised over loudspeakers. Loudspeaker reproduction differs from headphone reproduction in that the sound from one loudspeaker reaches both ears (a fact that is the basis of Blumlein's stereo reproduction technique, see earlier in this chapter), and binaural reproduction over headphones relies on the fact that the signal from one transducer only reaches one ear, that is, there is no crosstalk between the ears of the listener. The Transaural system is easier to explain if the following problem is considered. If a pulse is emitted from one of a pair of loudspeakers, what must happen for that pulse to only appear at one ear of the listener? This situation is shown in Figure 3.32, but is simplified by taking each ear as a microphone in a free field (i.e. no filtering of the sound will be present due to the head of the listener). Each of the two speakers are equidistant from the centre of the two microphones, and subtend an angle of 60 degrees (+/- 30^0).

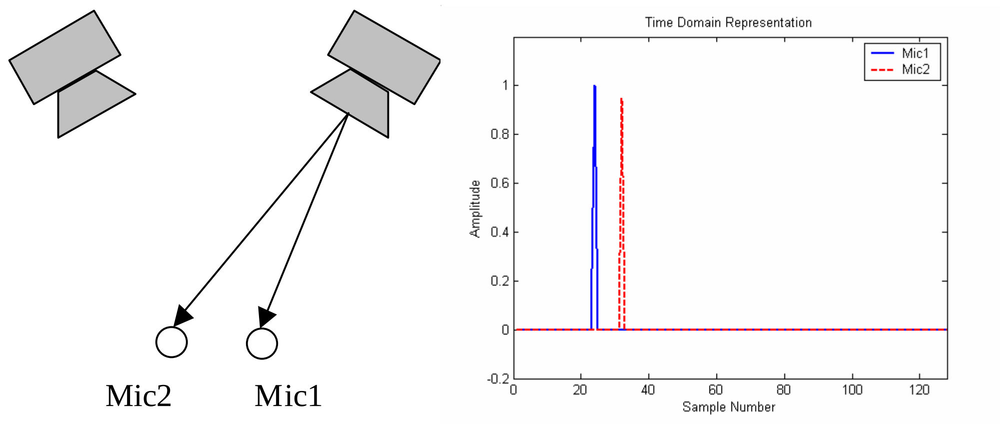


Figure 3.32 Graphical representation of the crosstalk cancellation problem.

It can be noted that Mic1 receives the pulse first, closely followed by Mic2 which receives the same pulse, except that the amplitude has attenuated and it arrives later in time due to the extra distance travelled. In order to cancel the sound arriving at Mic2, the left loudspeaker can be made to emit a sound so that the same amplitude as the signal arriving at Mic2 is achieved, but inverted (180^0 out of phase) as shown in Figure 3.33. This signal now cancels out the first sound picked up by Mic2 (see the microhpones response to each speaker’s output in Figure 3.33), but then the crosstalk produces another signal, again amplitude reduced, at Mic1. So another, amplitude reduced and phase inverted, signal is produced from the right loudspeaker to counteract the Mic1 crosstalk signal, and so on. As the amplitude of these pulses is always diminishing, a realisable and stable filter results, as shown in Figure 3.34. Also shown in Figure 3.34 is the block diagram for a typical implementation of a crosstalk cancellation system, note that this system will crosstalk cancel for both speakers, that is, the Left input signal will only appear at Mic2 and the Right input signal will only appear at Mic1. These two filters can be realised using a pair of I.I.R. filters[1]. However, this structure is not used, in practice, as the response of the listener’s head is not taken into account and so this form of crosstalk cancellation will be sub-optimal.

[1] Infinite Impulse Response filters using a feedforward/back loop and attenuating gain factors (typically).

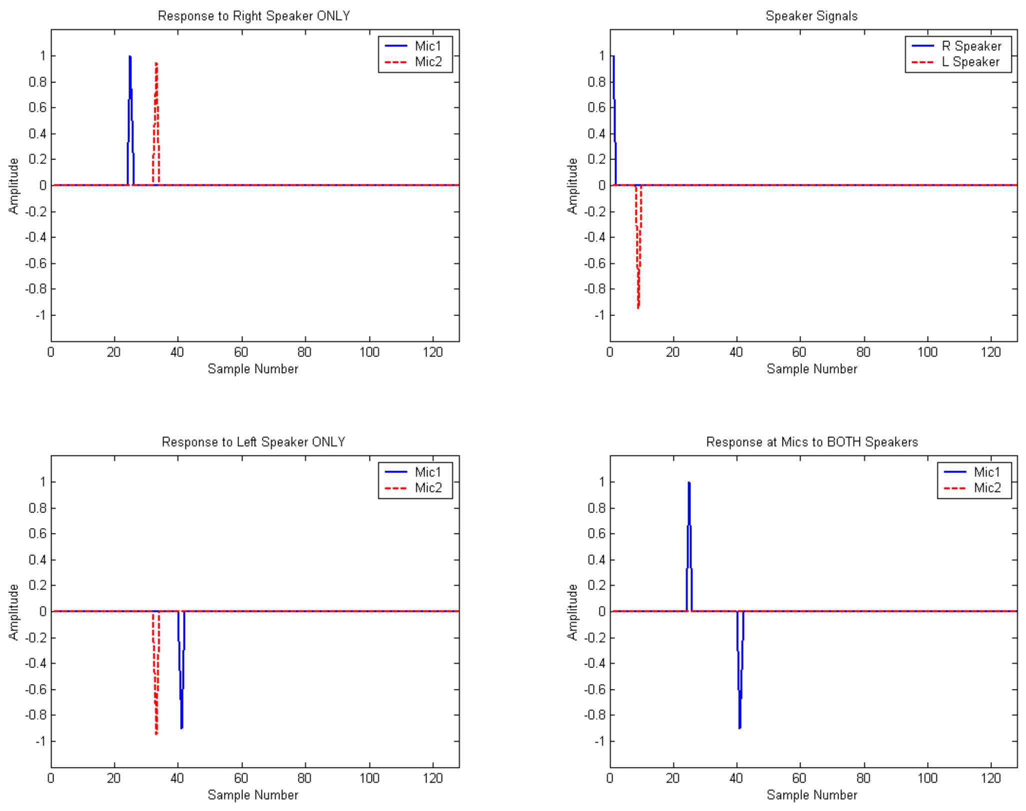


Figure 3.33 **Simulation of Figure 3.32 using the left loudspeaker to cancel the first sound arriving at Mic2.**

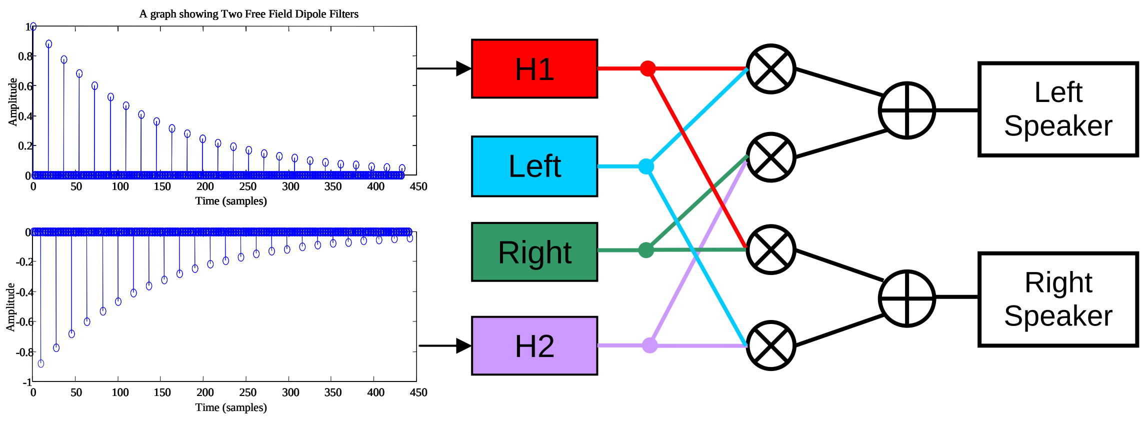


Figure 3.34 **Example of free-field crosstalk cancellation filters and an example implementation block diagram.**

Although this particular filtering model would never be used in practice, it will be used here to demonstrate the type of frequency response changes that occur due to the crosstalk cancellation filtering process. In theory, of course, the sounds heard at the two microphone positions will be as desired, but for off centre listening (and also, to some extent, listening in the sweet spot in a non-anechoic room) will have a response similar to that shown in Figure 3.35.

Although this seems slightly irrelevant for crosstalk cancellation filters designed with HRTF data, it does show some of the extreme filtering that can occur due to the system inversion process.

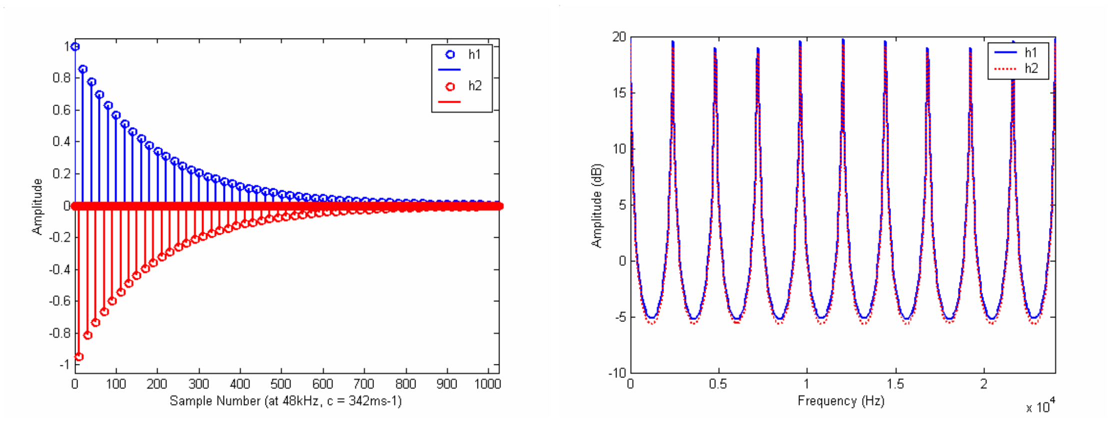


Figure 3.35 Frequency response of free field crosstalk cancellation filters

The process above, described as filter inversion is, in fact, slightly more complicated than this. Although the example above (crosstalk cancellation in the free field) is a good starting point for gaining an understanding of the processes involved in crosstalk cancellation algorithms, the equation has not yet been defined. If we again look at the problem shown in Figure 3.36, it can be seen that, for a symmetrical setup, only two transfer functions are present, c_1 – the response of the microphone to the near speaker, and c_2 – the response of the microphone to the far speaker.

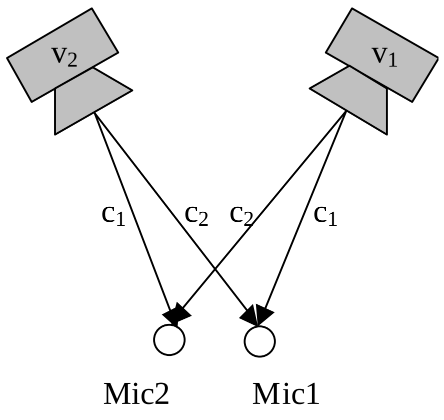


Figure 3.36 The Crosstalk cancellation problem, with responses shown.

The relationship between the signals emanating from the speakers, and what arrives at the two microphones is given in Equation (3.12).

$$\begin{bmatrix} Mic1 \\ Mic2 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 \\ c_2 & c_1 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

(3.12)

Therefore, if we wish to present to the system the signals that we wish to receive at Mic1 and Mic2, then the inverse of the transfer function matrix needs to be applied to the two signals, prior to transmission (Nelson *et al*., 1997) (which is what is happening in the system described in Figure 3.34) and is shown in Equation (3.13). The simplification to two filters, h_1 and h_2 can be made due to the crosstalk cancellation meaning that the signal at Mic2 will be forced to zero and the signal at Mic1 will be the desired signal at unity gain.

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \frac{1}{(c_1 \times c_1) - (c_2 \times c_2)} \begin{bmatrix} c_1 & -c_2 \\ -c_2 & c_1 \end{bmatrix} \cdot \begin{bmatrix} Mic1 \\ Mic2 \end{bmatrix}$$

$$\begin{matrix} v_1 = \dfrac{c_1}{c_1^{\ 2} - c_2^{\ 2}} \cdot Mic1 + \dfrac{-c_2}{c_1^{\ 2} - c_2^{\ 2}} \cdot Mic2 \\ v_2 = \dfrac{c_1}{c_1^{\ 2} - c_2^{\ 2}} \cdot Mic2 + \dfrac{-c_2}{c_1^{\ 2} - c_2^{\ 2}} \cdot Mic1 \end{matrix} \Rightarrow \begin{matrix} h_1 = \dfrac{c_1}{c_1^{\ 2} - c_2^{\ 2}} \\ h_2 = \dfrac{-c_2}{c_1^{\ 2} - c_2^{\ 2}} \end{matrix}$$

(3.13)

where: v_1 & v_2 are the speaker signals shown in Figure 3.36
c_1 & c_2 are the transfer functions from Figure 3.36.
h_1 & h_2 are the transfer functions used in Figure 3.34.

The final filters are shown in Equation (3.14) (the multiplying of $c_1^2 + c_2^2$ to both the numerator and denominator of the equation is also shown for compatibility with the frequency dependent inversion procedure) and is carried out in the frequency domain, adapted from Farina, *et al.* (2001), as inverting this system in the time domain can take a long time, even on fast computers. As an example, the calculation of the these filters in the frequency domain, using Matlab® and a filter size of 1024 points takes less than a second, however, using time domain signals coupled with the simple multiplications and divisions turning into convolutions and de-convolutions means that the same algorithm can take around half an hour to complete.

$$h_1 = c_1 \times \left(\frac{c_1^2 + c_2^2}{c_1^4 - c_2^4} \right) \qquad h_2 = -c_2 \times \left(\frac{c_1^2 + c_2^2}{c_1^4 - c_2^4} \right)$$

where: c_1 & c_2 are the transfer functions from Figure 3.36.

h_1 & h_2 are the transfer functions used in Figure 3.34.

(3.14)

It must also be noted that Equation (3.14) shows the inversion procedure for the symmetrical case (that is, the diagonals of the transfer function matrix are identical), and is not the general solution for this problem. Now that the mathematical equation has been defined, any transfer function can be used for c_1 and c_2 and a non-free field situation simulated. For example, if two speakers were spaced at +/- 30^0, as in a normal stereo triangle, then the corresponding crosstalk cancellation filters will be the same as shown in Figure 3.37.

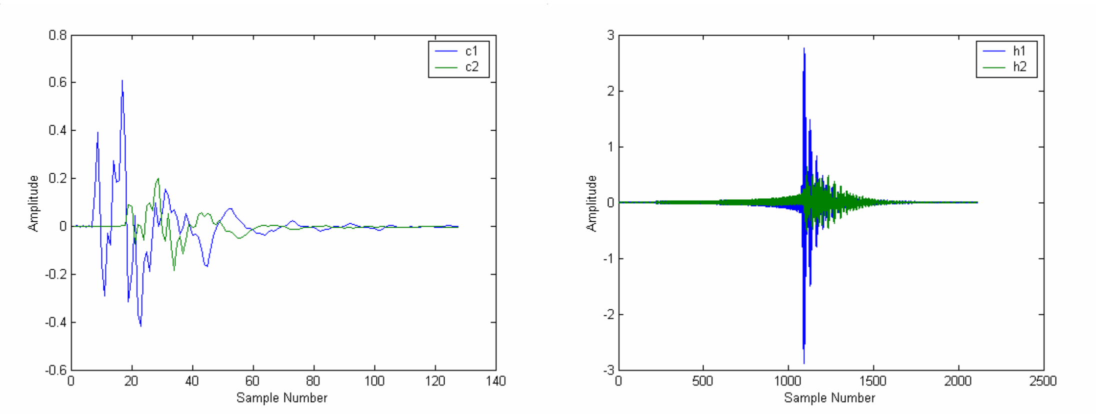


Figure 3.37 Transfer functions c_1 and c_2 for a speaker pair placed at +/- 30^0, and their corresponding crosstalk cancelling filters.

As can be seen in the right hand graph of Figure 3.37, the crosstalk cancellation filters actually have samples that are valued greater than one (which denotes potential clipping in many audio applications); however, in this case, they will not clip themselves (so long as storing these filters is not a problem). Nevertheless, when they are applied to a signal, much amplification will arise. The frequency responses of the two crosstalk cancellation filters are given in Figure 3.38.

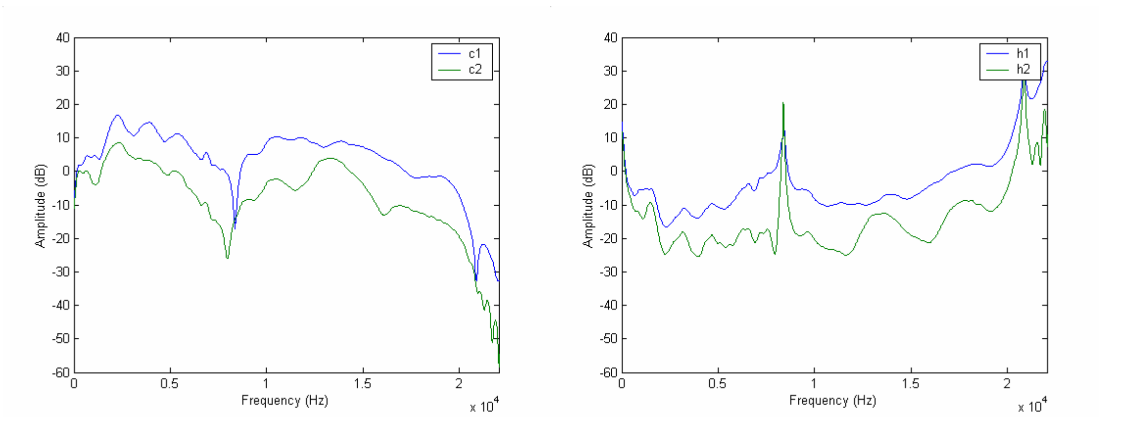


Figure 3.38 Frequency response of the two speaker to ear transfer functions (c_1 & c_2) and the two crosstalk cancellation filters (h_1 & h_2) given in figure 3.31.

It can clearly be seen that any dip in the response of the original transfer functions, c_1 and c_2, creates an almost corresponding boost in the inverse response (this sounds obvious, but h_1 and h_2 are not the inverse of c_1 and c_2 directly). In this case, the response is particularly troublesome at around 8 kHz, very low and very high frequencies. This is due partly to the ears' response (pinna etc.), the speaker response and the anti-aliasing filters in the recording of the HRTF responses respectively. To alleviate this problem a technique known as 'frequency dependent regularisation' has been developed (Kirkby *et al.*, 1999). As the peaks in the crosstalk cancellation filters are due to the filter inversion at a particular frequency, making the inversion 'sub-optimal' at these frequencies will flatten out the response at these points. The crosstalk cancellation equations using frequency dependent regularisation are given in Equation (3.15) (all transfer functions have been converted into the frequency domain).

$$h_1 = c_1 \times \left(\frac{c_1^2 + c_2^2}{c_1^4 - c_2^4 + \varepsilon} \right) \qquad h_2 = -c_2 \times \left(\frac{c_1^2 + c_2^2}{c_1^4 - c_2^4 + \varepsilon} \right)$$

(3.15)

where: c_1 & c_2 are the transfer functions from figure 3.30.
h_1 & h_2 are the transfer functions used in figure 3.28.
ε is the frequency dependant regularisation parameter (0 – full inversion, 1 – no inversion)

Figure 3.39 shows the effect on the frequency response of the two crosstalk cancellation filters using a regularisation parameter of 1 above 18 kHz. If the responses of c_1 and c_2 are observed (from Figure 3.38) it can be seen that having a regularisation parameter of 1 actually causes the resulting crosstalk cancellation filters to be the convolution of c_1 and c_2, which is why the high frequency roll-off is actually steeper in h_1 and h_2 than in c_1 and c_2.

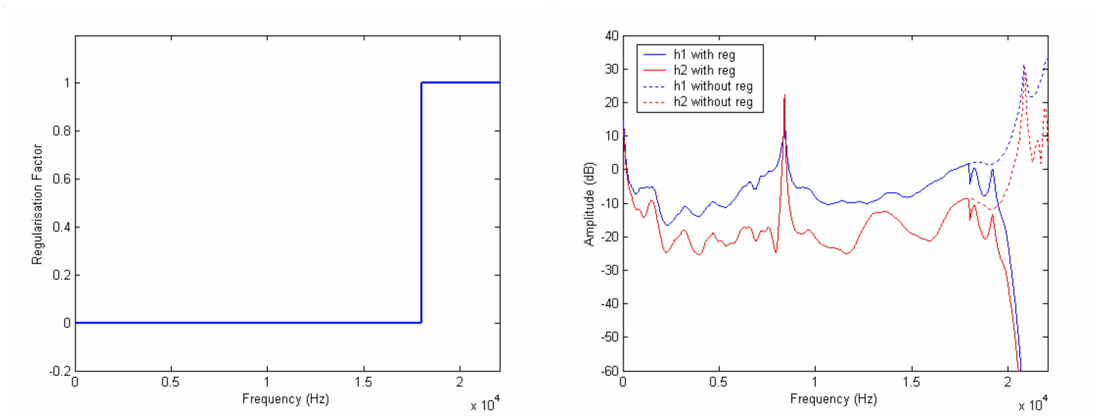


Figure 3.39 The regularisation parameter (left figure) and its effect on the frequency response of the crosstalk cancellation filters h_1 & h_2 (right figure).

Using this regularisation parameter, the response of the system can be tailored so that clipping is avoided, at the expense of sub-optimal cancellation at these frequencies. Figure 3.40 shows the crosstalk cancellation of a pulse emitted from the left speaker both with and without regularisation applied. The corresponding speaker feeds after the crosstalk cancellation filters have been applied so as to simulate the signals received by a listener.

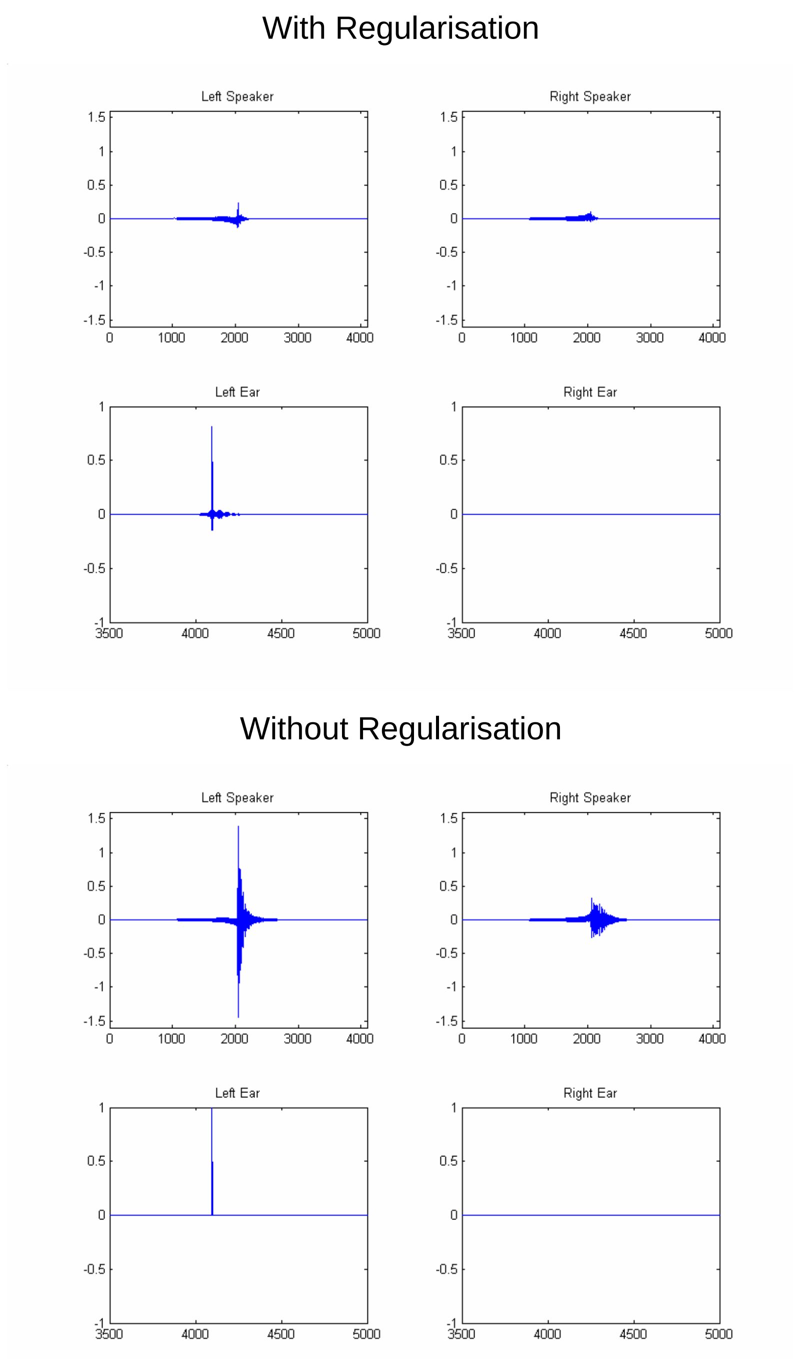


Figure 3.40 Simulation of crosstalk cancellation using a unit pulse from the left channel both with and without frequency dependent regularisation applied (as in Figure 3.39).

Assuming that any value greater than one will cause clipping of the signal then it can be clearly seen that when regularisation is applied to the crosstalk cancellation filters the system outputs much lower signals while still maintaining almost the same signal level at the ears of the listener (it must be noted that in this simulation the same HRTF data was used for both the simulation *and* the calculation of the crosstalk cancellation filters, and this will not be true in a real-life situation).

Apart from the frequency dependent regularisation parameter introduced above, much of the theory behind Transaural sound reproduction has not changed since its invention in 1962 (Atal, 1966). However, spacing the speakers as a standard stereo pair meant that the sweet spot (the area where crosstalk cancellation occurs) is small and very susceptible to errors due to head movement. To combat this, researchers at Southampton University discovered that this problem, and to a certain extent, that of excessive signal colouration, could be alleviated by moving the speakers closer together to span around 10^0. If a small speaker span is used then the area of successful crosstalk cancellation becomes larger as a line of crosstalk cancellation is created. This means that the position of the listener with respect to the distance from the loudspeakers is not so important, making the system more robust. Also, to demonstrate the signal colouration changes we will again consider the system shown in Figure 3.36. As the angular separation of the speakers becomes smaller, the more identical the transfer functions between each ear and the speakers (particularly at low frequencies) and hence, the greater the amplitude of the cancellation filters at these frequencies. This means that the angular separation of the speakers is limited by the amount of boost that must be applied to the low frequencies of the system (assuming regularisation is not used). An example of filters taking into account the HRTF of the listener is shown in Figure 3.42. This, to some extent, shows the 'swings and roundabouts' situation that can occur when dealing with the speaker placement of a Transaural system. Moving the speakers closer together makes for a more robust system, and moves much of the sound colouration into a higher frequency range, but creates a wider range of bass boost, which speakers generally find more difficult to recreate. Optimisation of this technique to alleviate some of these problems will be discussed in Chapter 5.

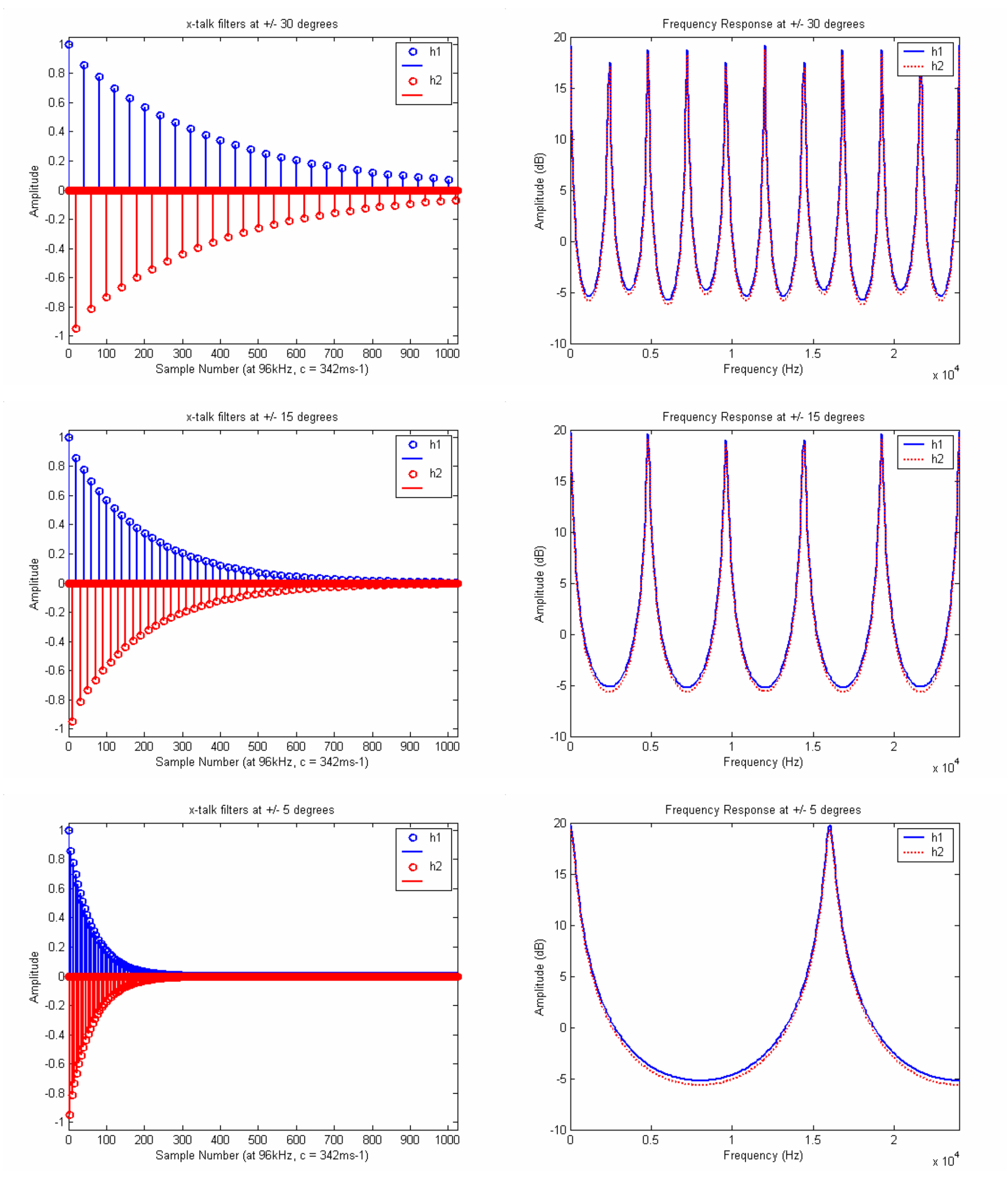


Figure 3.41 **Example of the effect of changing the angular separation of a pair of speakers used for crosstalk cancellation.**

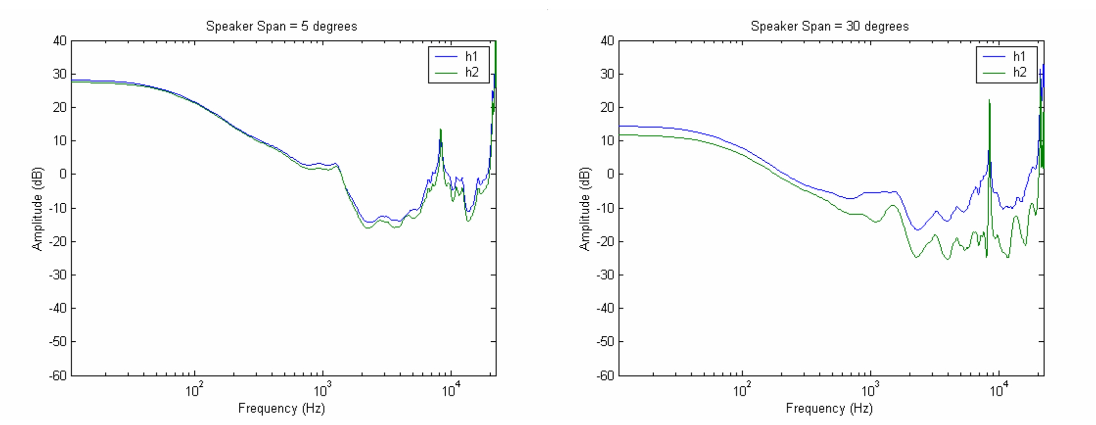


Figure 3.42 **Example of the effect of changing the angular separation of the speakers using HRTF data.**

3.3.6 Ambiophonics

The methods for recreating surround sound described above cover the current state of the art; however, there are now a number of emerging techniques that combine the virtues of more than one of these techniques in order to improve upon the usefulness of any one of these theories. Such a system is Ambiophonics (Glasgal, 2001). Ambiophonics differs from most of the systems described above as it does not attempt to be a general solution; that is, it is only designed for the listening of recorded material in a concert hall. It tries to recreate the ‘I am there’ situation. Ambiophonics is really a hybrid of binaural/transaural reproduction coupled with a more psychoacoustically correct reverb algorithm, so as to fool the ear/brain system into thinking that it is immersed within a real hall. However, this is also, to a certain extent, the remit for the Ambisonics system, so what are the main differences? The main difference is that Ambisonics uses a generic panning law so as to give equal priority (or localisation quality) to every direction, whereas Ambiophonics *always* assumes that the stage is in front of the listener and the ambience will be all around the listener. Therefore Ambisonics is a much more general surround sound solution, whereas Ambiophonics is limited in this way. However, due to this limiting factor a number of issues can be addressed. The front stage signal is recorded using (ideally) a pinna-less dummy head microphone (however, any stereo recording method will work, to some extent (Glasgal, 2001)). Also, it is a good idea to limit the amount of rear/side reflections that reach these microphones (which is normally done for stereo recordings, anyway, in order to avoid a recording that is too reverberant (Glasgal, 2003c)). Limiting the rear and side reflections picked up by this stereo recording is necessary due to the fact that these signals will be generated using convolution during the decoding stage. This stereo signal can then be replayed using a crosstalk cancellation system such as the system described in section 3.3.5. The surrounding ambience is then created and distributed using a number of speakers surrounding the listener. The main innovation here is that each speaker represents an early reflection direction. This means that, as these early reflections are being emitted from an actual source (rather than a panned position), all of the psychoacoustic cues associated with the angular directional aspect of these reflections will be absolutely correct, including the pinna cues, which are almost impossible to replicate using any other system (except Wavefield Synthesis). A typical layout for such a system is shown in Figure 3.43.

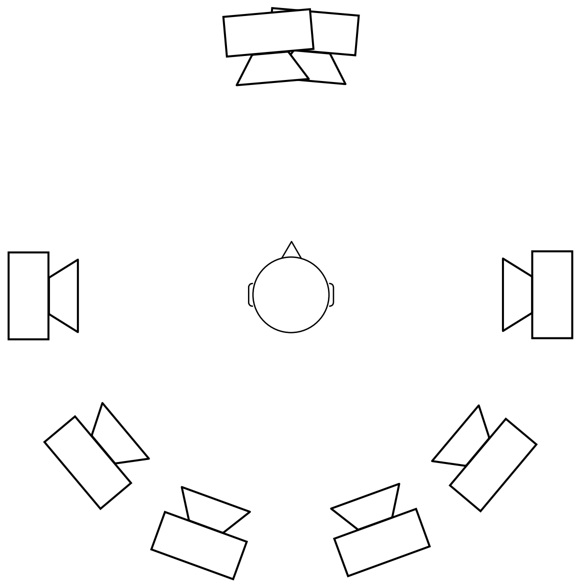


Figure 3.43 Example Ambiophonics layout.

As the crosstalk cancelled pair of speakers (typically set at +/- 5^0, which means multiple listeners sat in a line can experience the system) is reproducing the frontal hemisphere of the concert hall, fewer speakers are needed in front of the listener. The surround speakers are then fed with the stereo signal convolved with a stereo pair of impulse responses which contain no direct sound, a number of discrete reflections (one or more) and a diffuse, uncorrelated (compared to the other speakers) tail. The speakers need not be in an exact position as no exact inter-speaker imagery is to be taken into account; in fact, repositioning the speakers until the most desirable response is found is a good technique for the creation of the best sounding concert hall.

Using the Ambiophonics technique many of the cues needed for the localisation of sound and perception of a real space are met, with particular attention paid to the accuracy of the reverberation. That is not to say that the system must sound exactly like a real hall, but that the auditory cues present in the reverberation of the material are psychoacoustically very accurate and will sound like *a realistic* hall.

3.4 Summary

In this chapter, a number of techniques for the recording and reproduction of spatial sound have been investigated and discussed. It must be noted that the most popular panning algorithm, as far as the ITU 5 speaker layout is concerned, is a version of the V.B.A.P. algorithm, or pair-wise panned system. This method can work very well for frontal sources. However, at the sides of the listener, it has been shown (Gerzon, 1985) that pair-wise panning does not work correctly, with the ear/brain system finding it very difficult to decode such a system. This causes ‘holes’ in the recreated sound field, which is not too detrimental for film material, which is the medium this layout was designed for (as most material will come from the front, with occasional effects or ambience using the rear speakers). Also, it is not a particularly well defined system in that there is no agreed technique in the recording of pair-wise panned material, and recording for the ITU 5 speaker layout is quite often based upon extended Decca Tree arrangements (Theile, 2001) for a number of reasons:

- The decorrelation of low frequency components is thought to be very important in the perception of *spaciousness* in a sound field. Spacing the microphones that feed the array almost guarantees this decorrelation.
- The precedence effect can only be simulated using spaced microphone techniques. This is not to say that coincident microphone techniques

do not encode phase information (see Chapter 3), they just cannot represent time of arrival differences correctly as the microphone picks up sound from one point in space (theoretically).

However, these techniques do not lend themselves well to different speaker arrangements (that is, they are not hierarchical based formats), and now, as the media and technology for multi-channel sound reproduction is becoming more readily available, the industry is starting to realise that they do not want to rerecord/remix an album every time a new speaker layout is presented to them. For this reason this research focuses on the Ambisonics system, which is the only hierarchical system defined at this moment in time (although MPEG-4 is now being specified to address this, to some extent (MIT Media Lab, 2000)). If Ambisonics hierarchical system is used as a carrier format (in its 1st, 2nd or higher order variants) then the system can be decoded for any multi-speaker system. However, currently, a number of limitations are present using this system:

- Although Gerzon and Barton (1992) suggested a number of optimisation equations for use with irregular speaker arrangements, the equations are difficult to solve, and so no further research seems to have been published in this area giving optimal coefficients for use with the standard ITU five speaker layout.
- Although a method of converting Ambisonics and five speaker ITU surround sound to binaural reproduction has been suggested by McKeag & McGrath (1996 & 1997 respectively), no work has been carried out on the optimisation of these multi speaker systems in order to reproduce the correct psychoacoustic cues at the ears of the listener. This has been shown to be a trivial optimisation for a regular speaker array, but will rely on the work mentioned in the point above for the optimal auralisation of material if distributed on a medium carrying the standard 5.1 channels as specified by the ITU standard.
- Only a handful of software utilities for the encoding and decoding of Ambisonic material is available (McGriffy, 2002), and no psychoacoustically correct decoding software for irregular arrays exists.

These current limitations will be addressed in the following chapters of this thesis.

Chapter 4 - Development of a Hierarchical Surround Sound Format

4.1 Introduction

Although many surround sound decoding techniques are available, a number of problems are evident. For the majority of multi-speaker presentations, the material is composed specifically for a particular speaker layout, and Binaural/Transaural systems suffer from this same, inherent, problem. This does not, of course, create a problem initially, but as soon as the speaker layout becomes obsolete, or a Binaural or Transaural production needs to be replayed on a multi-speaker platform, a complete reworking of the sound piece is needed. For these reasons, this chapter will concentrate on the description of a hierarchical surround sound format, based on an amalgamation of currently available systems, in order to maximise the number of replay situations that the system is capable of satisfying. The benefits of this system are:

- The created piece will be much more portable in that, as long as a decoder is available, many different speaker layouts can be used.
- The recordings will become more future-proof as, if a speaker layout changes, just a re-decode is needed, rather than a whole remix of the piece.
- The composition/recording/monitoring of the piece will become more flexible as headphones, or just a few speakers can be used. This will result in less space being needed. This is particularly useful for on-location recordings, or small studios, where space may be limited.

4.2 Description of System

Such a system can be described diagrammatically as shown in Figure 4.1.

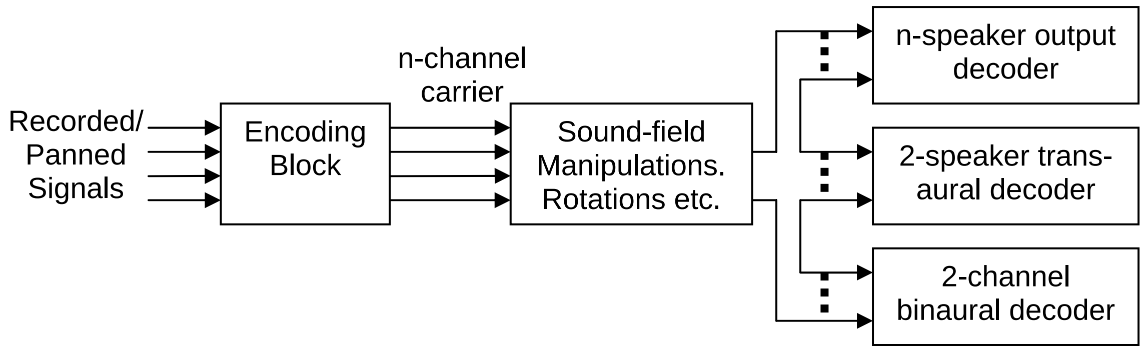


Figure 4.1 Ideal surround sound encoding/decoding scheme.

As can be seen in Figure 4.1, this ideal surround sound system should conform to the following criteria in order to maximise its flexibility and usefulness:

- A hierarchical carrier signal should be used. That is, a carrier system should be able to be understated (channels ignored, reducing localisation accuracy) or overstated (extra channels added later, increasing localisation accuracy).
- This encoded signal should be able to be manipulated *after* encoding, i.e. rotations about the x, y and z axis etc..
- The encoded signal should be able to be easily replayed over multiple listening situations including:
    - A number of different speaker arrangements, as almost no-one can place their speakers in the ITU or future speaker positions.
    - Over headphones.
    - Over a standard stereo pair (and other placement widths) of speakers.
- Efficient means of transferring from the carrier to one of the above systems.

If we take the current ‘state of the art’ surround standard as an example, and try to apply the above criteria to it, a number of shortcomings can be observed. In Dolby Digital 5.1, the carrier signal is six discrete channels, each one representing a speaker signal directly. Each speaker is assumed to be at the speaker locations specified in the ITU standard as shown in Figure 4.2.

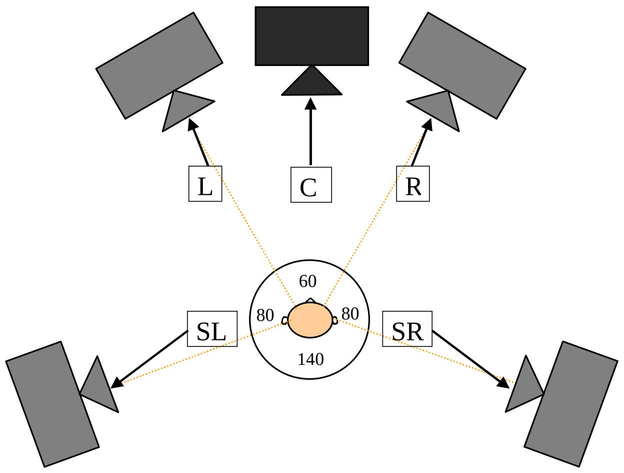


Figure 4.2 Standard speaker layout as specified in the ITU standard.

To listen to this system over headphones is not a difficult task and has been achieved by a number of companies (Mackerson *et al*., 1999; McKeag & McGrath, 1997). It is achieved by binaurally simulating speakers using HRTF data, and replaying the resulting two channels over headphones. As discussed in Chapter 3, the binaural reproduction of surround sound material needs to contain some form of psychoacoustically tangible reverb involved if a realistic, out-of-head experience is to be delivered.

When auralising 5.1 surround two approaches can be taken. The first approach assumes that the 5.1 surround system is trying to simulate an acoustic space where each speaker can be rendered using a pair of anechoic HRTFs, normally between 128 and 1024 samples in length. This approach will rely on the 5.1 decode to supply the ear/brain system with the appropriate reverberation, and is the most computationally efficient solution. However, the qualities and amount of the reverberation used on each recording may be psychoacoustically confusing and, therefore, not convincing enough to promote the out-of-head imaging possible with the binaural approach. The better approach (and the one used by Lake (McKeag & McGrath, 1997) and Stüder (Mackerson, *et al.*, 1999)) is where the speakers are simulated in a 'good' listening room, that is, each speaker will have its own reverb associated with it, on top of anything that is already recorded within the surround sound material. This can be done in one of two ways:

- Simulate the individual speakers using a pair of head related transfer functions per speaker, and then simulate the listening room using a binaural reverb algorithm (perhaps using discrete first order room reflections, again a pair of HRTFs per reflection, followed by a short, diffuse tail).

- Simulate the individual speakers and room together using a much longer pair of head related transfer functions per speaker.

The decision of which of the two approaches to use is really a question of processing power available. The difference in efficiency between the two methods can be quite high depending on the implementation used. Ideally the second method would be used, as this would provide a closer match to a real environment, and therefore maximising the performance of the binaural decode.

This method has been shown to work very well, especially when carried out with head-tracking (Makerson, *et al.*, 1999), although a good interpolation algorithm is then needed to stop the creation of clicks and pops due to the changing filter structures (in fact, the development and implementation of interpolation algorithms can be the most time consuming part of such a piece of professional audio hardware). Once the binaural version has been created it is then a relatively easy task to convert this recording for a 2 speaker, transaural reproduction by using a 2 x 2 matrix of correctly designed crosstalk cancellation filters.

However, what if the (real) speakers were not placed in the correct, ITU specified, positions in the listening room? Calculating new speaker feeds for a system that is defined by discrete channels is not necessarily an easy task (Gerzon, 1992a) when the encoding system cannot necessarily be assumed to be simple pair-wise panning.

A better technique would be to use Ambisonic B-format, or similar, to drive the system, or at least use a standard B-format decoding algorithm to derive the 6 discrete channels on a DVD and then, if desired, work out the B-format signals from these speaker feeds. Using a hierarchical carrier, such as B-format would result in the advantages given at the start of this section.

For example, if we were to take horizontal only B-format as the carrier signal then decoding this B-format carrier for the various different presentation methods can be carried out as shown in Equation (4.1) (it should be noted that this is a sub optimal decoder but this will be discussed in Chapter 5).

$$S_n(n) = \left(\sqrt{2} \times W\right) + \left(\cos(\theta) \times \cos(\phi) \times X\right) + \left(\sin(\theta) \times \cos(\phi) \times Y\right) + \left(\sin(\phi) \times Z\right) \tag{4.1}$$

where S_n is the signal sent to the nth speaker positioned at azimuth θ and elevation ϕ.

This simple decoding would produce the virtual microphone configuration shown in Figure 4.3.

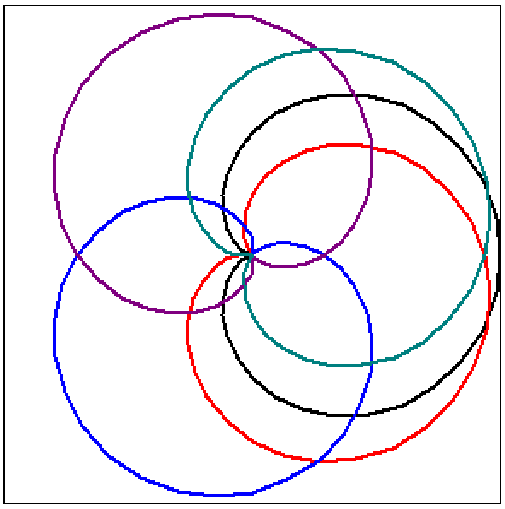


Figure 4.3 Virtual Microphone Configuration for Simple Ambisonic Decoding

4.3 B-Format to Binaural Reproduction

All multi-speaker formats can be converted to a Binaural signal, but B-Format to binaural conversion can be achieved very efficiently due to its hierarchical nature. The system can be summarized as shown in Figure 4.4.

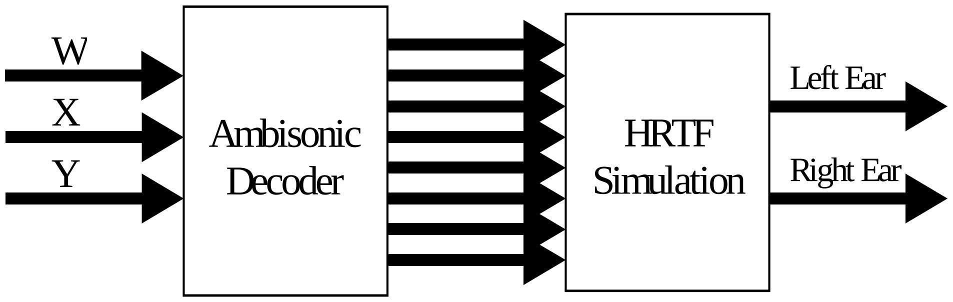


Figure 4.4 Horizontal B-Format to binaural conversion process.

As the system takes in 3 channels of audio and outputs two channels of audio, the actual Ambisonic decoding process can be contained within a pair of HRTFs representing each of W,X and Y. This means that *any* number of speakers can be simulated using just six HRTFs (three pairs). The equations describing this process for an eight speaker array are given in Equation (4.2).

$$
\begin{aligned}
W^{hrtf} &= \left(\sqrt{2}\right)\times \sum_{k=1}^{8}\left(S_k^{hrtf}\right)\\
X^{hrtf} &= \sum_{k=1}^{8}\left(\cos(\theta_k)\sin(\phi_k)\times S_k^{hrtf}\right)\\
Y^{hrtf} &= \sum_{k=1}^{8}\left(\sin(\theta_k)\sin(\phi_k)\times S_k^{hrtf}\right)\\
Z^{hrtf} &= \sum_{k=1}^{8}\left(\cos(\phi_k)\times S_k^{hrtf}\right)
\end{aligned}
$$

(4.2)

Where
θ = source azimuth
ϕ = source elevation (0 for horizontal only)
S_k^{hrtf} = Pair of HRTFs measured at speaker position, k.

The signals then required to be fed to each ear are given in Equation (4.3).

$$
\begin{aligned}
Left &= \left(W \otimes W_L^{hrtf}\right) + \left(X \otimes X_L^{hrtf}\right) + \left(Y \otimes Y_L^{hrtf}\right)\\
Right &= \left(W \otimes W_R^{hrtf}\right) + \left(X \otimes X_R^{hrtf}\right) + \left(Y \otimes Y_R^{hrtf}\right)
\end{aligned}
$$

(4.3)

Another optimisation that can be applied is that of assuming a left/right symmetrical room. For example, if the B-Format HRTFs shown in Figure 4.5 are studied it can be seen that both the left and right W HRTFs are the same, the left and right X HRTFs are the same, and the left and right Y HRTFs are the same, but phase inverted. So, in this symmetrical case only three HRTFs are needed to simulate a multi-speaker Ambisonic system with the new Left and Right ear feeds given in Equation (4.4).

$$
\begin{aligned}
Left &= \left(W \otimes W^{hrtf}\right) + \left(X \otimes X^{hrtf}\right) + \left(Y \otimes Y^{hrtf}\right)\\
Right &= \left(W \otimes W^{hrtf}\right) + \left(X \otimes X^{hrtf}\right) - \left(Y \otimes Y^{hrtf}\right)
\end{aligned}
$$

(4.4)

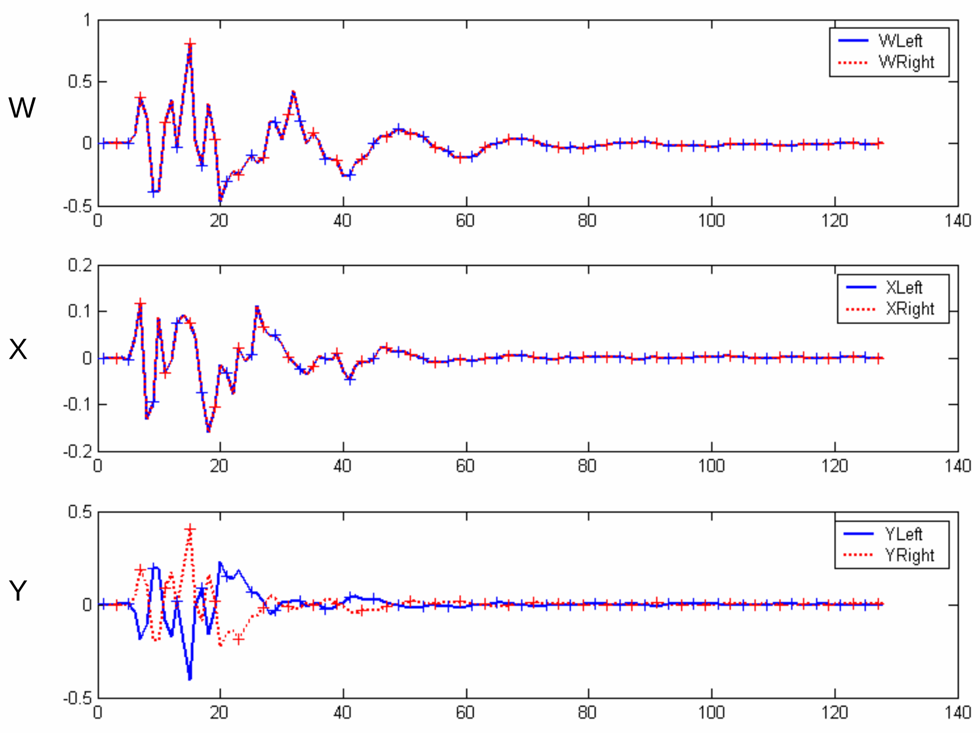


Figure 4.5 Example W, X and Y HRTFs Assuming a Symmetrical Room.

As can be seen from Equation (4.4), a symmetrical room will result in a total of three convolutions to be computed, as opposed to six for an unsymmetrical room, resulting in a 50% processing time saving (and, incidentally, this compares very favourably to the ten convolutions needed to auralise a standard five speaker when not driven by B-format).

Once the material has been ‘binauralised’, a two speaker Transaural presentation can then be created with the use of standard crosstalk cancellation filters.

For a four speaker configuration two options are available.

- If the speakers are arranged in a near square formation as shown in Figure 4.6, then the B-format signal can be decoded Ambisonically to feed these four speakers.

- If the speakers are arranged so that the speakers are placed close together (e.g. either side of a computer monitor) as shown in Figure 4.7, then a double crosstalk cancellation system would be best suited.

Both options can be utilised for most four speaker configurations, these two figures (Figure 4.6 and Figure 4.7) just show the ideal setup for each system. The system chosen would be dependant upon the listening situation and processing power available. A four speaker crosstalk cancellation has the advantage over a two speaker crosstalk cancellation system in that both front and rear hemispheres can be reproduced creating a more accurate, enveloping sound with much less noticeable front/back ambiguity, particularly if the speakers are arranged in a manner similar to Figure 4.7. This system, however, although delivering much better results than frontal crosstalk cancellation alone, is, potentially, the most processor intensive of all of the reproduction methods described in this report (although it will be shown, in Chapter 6, that this is not always the case). It can be seen from the block diagram shown in Figure 4.8 that this method of reproduction will require twice as many FIR filters than frontal crosstalk cancellation alone.

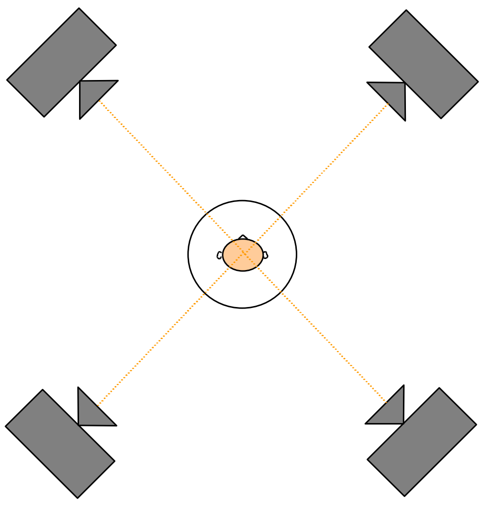


Figure 4.6 Ideal, 4-Speaker, Ambisonic Layout

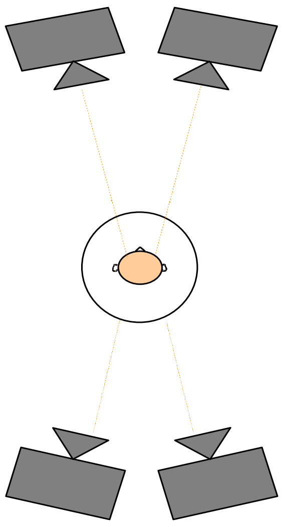


Figure 4.7 Ideal Double Crosstalk Cancellation Speaker Layout

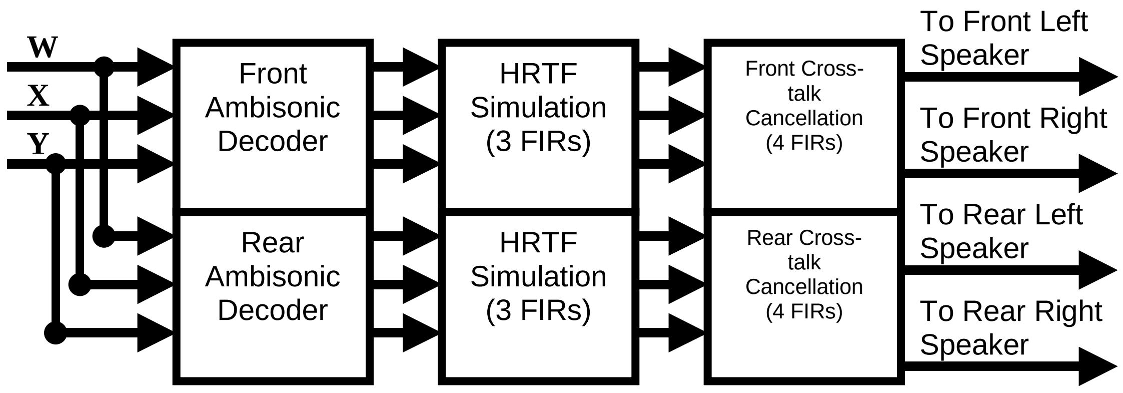


Figure 4.8 **Double Crosstalk Cancellation System**

The dual crosstalk cancelling system described by Figure 4.8, or the two speaker crosstalk cancellation system, can be made more efficient by changing the length of a number of the FIR filters when converting the B-format carrier to the Binaural signal since, as was mentioned above, non-anechoic HRTFs were utilised in order to help sound externalisation. When replaying binaural material over a crosstalk cancellation system, this is not necessary, as the sound will normally be perceived at a distance equal to the distance of the speakers. This can be observed by playing unprocessed, stereo material over a crosstalk cancelled system. In such a situation the sounds are perceived as coming from a hemisphere around the front of the listener as shown in Figure 4.9. Therefore, longer HRTFs that include some form of room response are not needed during the B-format to binaural conversion stage (as out of head localisation is already present), reducing the size of the HRTFs from over 8192 points to less than 1024 as shown in Figure 4.10, making B-format to Transaural conversion in real-time a viable option for most modern processors.

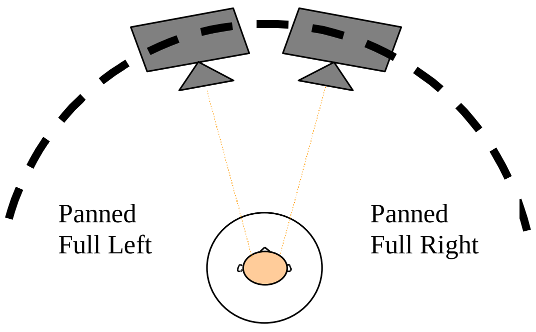


Figure 4.9 **Perceived localisation hemisphere when replaying stereophonic material over a crosstalk cancelled speaker pair.**

The four-speaker transaural system is particularly well suited to this type of speaker simulation system as standard binaural material (that is, recorded as two channels) cannot successfully be replayed on a four speaker Transaural system. It is obvious that once a binaural recording has been made, it can be played back over both the front and rear pairs of a four speaker, crosstalk cancellation system, but it is then up to the listener's ear/brain system to decide which sounds are coming from the front or the back as the same signal must be replayed from both crosstalk cancelling pairs, unless a 'four ear' dummy head recording is used. This gives many conflicting cues due to the imperfect manner in which Transaural systems crosstalk cancellation occurs. However, using the system mentioned above, total separation of the front and rear hemisphere's audio is possible resulting in a much less ambiguous listening situation, where the best possible use of each pair of speakers can be realised.

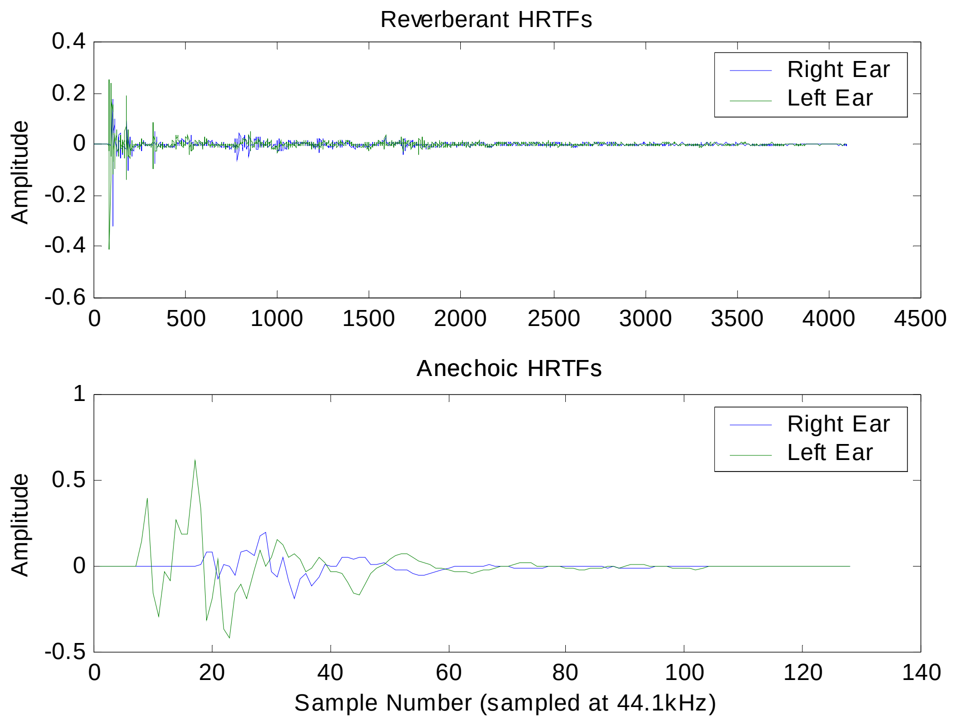


Figure 4.10 **Example of Anechoic and non-Anechoic HRTFs at a position of 30^0 from the listener.**

All of the above equations assume that the carrier signal for this hierarchical system is first order B-format. However, as DVD players already expect to see six channels, this is not the best use of the already available outputs. Ideally, a 2nd Order Ambisonic carrier would be used.

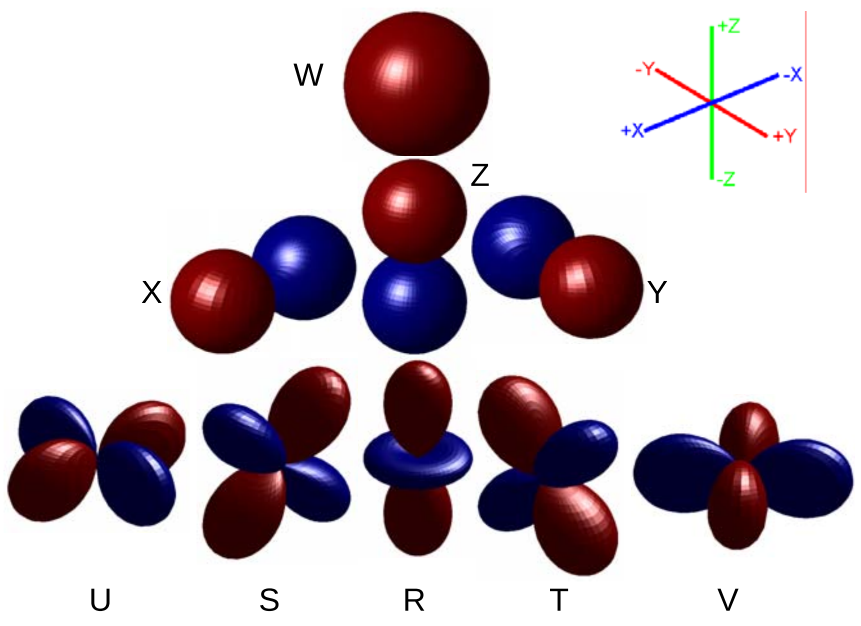


Figure 4.11 Spherical Harmonics up to the 2nd Order.

Second order Ambisonics, as mentioned in Chapter 3, would consist of nine channels to fully represent the three dimensional sound field: the four channels of 1st Order B-format, plus another five channels representing the sound field's 2nd Order components (as shown in Figure 4.11). The use of these extra harmonics increases the directionality of the virtual pickup patterns that can be constructed by combining the signals in various proportions. Figure 4.12 shows the difference between a 1st and 2nd order virtual polar pattern. At the present time, the ITU standard specifies 6 full bandwidth audio channels (note that even the .1 channel is actually stored as full bandwidth on the DVD Audio and Super Audio CD disks), and so a standard to be adopted that uses to a maximum of six channels would be preferable.

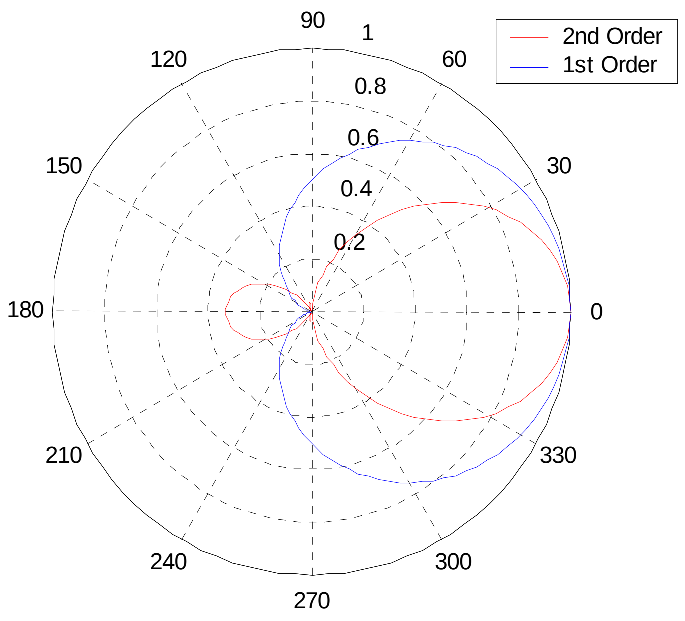


Figure 4.12 2D polar graph showing an example of a 1st and 2nd order virtual pickup pattern (0^0 point source decoded to a 360 speaker array).

The most logical way of achieving this is by specifying the horizontal plane to 2nd order resolution and the vertical plane to 1st order, resulting in a total of 6 channels (W, X, Y, Z, U & V) where most people with a horizontal five speaker, or less, system would utilise channels W, X and Y. Systems with height capability would use the Z channel and users with a higher number of speakers on the horizontal plane would also use the U and V signals. This six channel system has the advantage that the best possible resolution can be achieved on the horizontal plane (i.e. 2nd order). While the equations for tumbling and tilting the sound field will now only be fully utilisable when using the first order signals, rotating will still function, as only the horizontal Ambisonic channels are altered.

4.4 Conclusions

With the use of three existing systems, a system has been proposed that overcomes the weaknesses of the individual systems in isolation. This system has the benefit of future-proofing in terms of speaker layout and can be decoded to headphones or two or more speakers whilst still retaining spatial information. Basic algorithms for the conversion processes have been described and will be analysed, discussed and optimised in Chapter 5.

Chapter 5 - Surround Sound Optimisation Techniques

5.1 Introduction

In this chapter a number of optimisation methods will be discussed and demonstrated so as to maximise the performance of the hierarchical system discussed in Chapter 4. A large part of this research was based upon the use of HRTF data collected by Gardner & Martin (1994) which was used in order to help quantify and optimise the various decoding stages that are present in the proposed hierarchical system. The research was carried out in a number of stages which also corresponds to the layout of this chapter, as detailed below:

- Investigation into the use of HRTF data in the analysis of multi-channel sound reproduction algorithms.
- Optimisation of the Ambisonics decoding signal processing techniques.
- Optimisation of the binaural decoding signal processing techniques.
- Optimisation of the Transaural decoding signal processing techniques.

To this end, the first part of this investigation, documented in section 5.2, was to carry out a listening test, using the Multi-Channel Research Lab designed and installed as part of this research (Schillebeeckx *et al.*, 2001), to try and measure the potential strengths and weaknesses of the proposed HRTF analysis technique. As the listening tests were executed before the research into the Ambisonic optimisation methods were carried out, sub-optimal Ambisonic decodes were used in these tests. Also, as work had only just begun on the Transaural processing techniques, and due to the extremely sub-optimal performance of the designed filters, this work is not included.

Section 5.3 represents the bulk of this chapter, and concentrates on the optimisation of the Ambisonics system, as this is the base system from which the binaural and transaural representations will be derived from. Although it would be preferable to always derive the binaural/transaural feeds from the original B-format (or higher order) carrier, due to the standards used in current consumer and professional audio equipment (i.e. 5, 6 or 7 channel presentation for a 5, 6 or 7 speaker, irregular array) it is necessary to realise optimised Ambisonic decoders for irregular arrays not only to maximise the performance of the speaker decode, but to also make sure that the correct psychoacoustic cues are presented to a listener after this irregular decode is converted to a binaural or transaural reproduction.

The original optimisation, as proposed by Gerzon & Barton (1992) is an extension of the original Ambisonic energy and velocity vector theory used to optimise regular decoders (Gerzon, 1977a) but with the added suggestion of using one decoder for low frequencies and another for high frequencies. However, although Gerzon and Barton (1992) did solve these equations for a number of irregular speaker arrays, none of the arrays were similar to the ITU standard array that was finally proposed. No decoders optimised in this way have ever been produced for the ITU standard speaker array since that time, as was evident in the recent Project Verdi Listening Tests (Multi Media Projekt Verdi, 2002). The equations, a set of non-linear simultaneous equations, were difficult to solve, and only got more difficult when more speakers were added (Gerzon & Barton, 1992). For this reason one of the main aims of this work was to devise a system so that Ambisonic decoders for irregular speaker arrays could be easily designed via some form of automated system. After this was successfully implemented, the analysis method suggested in earlier work (see Wiggins *et al*, 2001) was used as the basis of new optimisation criterion for irregular Ambisonic decoders. As no method of differentiation between decoders optimised using the energy/velocity vector model currently exists (there are multiple solutions), this new method could then be used as a method to differentiate between already designed velocity/energy vector decoders.

Section 5.4 documents the work carried out on both Binaural and Transaural reproduction techniques. The work on binaural reproduction is used as an introduction to inverse filtering techniques, which are then applied to the Transaural reproduction system in order to improve its performance using the freely available HRTF data from MIT Media Lab (Gardner & Martin, 1994).

5.2 The Analysis of Multi-channel Sound Reproduction Algorithms Using HRTF Data

5.2.1 The Analysis of Surround Sound Systems

Much research has been carried out into the performance of multi-channel sound reproduction algorithms, both subjectively and objectively. Much of the quantitative data available on the subject has been calculated by mathematically simulating acoustical waves emitting from a number of fixed sources (speakers) (Bamford, 1995) or using mathematical functions that give an indication of the signals reaching the listener (Gerzon, 1992b). The resulting sound field can then be observed. In this section of Chapter 5, a new method of analysis will be described using Head Related Transfer Functions as a reference for the localisation cues needed to successfully localise a sound in space. This method will then be compared to results obtained from a listening test carried out at the University of Derby's Multi-Channel Sound Research Laboratory.

5.2.2 Analysis Using HRTF Data

The underlying theory behind this method of analysis is that of simple comparison. If a real source travels through 360^0 around the head (horizontally) and the sound pressure level at both ears is recorded, then the three widely accepted psychoacoustic localisation cues (Gulick *et al.*, 1989; Rossing, 1990) can be observed. These consist of the time difference between the sounds arriving at each ear due to different path lengths, the level difference between the sounds arriving at each ear due to different path lengths and body shadowing/pinna filtering, a combination of complex level and time differences due to the listeners own pinna and body. The most accurate way to analyse and/or reproduce these cues is with the use of Head Related Transfer Functions.

For the purpose of this analysis technique, the binaural synthesis of virtual sound sources is taken as the reference system, as the impulse responses used for this system are of real sources in real locations. The HRTF set used does not necessarily need to be optimal for all listeners (which can be an issue for binaural listening) so long as all of the various localisation cues can be easily identified. This is the case because this form of analysis compares the difference between real and virtual sources and as all systems will be synthesised using the same set of HRTFs, their performance when compared to another set of HRTFs should not be of great importance.

Once the system has been synthesised using HRTFs, impulse responses can be calculated for virtual sources from any angle so long as the panning laws for the system to be tested are known. Once these impulse responses have been created the three parameters used for localisation can be viewed and compared, with estimations made as to how well a particular system is able to produce accurate virtual images.

Advantages of this technique include:

- All forms of multi-channel sound can potentially be analysed meaningfully using this technique.
- Direct comparisons can be made between very different multi-channel systems as long as the HRTFs used to analyse the systems are the same.
- Systems can be auditioned over headphones.

5.2.3 Listening Tests

In order to have a set of results to use as a comparison for this form of analysis, a listening test was carried out. The listening test comprised of a set of ten tests for five different forms of surround sound:

- 1st Order Ambisonics over 8 speakers (horizontal only)
- 2nd Order Ambisonics over 8 speakers (horizontal only)
- 1st Order Ambisonics over a standard 5 speaker layout.
- Amplitude panned over a standard 5 speaker layout.
- Transaural reproduction using two speakers at +/- 5^0.

The tests were carried out in the University of Derby's Multi Channel Sound Research Laboratory with the speakers arranged as shown in Figure 5.1.

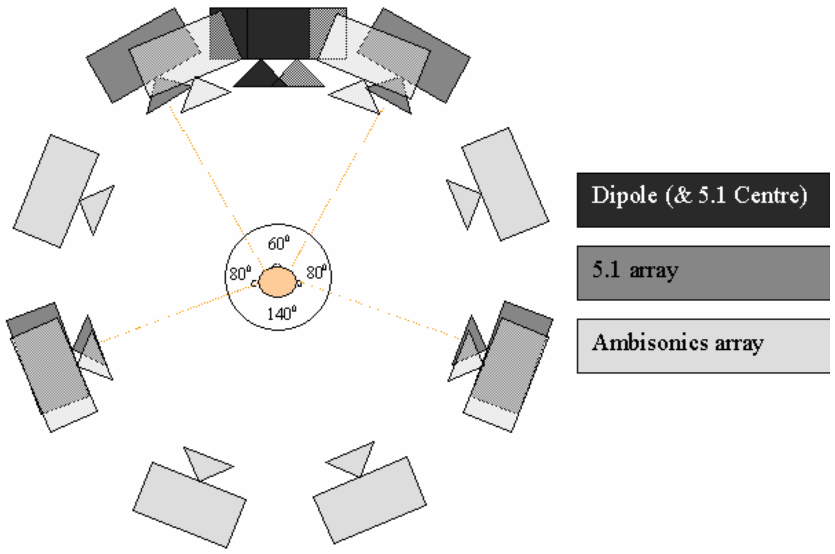


Figure 5.1 Speaker Arrangement of Multi-channel Sound Research Lab.

The listing room has been acoustically treated and a measurement of the ambient noise in the room gave around 43 dBA in most 1/3-octave bands, with a peak at 100 Hz of 52.1 dBA and a small peak at 8 kHz of 44.4 dBA. The RT60 of the room is 0.42 seconds on average, but is shown in 1/3-octave bands in Figure 5.17.

Using a PC and a multi-channel soundcard (Soundscape Mixtreme) all of the speakers could be accessed simultaneously (Schillebeeckx *et al.*, 2001), if needed, and so tests on all of the systems could be carried out in a single session without any pauses or equipment changes/repatching.

A flexible framework was devised using Matlab and Simulink (The Mathworks, 2003) so that listening test variables could be changed with minimal effort, with the added bonus that the framework would be reusable for future tests. A Simulink 'template' file was created for each of the five systems that could take variables from the Matlab workspace, such as input signal, overall gain and panning angle, as shown in Figure 5.2. Then a GUI was created where all of the variables could be entered and the individual tests run. A screen shot of the final GUI is shown in Figure 5.3.

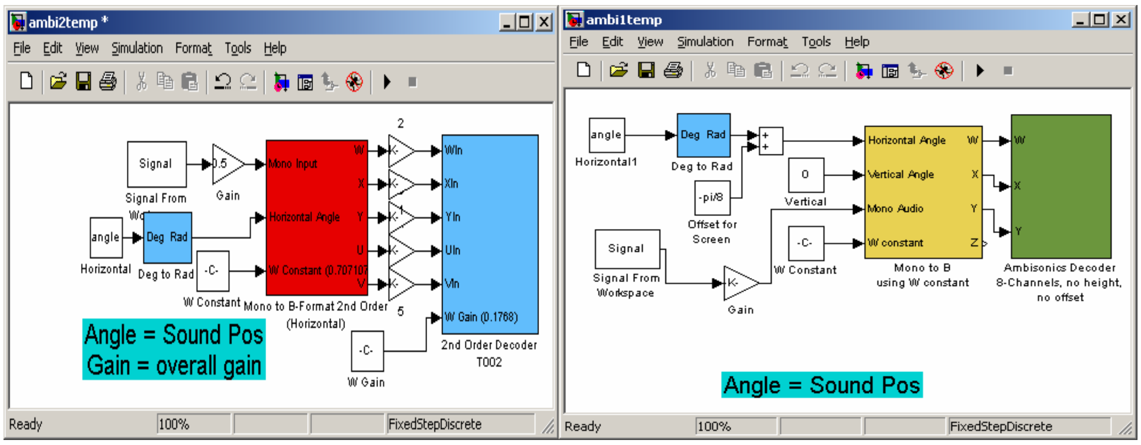
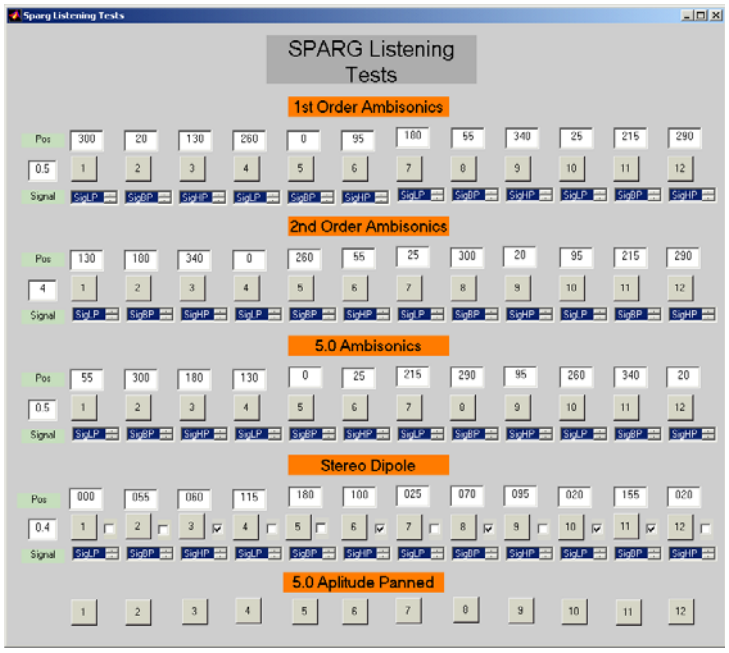


Figure 5.2 **Screen shot of two Simulink models used in the listening tests.**



Sparg Listening Tests

SPARG Listening Tests

1st Order Ambisonics

Pos	300	20	130	260	0	95	180	55	340	25	215	290
0.5	1	2	3	4	5	6	7	8	9	10	11	12
Signal	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP

2nd Order Ambisonics

Pos	130	180	340	0	260	55	25	300	20	95	215	290
4	1	2	3	4	5	6	7	8	9	10	11	12
Signal	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP

5.0 Ambisonics

Pos	55	300	180	130	0	25	215	290	95	260	340	20
0.5	1	2	3	4	5	6	7	8	9	10	11	12
Signal	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP

Stereo Dipole

Pos	000	055	060	115	180	100	025	070	095	020	155	020
0.4	1 ☐	2 ☐	3 ☑	4 ☐	5 ☐	6 ☑	7 ☐	8 ☑	9 ☐	10 ☑	11 ☑	12 ☐
Signal	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP	SigLP	SigBP	SigHP

5.0 Aplitude Panned

1	2	3	4	5	6	7	8	9	10	11	12

Figure 5.3 **Screen shot of listening test GUI.**

The overall gain parameter was included so each of the different systems could be configured to have a similar subjective gain, with the angle of the virtual source specified in degrees. The only exception to this was the 5.0 Amplitude panned system where the speaker feeds were calculated off line using the Mixtreme soundcards internal mixing feature. The extra parameter (tick box) in the Stereo Dipole (transaural) section was used to indicate which side of the listener the virtual source would be placed as the HRTF set used (Gardner & Martin, 1994) only had impulse responses for the right hemisphere and must be reversed in order to simulate sounds originating from the left (indicated by a tick).

After consulting papers documenting listening tests of various multi-channel sound systems, it was found that noise (band-limited and wide-band) was often used as a testing source (see Moller *et al.*, 1999, Kahana *et al.*, 1997, Nielsen, 1991 Orduna *et al.*, 1995 and Zacharov *et al*, 1999, as typical examples). The noise signals used in this test were band limited and pulsed, three pulses per signal, with each pulse lasting two seconds with one second of silence between each pulse. The pulsed noise was chosen as it was more easily localised in the listening room when compared to steady state noise. Each signal was band limited according to one of the three localisation frequency ranges taken from two texts (Gulick *et al.*, 1989; Rossing, 1990). These frequencies are not to be taken as absolutes, just a starting point for this line of research. A plot of the frequency ranges for each of the three signals is shown in Figure 5.4.

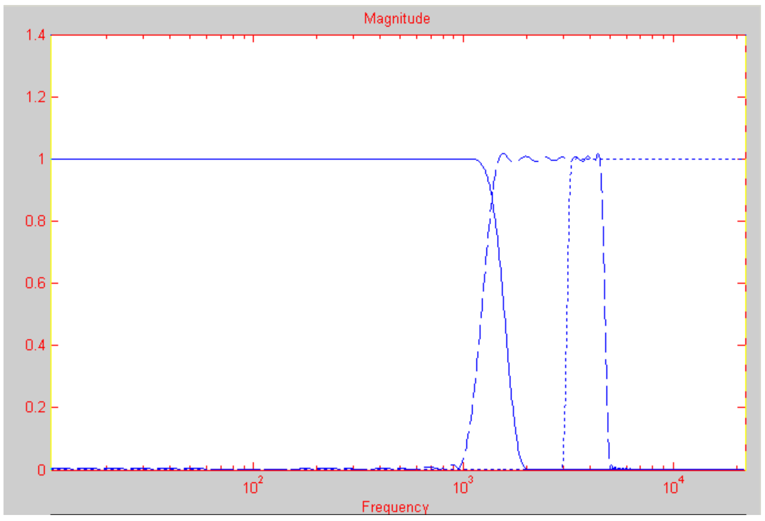


Figure 5.4 Filters used for listening test signals.

Twenty eight test subjects were used, most of whom had never taken part in a listening test before. The test subjects were all enrolled on the 3rd year of the University's Music Technology and Audio System Design course, and so knew the theory behind some surround sound systems, but had little or no listening experience of the systems at this point. Each listener was asked to try to move their head as little as possible while listening (i.e. don't face the source), and to indicate the direction of the source by writing the angle, in degrees, on an answer paper provided. It must be noted that the head of the listeners were not fixed and so small head movements would have been available to the listeners as a potential localisation cue (as it would be when listening anyway). Listeners could ask to hear a signal again if they needed to, and the operator only started the next signal after an answer had been recorded. The listeners were given a sheet of paper to help them with angle locations with all of the speaker positions marked in a similar fashion to Figure 5.5 (although the sheet presented to the test subjects was labelled in 5^0 intervals with a tick size of 1^0, not 15^0 intervals with a tick size of 3^0 as shown in Figure 5.5).

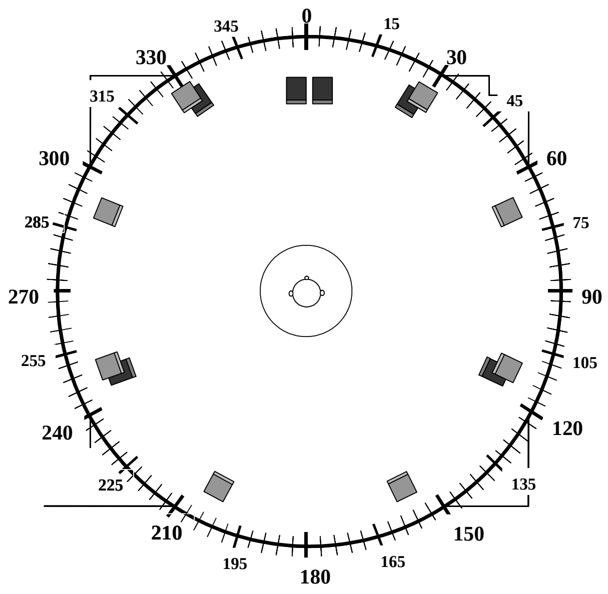


Figure 5.5 **Figure indicating the layout of the listening room given to the testees as a guide to estimating source position.**

5.2.4 HRTF Simulation

As described in section 5.1 three of the five systems will be analysed using the HRTF method described above:

- 1st Order Ambisonics
- 2nd Order Ambisonics
- 1st Order Ambisonics over 5 speakers.

The listening test results for the amplitude panned 5 speaker system are also included.

The set of HRTFs used for this analysis were the MIT media lab set of HRTFs, specifically the compact set (Gardner & Martin, 1994). As mentioned earlier, it is not necessarily important that these are not the best HRTF set available, just that all of the localisation cues are easily identifiable.

All systems can be simulated binaurally but Ambisonics is a slightly special case as it is a matrixed system comprising the steps shown in Figure 5.6.

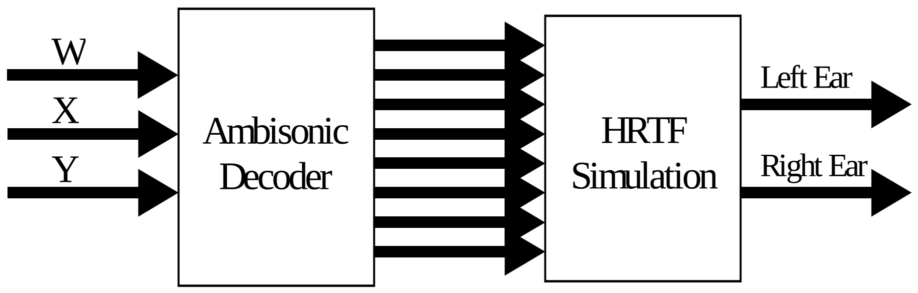


Figure 5.6 The Ambisonic to binaural conversion process.

Because the system takes in three channels which are decoded to eight speaker feeds, which are then decoded again to two channels, the intermediate decoding to eight speakers can be incorporated into the HRTFs calculated for W, X and Y meaning that only six individual HRTFs are needed for any speaker arrangement, Equation (5.1). If the head is assumed to be symmetrical (which it is in the MIT set of compact HRTFs) then even fewer HRTFs are needed as Wleft and Wright will be the same (Ambisonics omni-directional component), Xleft and Xright will be the same (Ambisonics front/back component) and Yleft will be phase inverted with respect to Yright. This means a complete 1st order Ambisonic system comprising any number of speakers can be simulated using just three HRTF filters, as shown in equation (5.1).

$$W^{hrtf} = \left(\sqrt{2}\right)\times\sum_{k=1}^{8}\left(S_k^{hrtf}\right)$$

$$X^{hrtf} = \sum_{k=1}^{8}\left(\cos(\theta_k)\sin(\phi_k)\times S_k^{hrtf}\right)$$

$$Y^{hrtf} = \sum_{k=1}^{8}\left(\sin(\theta_k)\sin(\phi_k)\times S_k^{hrtf}\right)$$

Where
θ = source azimuth
ϕ = source elevation (0 for horizontal only)
S_k^{hrtf} = Pair of Speakers positional HRTFs.

(5.1)

Once the HRTFs for W, X and Y are known, a virtual source can be simulated by using the first order Ambisonics encoding equations shown in Equation (5.2), (Malham, 1998).

$$W = \left(1/\sqrt{2}\right) \times x(n)$$
$$X = \cos(\theta) \times \sin(\phi) \times x(n)$$
$$Y = \sin(\theta) \times \sin(\phi) \times x(n)$$

Where x(n) is the signal to be placed in virtual space.

(5.2)

Using two sets of the W, X and Y HRTFs (one for eight and one for five speaker 1st order Ambisonics) and one set of W, X, Y, U and V (Bamford, 1995; Furse, n.d.) for the 2nd order Ambisonics, sources were simulated from 0^0 to 360^0 in 5^0 intervals. The 5^0 interval was dictated by the HRTF set used since, although the speaker systems could now be simulated for any source angle, the real sources (used for comparison) could only be simulated at 5^0 intervals (without the need for interpolation). An example pair of HRTFs for a real and a virtual source are shown in Figure 5.7.

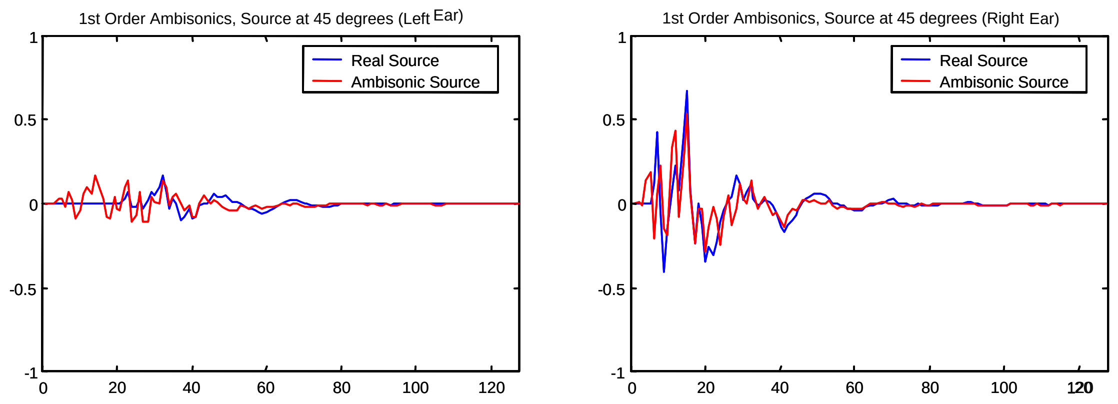


Figure 5.7 **Example left and right HRTFs for a real and virtual source (1st Order Ambisonics) at 45^0 clockwise from centre front.**

5.2.5 Impulse Response Analysis

As mentioned in Section 5.2.2, three localisation cues were analysed, interaural level difference, interaural time difference, and pinna filtering effects. The impulse responses contain all three of these cues together meaning that although a clear filter delay and level difference can be seen by inspection; the pinna filtering will make both the time and level differences frequency dependant. These three cues were extracted from the HRTF data using the following methods:

- Interaural Amplitude Difference – Mean amplitude difference between the two ears, taken from an FFT of the impulse responses.
- Interaural Time Difference – Mean time difference between the two ears, taken from the group delay of the impulse responses.
- Pinna filtering – Actual time and amplitude values, taken from the group delay and an FFT of the impulse responses.

Once the various psychoacoustic cues had been separated, comparisons were made between the cues present in a multi-speaker decode compared with the cues of an actual source (i.e. the individual HRTFs) and estimations of where the sounds may appear to come from can be made using each of the localisation parameters in turn. As the analysis is carried out in the frequency domain, band limiting the results (to coincide with the source material used in the listening tests) is simply the case of ignoring any data that is outside the range to be tested.

As an example, Figure 5.8 shows the low, mid and high frequency results for real sources and the three Ambisonic systems for averaged time and amplitude differences between the ears.

These graphs show a number of interesting points about the various Ambisonic systems. Firstly, the 2nd order system actually has a greater amplitude difference between the ears at low frequencies when compared to a real source, and this is also the frequency range where all of the systems seem to correlate best with real sources. However, the ear tends to use amplitude cues more in the mid frequency range, and another unexpected result was also discovered here. It seems that the 1st order, five speaker system actually outperforms the 1st order, eight speaker system at mid frequencies, and seems to be equally as good as the eight speaker, second order system. This is not evident in the listening tests, but if the average time difference graphs are observed it can be seen that the five speaker system has a number of major errors around the 90^0 and 270^0 source positions and shows the 2nd order system to hold the best correlation. The time difference plots all show that the five speaker system still outperforms the 1st order, eight speaker system, apart from the major disparities, mentioned above, at low frequencies. It can be seen from the listening test results (Figure 5.12) that the five speaker system does seem to be at least as good as the eight speaker system over all three of the frequency ranges, which was not expected. The mid and high frequency range graphs are a little too complicated to analyse by inspection and so will be considered later in this chapter using a different technique. It must also be noted that, due to the frequency ranges originally chosen, interaural level differences at low frequencies are comparable to the interaural level differences at mid frequencies. Had a lower cut off frequency been chosen (as shown later in this Chapter) this would not have been the case and this suggests that the original frequency ranges were not ideal.

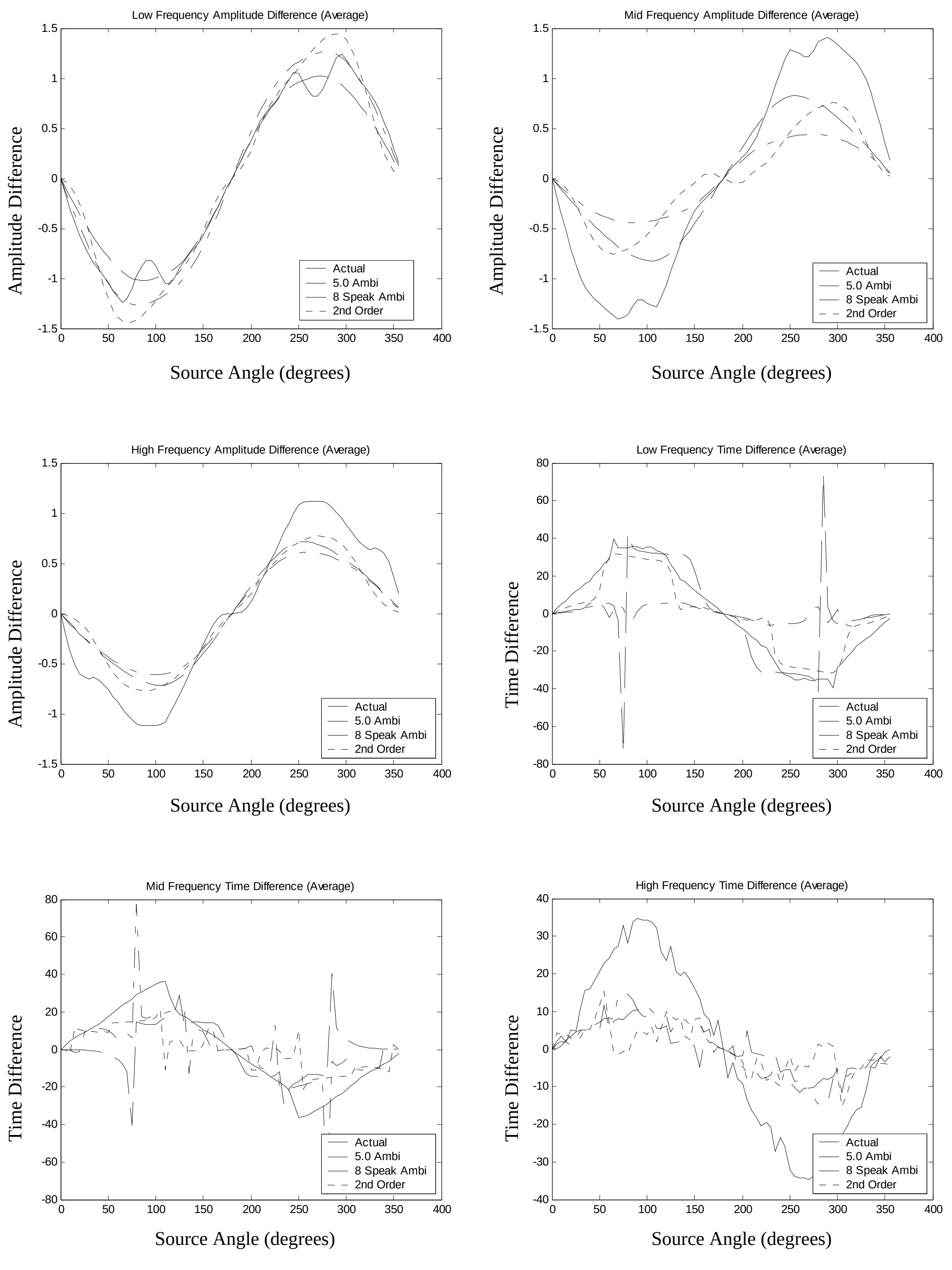


Figure 5.8 **The average amplitude and time differences between the ears for low, mid and high frequency ranges.**

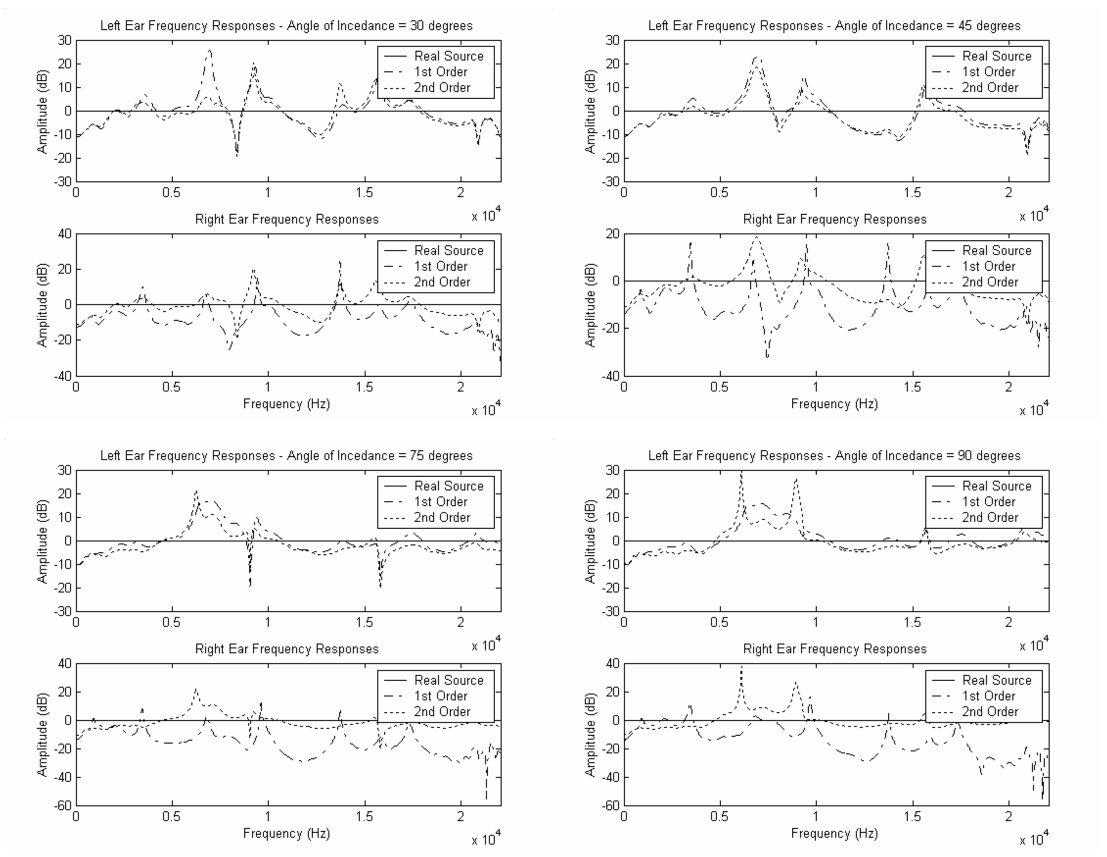


Figure 5.9 The difference in pinna amplitude filtering of a real source and 1st and 2nd order Ambisonics (eight speaker) when compared to a real source.

One attribute that has not really been touched on yet, when discussing multi-speaker systems, which is one of the major consequences of the phantom imaging scenario, is pinna cue errors. When an image is created with more than one speaker, although it is possible to create a correct level and phase difference at the ears of a listener, for a panned source, it will be far more difficult to create correct pinna cues due to the direction dependant filtering that the pinnae apply to *real* sound sources. Instead, the pinna cues from the speakers creating the phantom image will be summed and weighted dependant on the speakers' contributions. As everyone's pinnae are different, it is impossible to correct for this in a generic way (and even from an individual's response point of view, only one listener orientation could be corrected for, i.e., facing straight ahead). The pinna filtering can be clearly seen in the simulation, but is a more complex attribute to analyse directly, although it has been useful to look at for a number of reasons. For example, if the non-averaged amplitude or group delay parameters are looked at over the full 360^0 (the non-averaged amplitude responses are shown in Figure 5.9) it can be seen that they both change radically due to virtual source position (as does a source in reality). However, the virtual sources change differently when compared to real sources. This change will also occur if the head is rotated (in the same way as a source moving for a regular rig, or a slightly more complex way for an irregular five speaker set-up) and this could be part of the 'phasiness' parameter that Gerzon often mentioned in his papers regarding the problems of Ambisonics (Gerzon, 1992b). This problem, however, is not strictly apparent as a timbral change (at least, not straight away) when a source or the listener's head moves, but instead probably just aids in confusing the brain as to the sound source's real location, increasing source location ambiguity and source movement when the listener's head is turned. This parameter is more easily observed using an animated graph, but it is shown as a number of stills in Figure 5.9. These graphs show the differences between the three systems, which is why the 'real source' is just a 0dB line, as it has no amplitude difference with itself.

Due to the complexity of the results obtained using the HRTF simulation for the pinna filtering, it is difficult to utilise these results in any estimation of localisation error, although further work will be carried out to make use of this information. However, using the average time and amplitude differences to estimate the perceived direction of the virtual sound source is a relatively trivial task using simple correlation between the actual and virtual sources. In order to plot these results, a Matlab routine was constructed that gave a localisation estimation using the HRTFs derived from the various decoders and compared these to the figures obtained from the real HRTFs. This was carried out for both amplitude and time differences in the various frequency bands tested. Because no pinna filtering effects were taken into account, each value of amplitude and time/phase difference will have two corresponding possible localisation angles (see the cone of confusion in chapter 2.2.1). Figure 5.10, Figure 5.11 and Figure 5.12 show the listening test results with the estimated localisations also shown, using the average amplitude and the average time differences at low and mid frequencies.

The listening tests themselves gave reasonably expected results as far as to the system that performed best (the 2nd Order Ambisonics system). However the other three systems (1st order eight and five speaker, and amplitude panned 5.0) all seemed to perform equally as well, which was not expected. Although it must be noted, that all of these listening tests were carried out using 'unoptimised' decoders, with only the five speaker irregular decoder having been empirically adjusted regarding the amplitude levels of the three speaker sets (centre, front pair and rear pair). Nevertheless, the empirically derived gain settings reasonably matched the optimised sets described later (quiet centre speaker with additional gain applied to the rear pair) but with all speakers using a cardioid pattern feed.

The speakers used for the eight and five speaker systems were different, but as all listeners had the speakers pointed directly at them, and were tested using band-limited noise, the frequency response and dispersion patterns of the speakers should not have been critical in this experiment. Also, the HRTF simulation and comparison should be a valid one as long as the speakers used in each system are matched (as opposed to the speakers across all systems being the same).

The frequency content of the sounds did not seem to make any significant difference to the perceived localisation of the sound sources, although a more extensive test would have to be undertaken to confirm this, as the purpose of this test was to test between any large differences between the three localisation frequency ranges. Another interesting result was the virtual source at 0^0 on the amplitude panned system (see Figure 5.13). As there is a centre front speaker, a virtual source at 0^0 just radiates from the centre speaker, i.e. it is a real source at 0^0. However, around 30% of the subjects recorded that the source came from behind them. Front/back reversals were actually less common in all of the other systems (at 0^0), apart from 2nd order Ambisonics (the system that performed best).

The source position estimation gave reasonably good results when compared with the results taken from the listening tests, with any trends above or below the diagonal, representing a perfect score, being estimated successfully. If the graphs represented truly what is expected from the different types of psychoacoustic sound localisation, then the low frequency time graph and the mid frequency amplitude graph should be the best indicator of where the source is coming from. However it is well known (Gulick *et al.*, 1989) that if one localisation cue points to one direction, and the other cue points to another, then it may be some direction between these two localisation angles that the sound is actually perceived to originate from. The HRTF analysis does not take this into account at the moment and so some error is expected. Also, the compact set of HRTFs used is the minimum phase versions of the actual HRTFs recorded which may contribute to the time difference estimation results (although the cues seem reasonable when looked at for the actual sources). As mentioned, there was no major difference between the three different signals in terms of localisation error. Because of this the plots showing the estimated localisation using the whole frequency range are shown in Figure 5.14 - Figure 5.16 which also show the interaural amplitude difference as a better localisation approximation.

5.2.6 Summary

The HRTF analysis of the three surround systems described in this section seems to work well giving a reasonably good indication as to the possible localisation that a listener will attach to a sound object. This method is definitely worth pursuing as a technique that can be used to evaluate and compare all forms of surround sound systems equally. Although the errors seen in the estimation when compared to the listening test results can be quite large, the general trends were shown accurately, even with such a simple correlation model used.

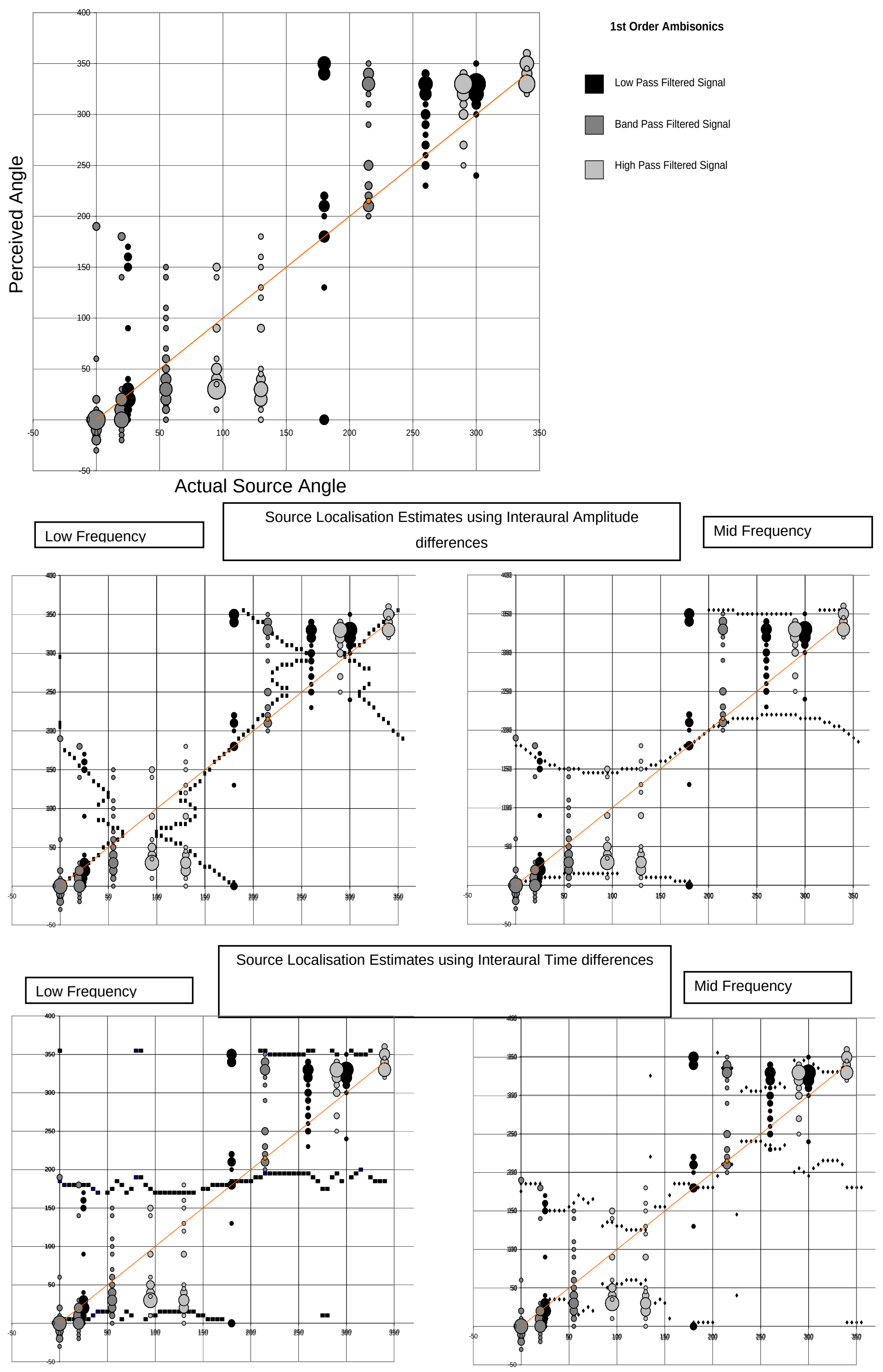
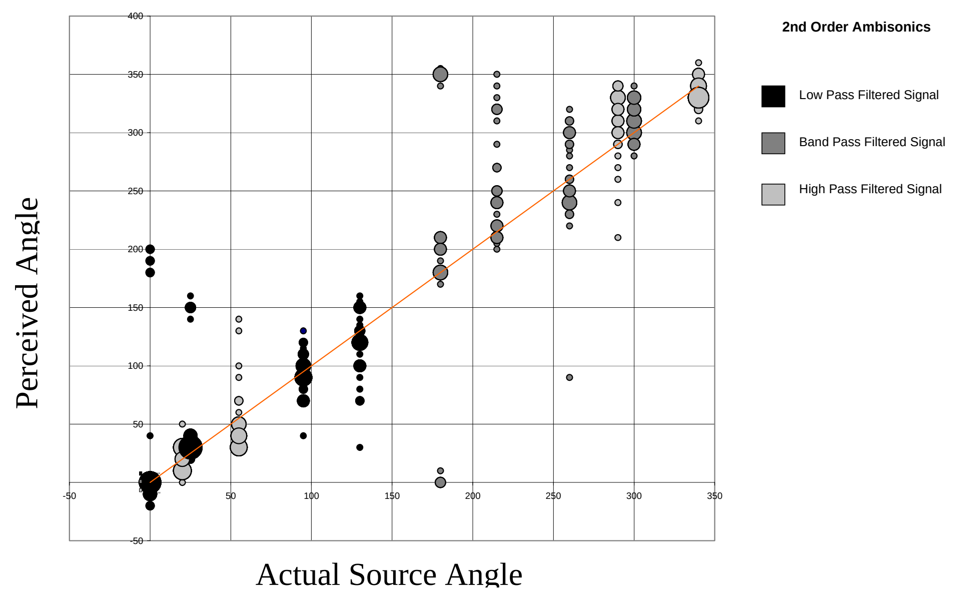
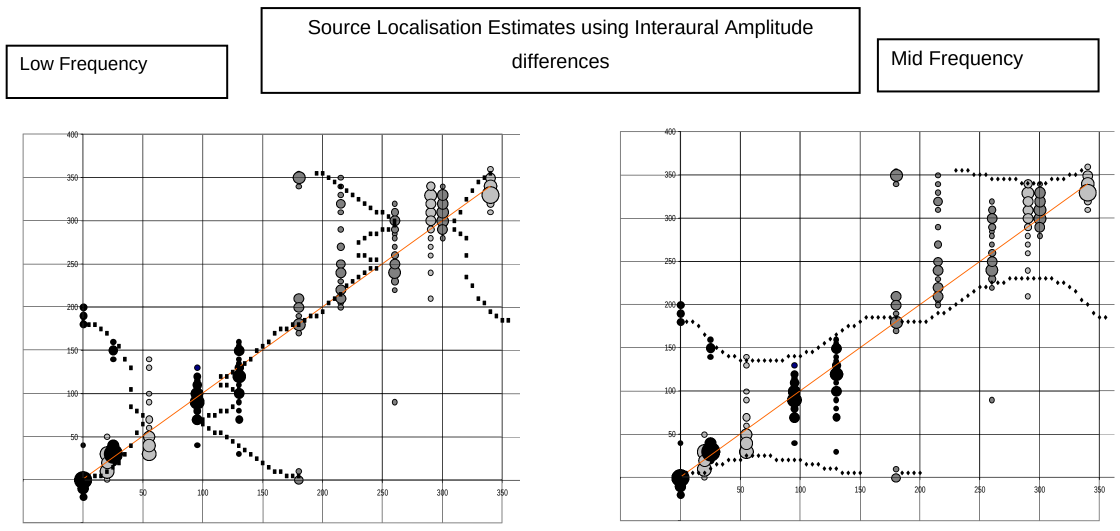


Figure 5.10 **Listening Test results and estimated source localisation for 1st Order Ambisonics**





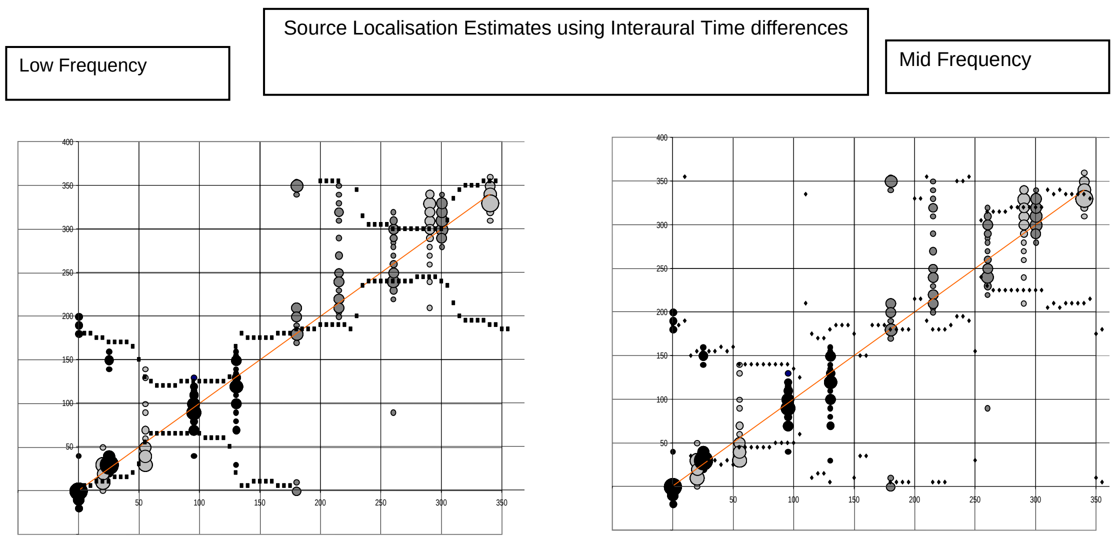
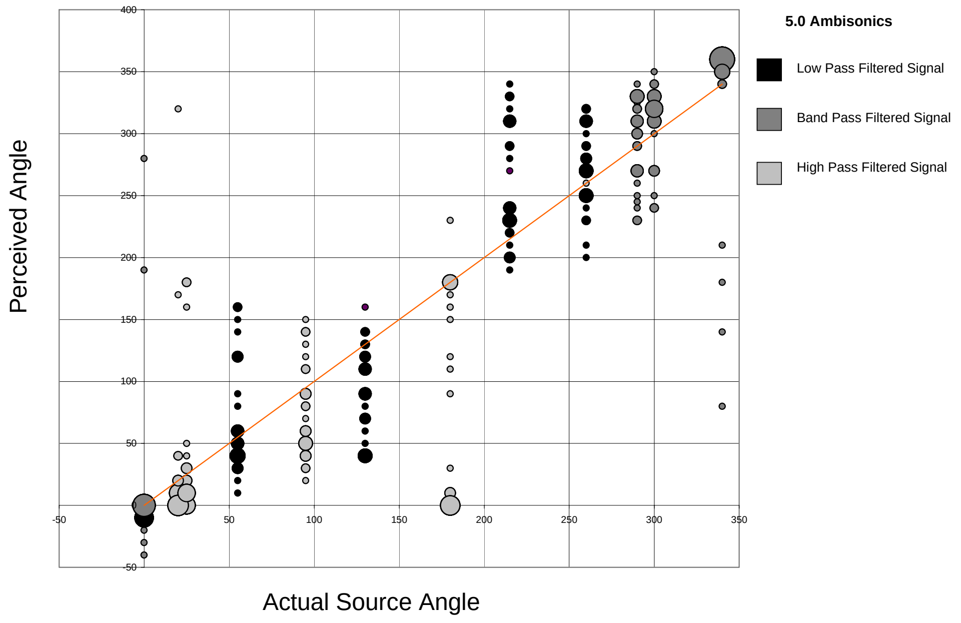


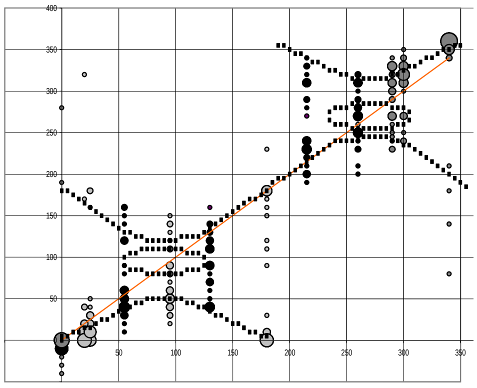
Figure 5.11 Listening Test results and estimated source localisation for 2nd Order Ambisonics

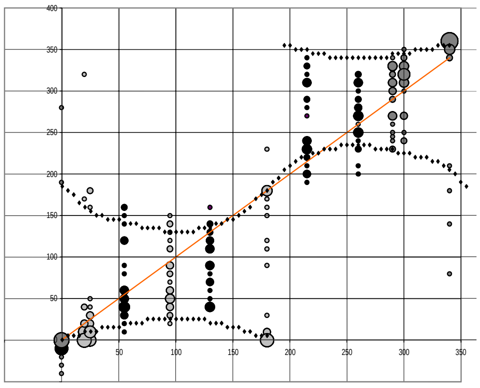


Source Localisation Estimates using Interaural Amplitude differences

Low Frequency

Mid Frequency

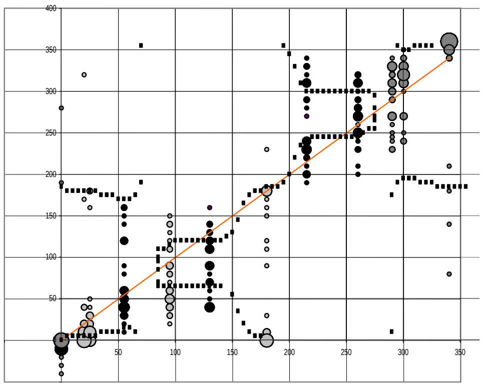




Source Localisation Estimates using Interaural Time differences

Low Frequency

Mid Frequency



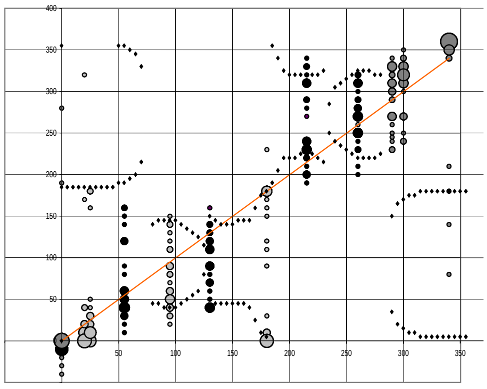


Figure 5.12 Listening Test results and estimated source localisation for five speaker 1st Order Ambisonics

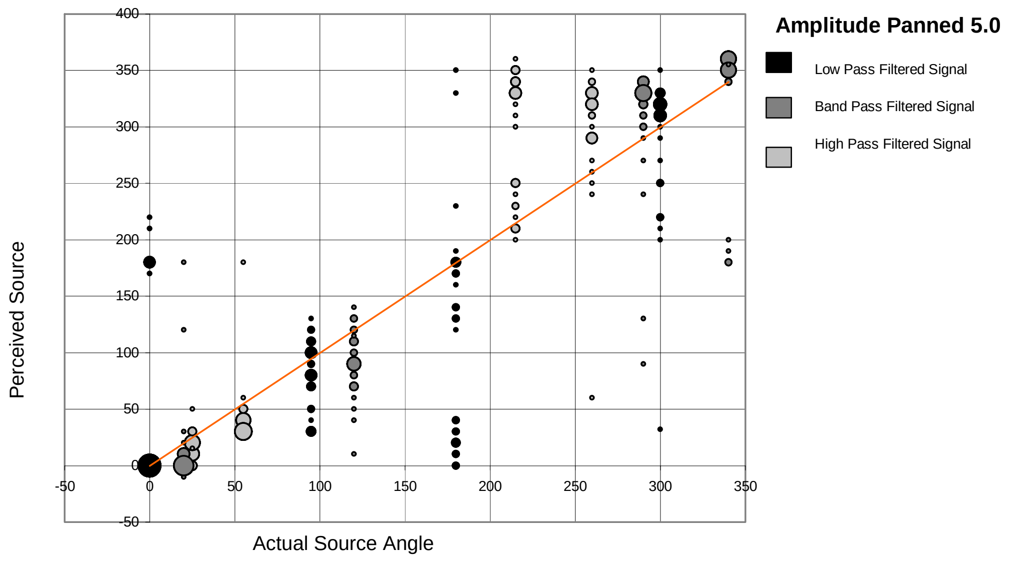


Figure 5.13 Listening test results for Amplitude Panned five speaker system.

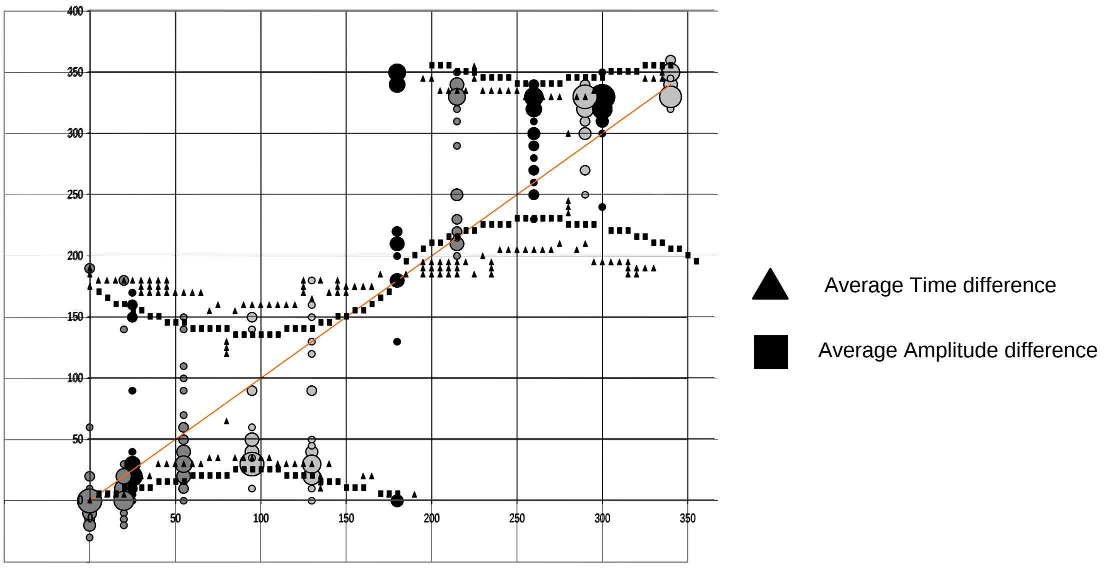


Figure 5.14 Average Time and Frequency Localisation Estimate for 1st Order Ambisonics.

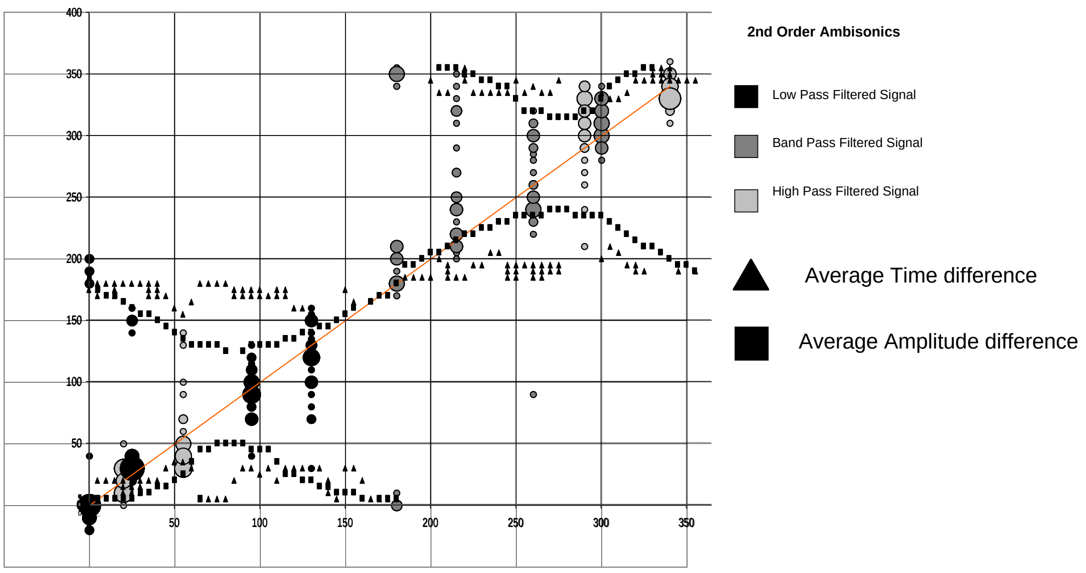


Figure 5.15 **Average Time and Frequency Localisation Estimate for 2nd Order Ambisonics.**

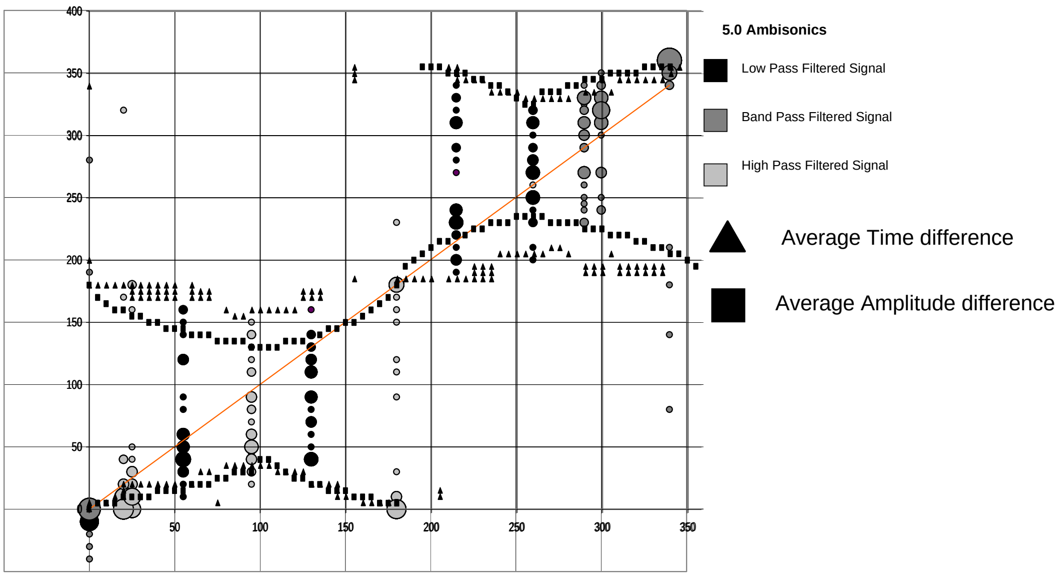
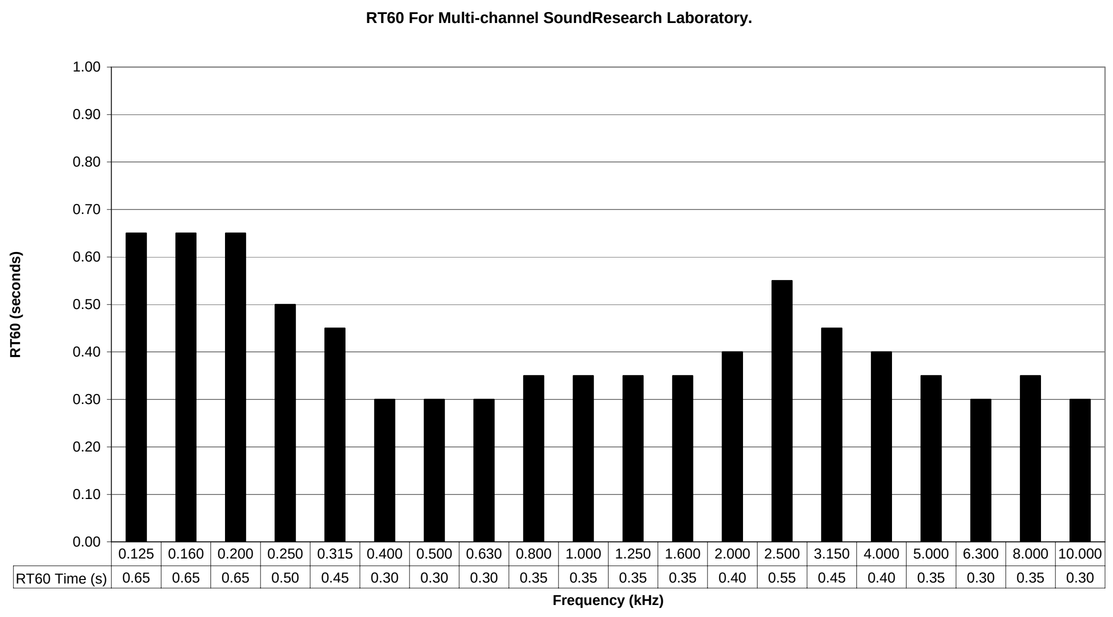


Figure 5.16 **Average Time and Frequency Localisation Estimate for five speaker 1st Order Ambisonics.**



	0.125	0.160	0.200	0.250	0.315	0.400	0.500	0.630	0.800	1.000	1.250	1.600	2.000	2.500	3.150	4.000	5.000	6.300	8.000	10.000
RT60 Time (s)	0.65	0.65	0.65	0.50	0.45	0.30	0.30	0.30	0.35	0.35	0.35	0.35	0.40	0.55	0.45	0.40	0.35	0.30	0.35	0.30

Figure 5.17 RT60 Measurement of the University of Derby's multi-channel sound research laboratory, shown in $^{1}/_{3}$ octave bands.

5.3 Optimisation of the Ambisonics system

5.3.1 Introduction

In this part of the chapter the decoding techniques that have been utilised in the system described in Chapter 4 (Ambisonics, binaural and transaural) will be discussed and optimised so as to both maximise their spatial performance and sound quality. Some of these optimisations are more logically formulated than others, with the optimisation of the Ambisonics system being the most involved, both mathematically and perceptually, so this system will be considered first.

As discussed in Chapter 4, the Ambisonics system will be the basis for the proposed hierarchical multi-channel system, but while the encoding process is a fixed standard (using the spherical harmonics described in Chapter 3) the decoding process is not necessarily as straightforward. As the Ambisonics system is very flexible, any 1st order microphone response can be chosen, along with the virtual microphone's direction. Gerzon's original theory stated that the virtual microphone response for the decoder (he concentrated on regular setups initially) should be chosen according to a number of mathematical approximations to the signals that would reach the ear of a listener (Gerzon, 1974) and, for regular speaker arrays, this was a relatively straightforward optimisation to perform (see section 3.3.1.2). However, since the introduction of the DVD, the standard speaker layout as specified by the ITU is a five speaker layout as shown in Figure 5.18. This is likely to be expanded upon in the near future, and other, larger, venues are likely to have more speakers to cover a larger listening area.

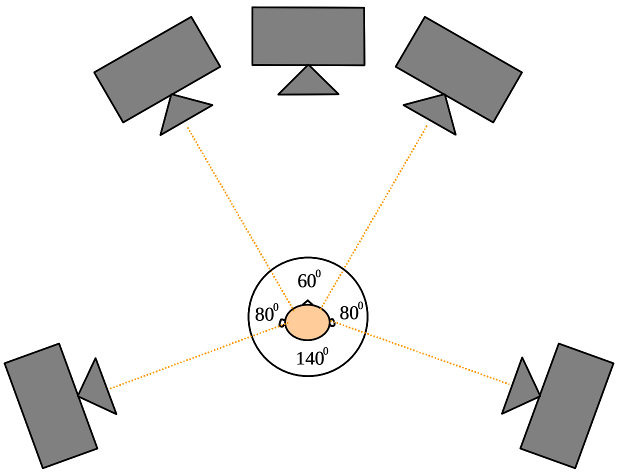


Figure 5.18 Recommended loudspeaker layout, as specified by the ITU.

Due to the likelihood of ever changing reproduction layouts a more portable approach should be used in the creation of multi-channel material, and such a system has been around since the 1960s (Borwick, 1981).

Ambisonic systems are based on a spherical decomposition of the sound field to a set order (typically 1st or 2nd order (Malham, 2002; Leese, n.d.)). The main benefit of the Ambisonic system is that it is a hierarchical system, that is, once the sound field is encoded in this way (into four channels for 1st order, and 9 channels for 2nd order) it is the decoder that decides how this sound field is reconstructed using the Ambisonic decoding equations (Gerzon, 1977b). This system has been researched, mainly by Gerzon, and in 1992 papers were published suggesting a method of optimising Ambisonic decoders for irregular speaker arrays (Gerzon & Barton, 1992) as the original decoding equations were difficult to solve for irregular speaker arrays in the conventional way (use of shelving filters (Gerzon, 1974)).

5.3.2 Irregular Ambisonic Decoding

In order to quantify decoder designs Gerzon decided on two main criteria for designing and evaluating multi-speaker surround sound systems in terms of their localisation performance. These represent the energy and velocity vector components of the sound field (Gerzon, 1992c). The vector lengths represent a measure of the ‘quality’ of localisation, with the vector angle representing the direction that the sound is perceived to originate from, with a vector length of one indicating a good localisation effect. These are evaluated as shown in Equation (5.3)

$$\begin{aligned}
P &= \sum_{i=1}^{n} g_i & E &= \sum_{i=1}^{n} g_i^2 \\
Vx &= \sum_{i=0}^{n} g_i \cos(\theta_i)/P & Ex &= \sum_{i=0}^{n} g_i^2 \cos(\theta_i)/E \\
Vy &= \sum_{i=0}^{n} g_i \sin(\theta_i)/P & Ey &= \sum_{i=0}^{n} g_i^2 \sin(\theta_i)/E
\end{aligned}$$

(5.3)

Where:
g_i represents the gain of a speaker (assumed real for simplicity).
n is the number of speakers.
θ_i is the angular position of the ith speaker.

For regular speaker arrays, this was simply a case of using one virtual microphone response for low frequencies and a slightly different virtual microphone response for the mid and high frequencies by the use of shelving filters (Farino & Uglotti, 1998) as shown in Figure 5.19 and Figure 5.20. This is extremely similar to the theory and techniques used by Blumlein’s spatial equalisation described in Chapter 2.

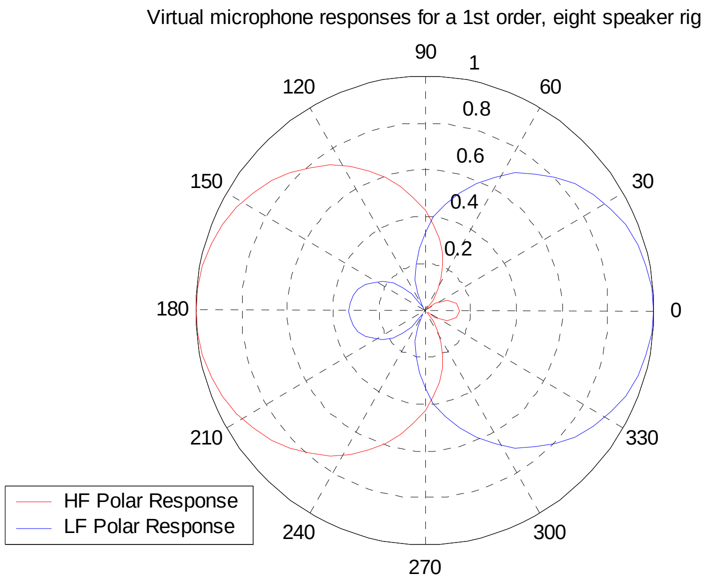


Figure 5.19 **Virtual microphone polar plots that bring the vector lengths in Equation (5.3) as close to unity as possible (as shown in Figure 5.21), for a 1st order, eight speaker rig.**

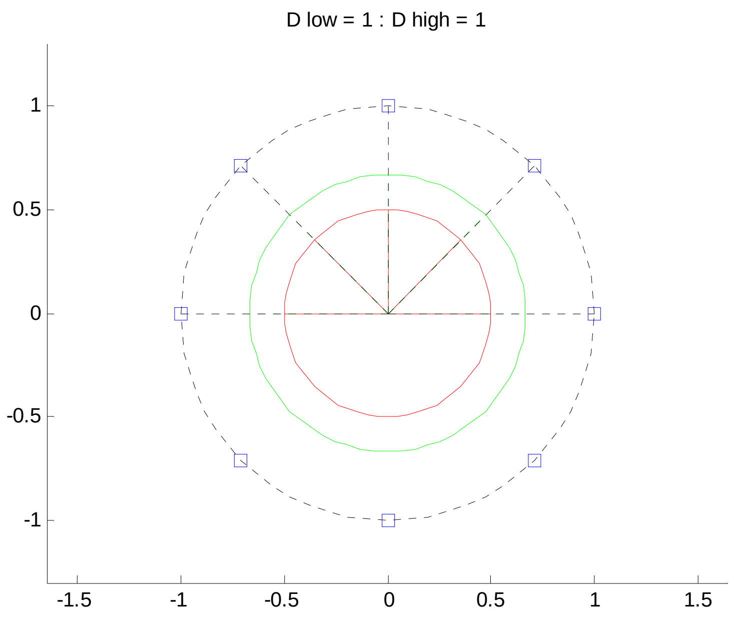


Figure 5.20 **Velocity and energy localisation vectors. Magnitude plotted over 360^0 and angle plotted at five discrete values. Inner circle represents energy vector, outer circle represents velocity vector. Using virtual cardioids.**

As long as the virtual microphone patterns were the same for each speaker, the localisation angle was always the same as the encoded source angle, just the localisation quality (length of the vector) was affected by changing the polar patterns.

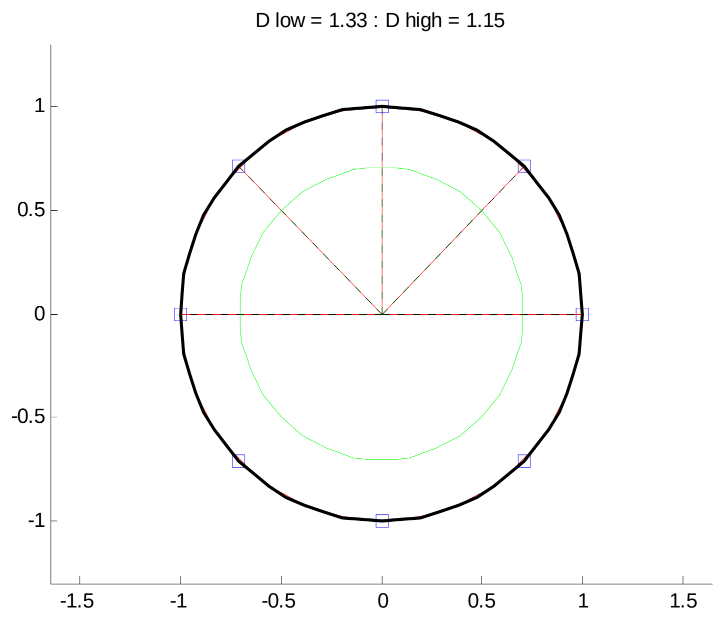


Figure 5.21 **Velocity and energy localisation vectors. Magnitude plotted over 360^0 and angle plotted at five discrete values. Inner circle represents energy vector, outer circle represents velocity vector. Using virtual patterns from Figure 5.19.**

However, when non-regular speaker arrays are used, not only do the vector magnitudes need to be compensated for, but the replay angle and overall volume of the decoded sound need to be taken into account. This results from the non-uniformity of the speaker layout. For example, if all of the speakers had the same polar pattern then a sound encoded to the front of a listener would be louder over an ITU five speaker system than a sound emanating from the rear, due to the higher density of speakers at the front of the speaker array. Also, the perceived direction of the reproduced sound would also be distorted, as shown in Figure 5.22.

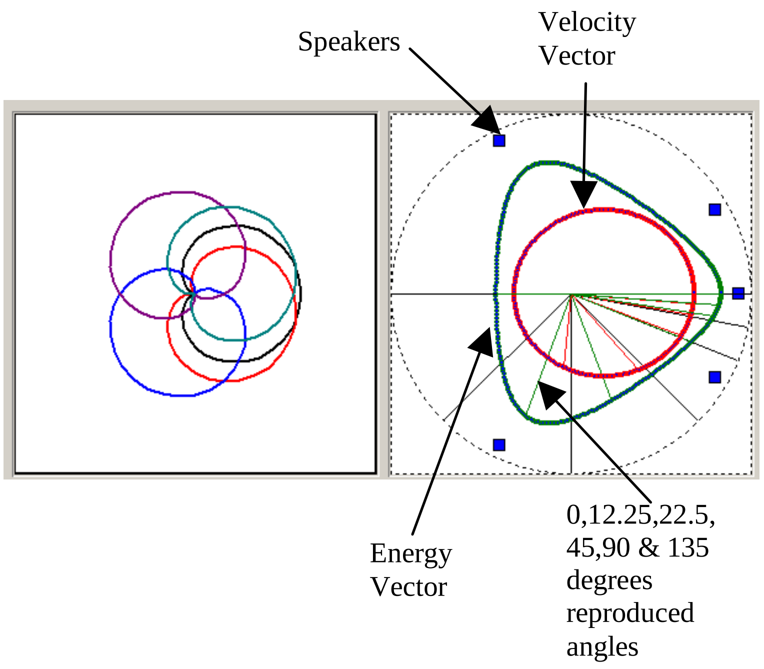


Figure 5.22 Energy and velocity vector response of an ITU 5-speaker system, using virtual cardioids.

These artefacts are not a problem when you are producing audio for a fixed setup (i.e. amplitude panned 5.1) as material is mixed so it sounds correct on the chosen speaker layout. However, as the point of using a hierarchical surround sound format is that an audio piece should sound as similar as possible on as many speaker layouts as possible, these artefacts must be corrected after the encoding has occured, that is, during the decoding stage.

Due to the added complexity of the speaker array's response to an Ambisonic system, Gerzon and Barton (1992) proposed that two separate decoders be used, one for low frequency (<~700Hz) and another for high frequencies (>~700 Hz). This can be achieved using a simple cross-over network feeding low and high passed versions of the Ambisonic B-format signals to the two decoders. It is also important that the cross-over filters are perfectly phase matched so that the reinforcement and cancellation principles used by Ambisonics still function correctly.

5.3.3 Decoder system

1st order Ambisonics is comprised of four different signals, as shown in Figure 5.23, and comprises of an omni-directional pressure signal (W), a front-back figure of eight (X), a left-right figure of eight (Y), and an up-down figure of eight (Z).

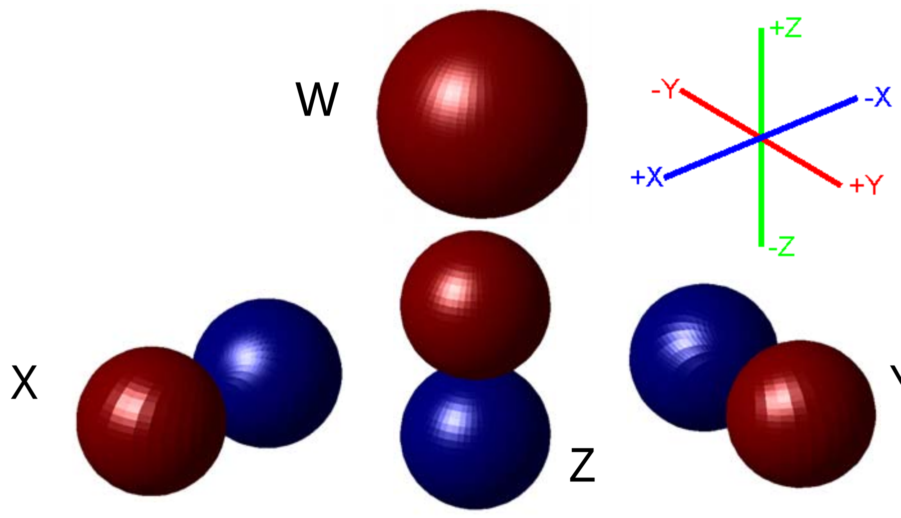


Figure 5.23 Polar patterns of the four B-format signals used in 1st order Ambisonics.

As the 5-speaker system shown in Figure 5.18 is a horizontal only system, only three of the four available B-format signals are needed to feed the decoder (W, X and Y). Also, as the speaker array in Figure 5.18 is left/right symmetric, we can also assume that the decoder coefficients work in pairs (i.e. sums and differences). The Ambisonic encoding equations are given in Equation (5.4).

$$\begin{aligned} W &= 1/\sqrt{2} \\ X &= \cos(\theta) \\ Y &= \sin(\theta) \end{aligned} \tag{5.4}$$

where θ is the encoded angle, taken anti-clockwise from straight ahead.

As another tool in the decoding of the sound field, it will be seen that the use of a 'frontal dominance' parameter is useful, as shown in Equation (5.5). This is not the best form of the frontal dominance equation (it has a non-linear response to the dominance parameter), but it is used to keep compatibility with Gerzon's previous paper on this subject (Gerzon & Barton, 1992).

$$\begin{aligned}
W' &= 0.5\left(\lambda + \lambda^{-1}\right)W + 8^{-\frac{1}{2}}\left(\lambda - \lambda^{-1}\right)X \\
X' &= 0.5\left(\lambda + \lambda^{-1}\right)X + 2^{-\frac{1}{2}}\left(\lambda - \lambda^{-1}\right)W \\
Y' &= Y
\end{aligned}$$

(5.5)

where λ is the forward dominance parameter (>1 for front, and <1 for rear dominance).

These encoding equations are then substituted into the decoding equations to give a numerical value for each speaker's output to a particular signal as given in Equation (5.6). In this equation it can be seen that what were previously sine and cosine (i.e. directionally dependant) weightings are now arbitrary values (nominally to be chosen between 0 and 1), denoted by kW, kX and kY.

$$\begin{aligned}
C_F &= (kW_C \times W') + (kX_C \times X') \\
L_F &= (kW_F \times W') + (kX_F \times X') + \left(kY_F \times Y'\right) \\
R_F &= (kW_F \times W') + (kX_F \times X') - \left(kY_F \times Y'\right) \\
L_B &= (kW_B \times W') + (kX_B \times X') + \left(kY_B \times Y'\right) \\
R_B &= (kW_B \times W') + (kX_B \times X') - \left(kY_B \times Y'\right)
\end{aligned}$$

(5.6)

where k denotes a decoding coefficient (e.g. kW_c represents the weighting given to the W channel for centre front speaker).

$_F$, $_B$ and $_C$ denote front, back and centre speakers respectively.

W',X' and Y' represent the incoming B-format signals after potential transformation by the forward dominance equation.

C, L and R denote centre, left and right speakers

The values for λ and the 'k' values are to be chosen to optimise the decoder's output, with λ having possible values between 0 and 2, and 'k' values having a nominal range between 0 and 1.

Equation (5.7) shows the conditions which are used to assess the performance of a given solution. The conditions that must be met are:

Radius of the localisation vector lengths (R_V and R_E) should be as close to 1 as possible for all values of θ.

$\theta = \theta_V = \theta_E$ for all values of θ.

$P_V = P_E$ and must be constant for all values of θ.

$$Vx = \sum_{i=1}^{N} g_i \times \cos(SPos_i) / P_V$$

$$Vy = \sum_{i=1}^{N} g_i \times \sin(SPos_i) / P_V$$

$$Ex = \sum_{i=1}^{N} g_i^2 \times \cos(SPos_i) / P_E$$

$$Ey = \sum_{i=1}^{N} g_i^2 \times \sin(SPos_i) / P_E$$

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

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

$$P_V = \sum_{i=1}^{n} g_i \qquad P_E = \sum_{i=1}^{n} g_i^2$$

(5.7)

where:

g_i = Gain of the ith speaker

$SPos_i$ = Angular position of the ith speaker.

V denotes velocity vector

E denotes energy vector

The reason that these equations are difficult to solve is that the best result must be found over the whole listening area, spanning 360^0. Even Gerzon admitted that these equations were laborious to solve for five speakers, and the more speakers present, i.e. the more values that must be optimised, the more laborious and time consuming finding the solution becomes. Also, there is more than one valid solution for each decoder (low frequency and high frequency) meaning that a group of solutions need to be found, and then auditioned, to determine the best set of coefficients.

A system that can automatically calculate decoder coefficients is needed, and possibly one that can distinguish between sets of coefficients that meet the criteria set out by the energy and velocity vector theories. This system does not need to be particularly fast, as once a group of solutions are found the program should not need to be used again, unless the speaker layout changes.

5.3.4 The Heuristic Search Methods

As a result of the fact that each parameter in the Ambisonic decoding equations will have a value within a well defined range, 0 to 1 or 0 to 2, a search method offers an effective solution to the array optimisation problem. However, if we wish to determine the settings to two decimal places there are 2 x 10^{18} possible solutions (given that there are 9 search parameters) and an exhaustive search is not feasible (Wiggins *et al*, 2003). When deciding on the type of heuristic method, an empirical approach was used. The most important part of any heuristic search method is the development of the fitness equations. These are the functions that give the heuristic search method the measure of the success of its choice. Care must be taken when choosing these functions to make sure that it is not possible for different error conditions to cancel each other out with the most logical solution to this problem being to ensure that any error in the decode results in a positive number. The fitness equations developed for this project are described later in this chapter. The first avenue of research taken was that of a Genetic Algorithm approach, as this is one of the better known heuristic methods. This was first implemented as a Matlab script and did not seem to converge to a good result, so the next system to try was one using an algorithm based on the Tabu search as this has been shown to converge more accurately when used in a small search space (Berry, S. & Lowndes V., 2001). It was while developing this algorithm that it was discovered that the initial velocity and energy vector calculations contained errors, and once corrected, the Tabu search algorithm performed as expected. As this tabu algorithm performed well, the genetic algorithm was not tried again at this point due to its known convergence problems as described above (Genetic Algorithms are better suited to a very large search space, which this problem did not have).

This adapted form of Tabu search works by having the decoder coefficients initialised at random values (or values of a previous decoder, if these values are to be optimised further). Then the Tabu search program tries changing each of the 'tweakable' values, plus or minus the step size. The best result is then kept and the parameter changed is then restricted to only move in the successful direction for a set number of iterations (which, of course, will only happen if this parameter, again, is the best one to move). It must be noted that the random start position is of great importance, as it is this that helps in the search for a wide range of solutions.

The most important part of the Tabu search algorithm is the equations used to measure the fitness of the decoder coefficients, as it is this one numerical value that will determine the course that the Tabu search takes. As mentioned above, three parameters must be used in an equation that represents the overall fitness of the decoder coefficients presented. These are:

- Localisation measure (vector lengths, R_V & R_E).
- Localisation Angle (vector angles, θ_V & θ_E).
- Volume (Sound pressure gain, P_V & energy gain, P_E) of each encoded direction.

As each of the parameters must be as good a fit as possible for the whole 360^0 sound stage, the three parameters must be evaluated for a number of different encoded source positions. Gerzon evaluated these parameters at 14 points around the unit circle (7 around a semi-circle assuming left/right symmetry), but as computers can calculate these results so quickly, an encoded source resolution of 4^0 intervals would be used (90 points around the unit circle). Due to the large number of results for each of the fitness values an average was taken for each fitness parameter using a root mean square approach. If we take the example of the fitness of the vector lengths (localisation quality parameter), then if a mean average is taken, a less than one vector length in one part of the circle could be compensated for by a greater than one vector length elsewhere. However, if we take a good fit to be zero, and use a root mean square approach then a non-perfect fit around the circle will always give a positive error value, meaning that it is a true measure of the fitness. The equations used for each of the fitness parameters are shown in Equation (5.8).

$$VFit = \sqrt{\sum_{i=0}^{n} \frac{\left(1 - {P_0}/{P_i}\right)^2}{n}}$$

$$MFit = \sqrt{\sum_{i=0}^{n} \frac{(1 - R_i)^2}{n}}$$

$$AFit = \sqrt{\sum_{i=0}^{n} \frac{\left(\overset{Enc}{\theta}{}_i - \theta_i\right)^2}{n}}$$

(5.8)

where:

P_0 is the pressure at an encoded direction of 0^0.

R represents the length of the vector at a direction, i.

n is the number of points taken around the unit circle.

$\overset{Enc}{\theta}$ is the encoded source angle and θ is the localisation angle.

V, M and AFit are the numerical fitness parameters used to measure the performance of a particular decoder (Volume, Magnitude and Angle).

Given the three measures of fitness in Equation (5.8), the overall fitness for the high and low frequency versions of the decoder are actually calculated slightly differently. The low frequency decoder can achieve a near perfect fit, but the best fit that the high frequency decoder can expect to achieve is shown in Figure 5.32. The best results were obtained from the Tabu search algorithm if the overall fitness was weighted more towards the angle fitness, Afit from Equation (5.8), as shown in Equation (5.9).

$$LFFitness = AFit + MFit + VFit$$
$$HFFitness = AFit + (MFit + VFit)/2$$

(5.9)

A block diagram of the tabu search algorithm used in this research is shown in Figure 5.24.

The main benefit of the Tabu search method is that all three of the conditions to be met can be optimised simultaneously, which had not been accomplished in Gerzon’s Vienna paper (Gerzon & Barton, 1992). For example if we take the speaker layout used in the Vienna paper, which is not the ITU standard but is reasonably similar (it is a more regular layout than the one the ITU specified after Gerzon’s paper was published), then the coefficients derived by Gerzon and Barton would give an energy and velocity vector response as shown in Figure 5.25. Several points are apparent from this figure. There is a high/low localisation angle mismatch due to the forward dominance being applied to the high frequency decoder’s input *after* the localisation parameters were used to calculate the values of the coefficients (as first reported in Wiggins *et al*., 2003). If the frontal dominance is applied to both the high and low frequency decoders, a perceived volume mismatch occurs with the low frequency decoder replaying sounds that are louder in the frontal hemisphere than in the rear. Also, even if these mismatches were not present (that is, the frontal dominance is not applied) every set of results presented in the Vienna paper showed a distortion of the decoder’s reproduced angles. Figure 5.25 shows a set of coefficients calculated using the Tabu search algorithm described in Figure 5.24 and demonstrates that if all three criteria are optimised simultaneously a decoder can be designed that has no angle or volume mismatches, and should reproduce a recording more faithfully than has been achieved in previous Ambisonic decoders for irregular arrays.

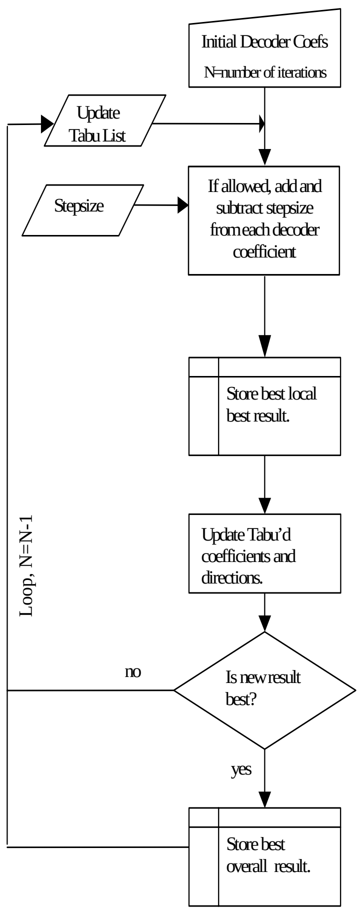

Initial Decoder Coefs
N=number of iterations
Update Tabu List
Stepsize
If allowed, add and subtract stepsize from each decoder coefficient
Store best local best result.
Update Tabu'd coefficients and directions.
Loop, N=N-1
no
Is new result best?
yes
Store best overall result.

Figure 5.24 **A simple Tabu Search application.**

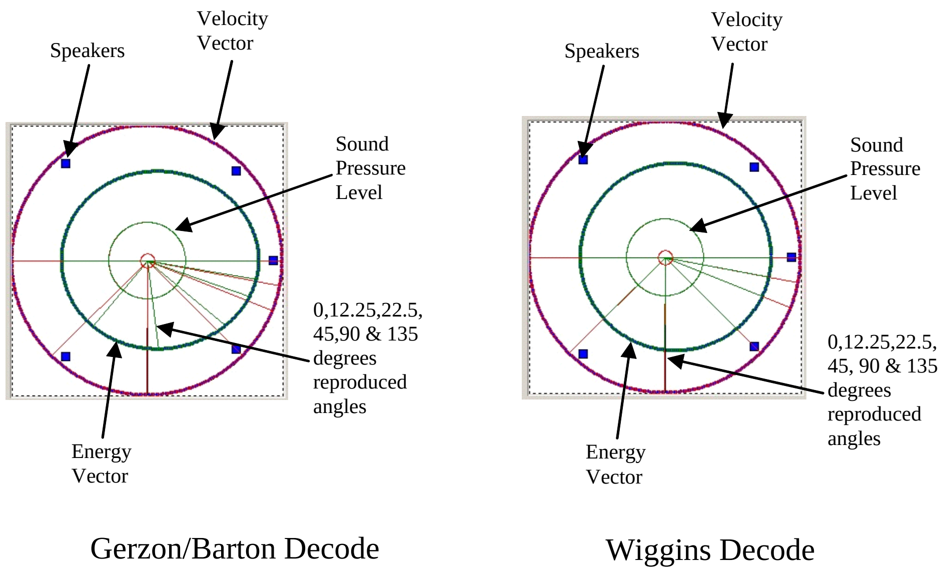


Figure 5.25 **Graphical plot of the Gerzon/Barton coefficients published in the Vienna paper and the Wiggins coefficients derived using a Tabu search algorithm. Encoded/decoded direction angles shown are 0^0, 12.25^0, 22.5^0, 45^0, 90^0, 135^0 and 180^0.**

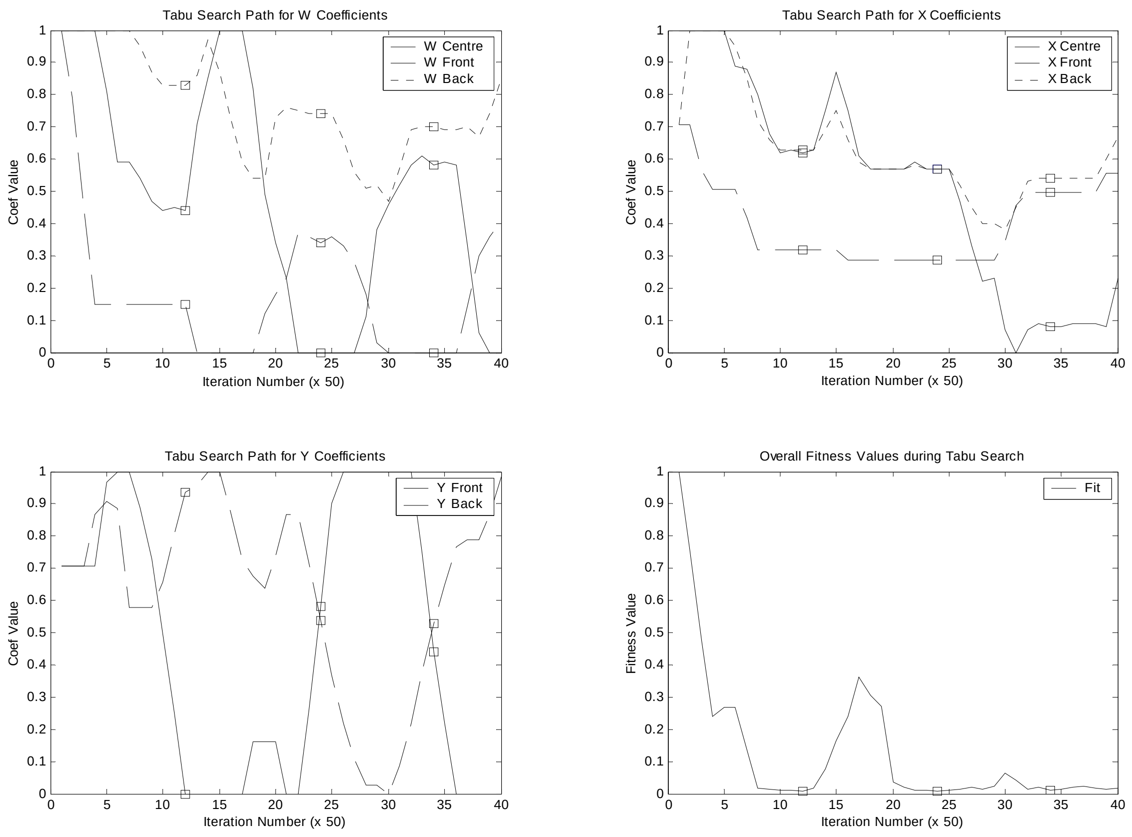


Figure 5.26 **The transition of the eight coefficients in a typical low frequency Tabu search run (2000 iterations). The square markers indicate the three most accurate sets of decoder coefficients (low fitness).**

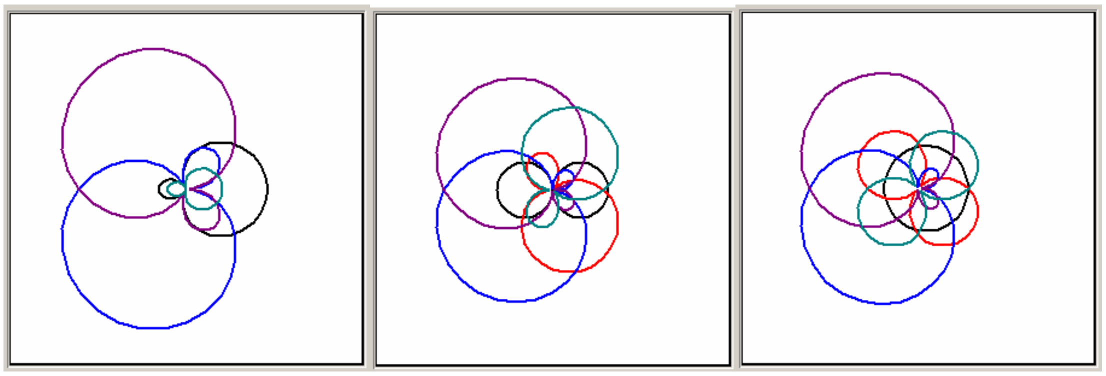


Figure 5.27 **The virtual microphone patterns obtained from the three optimum solutions indicated by the squares in figure 5.25.**

While writing up this research thesis, Craven (2003) released a paper detailing how 4th order circular harmonics (i.e. Ambisonic, spherical harmonics without the height information) could be used to create an improved panning law for irregular speaker arrays. The example decoder Craven includes in his paper has the velocity/energy vector representation and virtual microphone patterns as shown in Figure 5.28 and Figure 5.29 respectively.

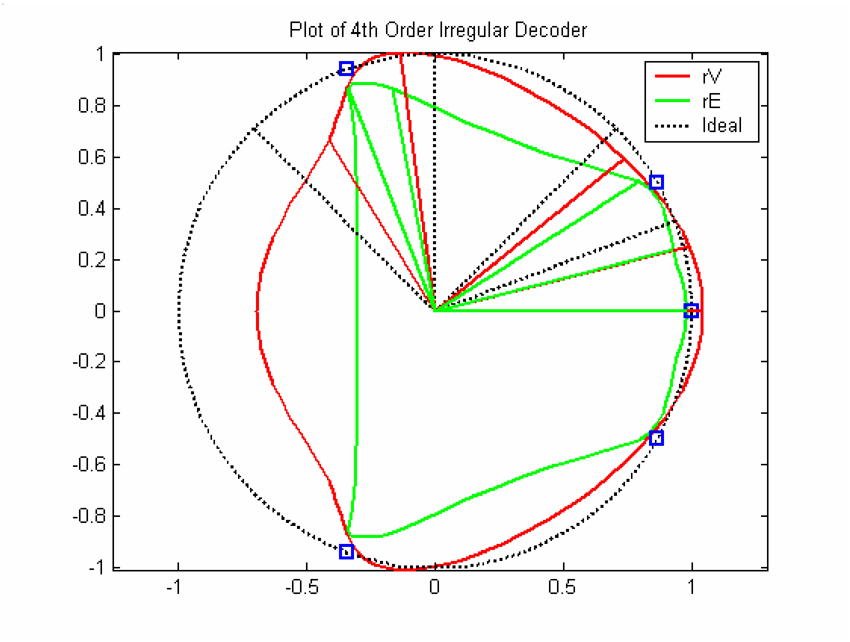


Figure 5.28 **Energy and Velocity Vector Analysis of a 4th Order Ambisonic decoder for use with the ITU irregular speaker array, as proposed by Craven (2003).**

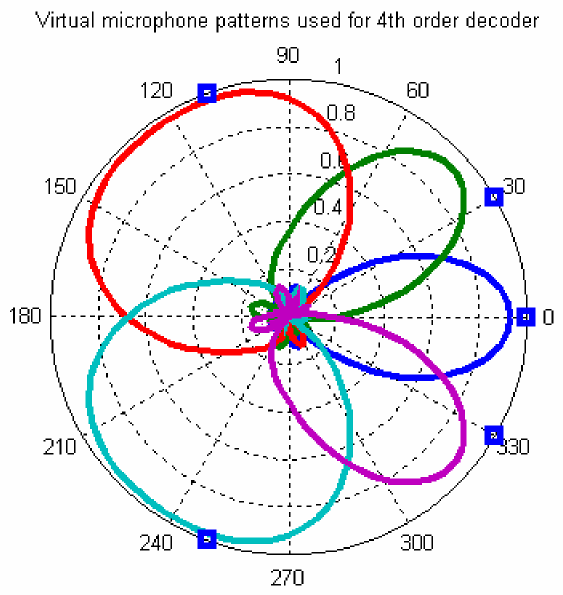
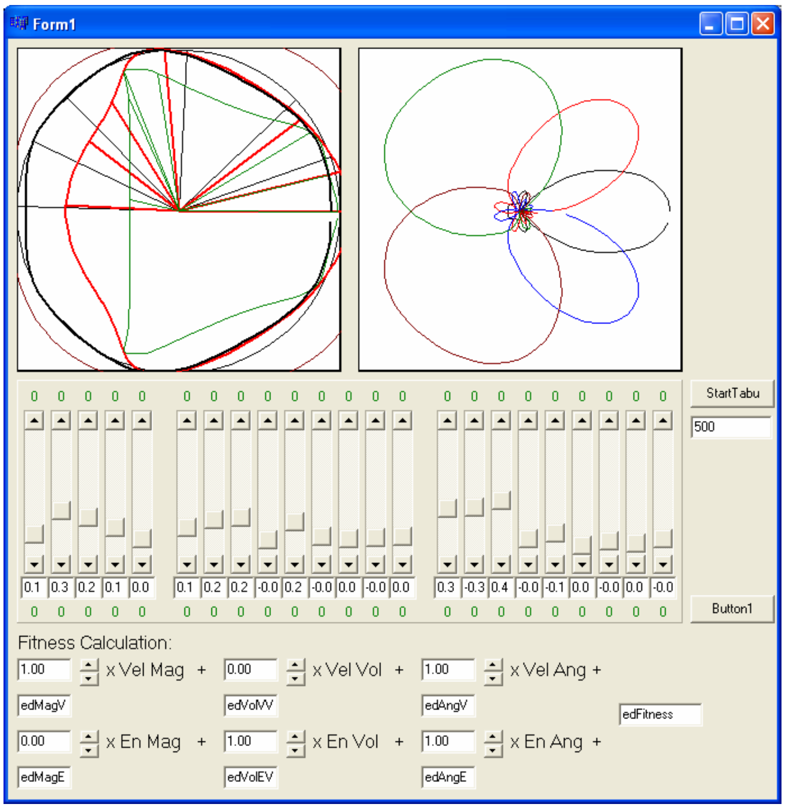


Figure 5.29 **Virtual microphone patterns used for the irregular Ambisonic decoder as shown in Figure 5.28.**

The method Craven used to derive this new decoder is not detailed in his paper, and he has opted for a frequency independent decoder, no doubt, in order to make the panning law easily realisable on current software/hardware platforms. It can be seen that the performance of the high frequency energy vector analysis is very good, with respect to the vector length, however, the matching of the high and low frequency vector angles is not ideal, and also the vector length of the low frequency velocity vector should be designed as close to 1 as possible (Gerzon & Barton, 1992). These problems are mostly due to the fact that a frequency independent decoder has been presented, so any decoder will always be a compromise between optimising for the energy vector and optimising for the velocity vector's three fitness parameters of length, perceived direction, and perceived amplitude. However, using the Tabu method just described, it is a simple matter of changing the weightings of the fitness equations, as shown in equations (5.8) and (5.9), in order to design a decoder with more coherent lateralisation cues.

In order to experiment with higher order decoder optimisation, a new Tabu search application was developed, using the same fitness criterion as before, but with user editable weighting functions. A Screenshot of this can be seen in Figure 5.30.



0 0 0 0 0   0 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0 0

0.1 0.3 0.2 0.1 0.0   0.1 0.2 0.2 -0.0 0.2 -0.0 0.0 -0.0 0.0   0.3 -0.3 0.4 -0.0 -0.1 0.0 -0.0 0.0 -0.0

0 0 0 0 0   0 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0 0

500

Fitness Calculation:

1.00 x Vel Mag + 0.00 x Vel Vol + 1.00 x Vel Ang +

edMagV edVolVV edAngV

edFitness

0.00 x En Mag + 1.00 x En Vol + 1.00 x En Ang +

edMagE edVolEV edAngE

Figure 5.30 Screenshot of the 4th Order Ambisonic Decoder Optimisation using a Tabu Search Algorithm application.

The sets of up/down arrows in the 'Fitness Calculation' box are where the user can set the weightings of each of the individual fitness values, in order to influence the performance of the Tabu search algorithm. It can be seen, in Figure 5.30, that the perceived volume fitness is governed by the Energy (‘En Vol’, high frequency) rather than the pressure (‘Vel Vol’, low frequency). Due to the frequency independent nature of these decoders, one or the other must be chosen, and as the energy vector covers a much wider frequency band for a centre listener (>700 Hz) and an even larger frequency band for off-centre listeners, it is always advisable to use the average energy as an indicator for the perceived amplitude of a decoded source (Gerzon, 1977a).

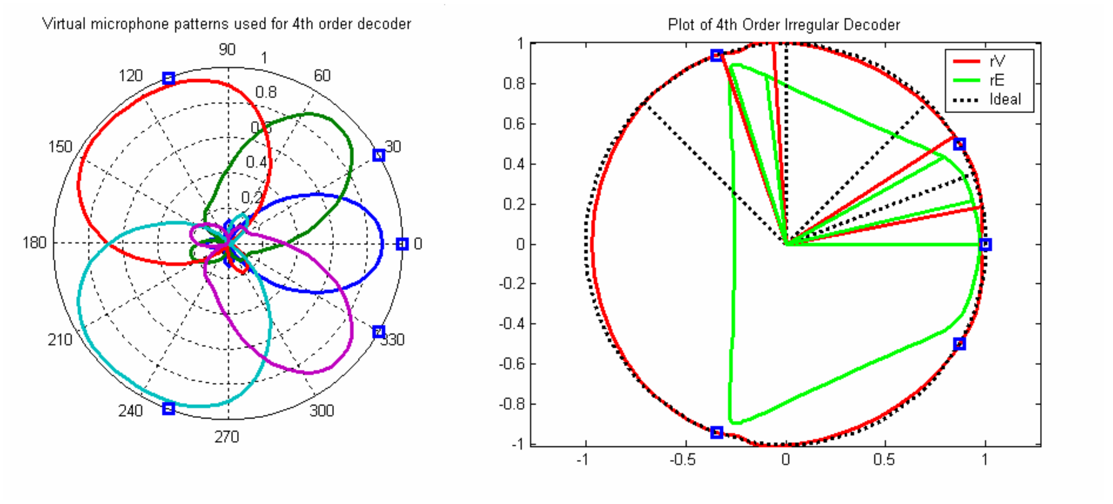


Figure 5.31 Graph showing polar pattern and velocity/energy vector analysis of a 4th order decoder optimised for the 5 speaker ITU array using a tabu search algorithm.

Figure 5.31 shows a 4th order decoder optimised by the Tabu search application shown in Figure 5.30. It can clearly be seen that although the length (and therefore, shape) of the energy vector plot is very similar to that of Craven’s decoder shown in Figure 5.28, showing a similar performance, this Tabu search optimised decoder shows improvements in other aspects:

- The low frequency velocity vector has a length much closer to 1 for a source panned in any direction.
- The low and high frequency perceived directions are in better agreement.

The optimisation of a 4th order decoder as proposed by Craven (2003) shows the robust and extensible nature of the tabu search algorithm described in this report, as over double the number of alterable parameters (23 as opposed to 9) were used in this program.

5.3.5 Validation of the Energy and Velocity Vector

It can be seen in Figure 5.26 and Figure 5.27 that, according to the velocity vector, it is possible to design a low frequency decoder that satisfies all of the fitness parameters discussed in the previous section. This is even possible when the ITU standard speaker layout is used (although the high frequency decode suffers, theoretically, in this configuration) as shown in Figure 5.32. If we take the velocity vector as a measure of the low frequency localisation, which is dominated by time/phase differences between the ears, and the energy vector as a measure of the mid frequency localisation, which is dominated by level differences between the ears, then this theory can be tested using head related transfer functions (Wiggins *et al.*, 2001). The HRTF data is used from (Gardner & Martin, 1994). Assuming the head will remain pointing straight ahead, the speakers will remain in a fixed position in relation to the head and time and level difference plots can be obtained.

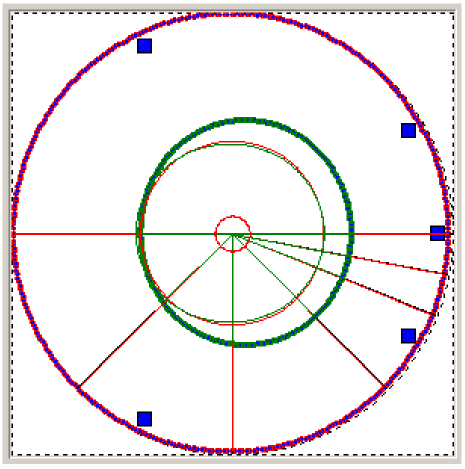


Figure 5.32 A decoder optimised for the ITU speaker standard.

Using the average group delay between 0 and 700Hz to obtain the time differences between the ears and the average magnitude between 700Hz and 3 kHz, reference plots can be calculated, which the decoder's output must follow in order to fool the ear/brain system successfully. The head related transfer functions for the Ambisonic array can be calculated in one of two ways:

- A pair of HRTFs can be applied to each speaker's output, and then left and right ear responses are summed resulting in a single response pair (for each encoded direction)

- The decoder can be encoded into a pair of HRTFs for each input signal (W,X and Y in this case) using the method described in section 5.2.4

Both of the above methods ultimately arrive at the same results and if only off-line analysis is needed, then either of these methods can be chosen (the 2nd is computationally more efficient if auralisation of the decoder is desired (Wiggins, *et al.*, 2001) and becomes more efficient the greater the number of speakers used, when compared to the 1st method). Two resulting pairs of HRTF responses have been produced for encoded sources all around a listener, one pair for the low frequency decoder, and one pair for the high frequency decoder.

A graph showing the level and time differences of real and Ambisonically decoded signals is shown in Figure 5.33 (note, an Ambisonic decode to a five speaker rig is often referred to as G format).

The HRTF analysis graphs have been constructed using the anechoic HRTFs measured by MIT (Gardner B., Martin K., 1994). A real source is taken as a single pair of these HRTFs, and the Ambisonic (G-Format) output has been constructed from a combination of these anechoic HRTFs weighted to various degrees depending on the simulated source direction (i.e. a simulation of an Ambisonic decode). When using the HRTF analysis, the low frequency range was 0 Hz – 700 Hz, and the mid frequency range was from 700 Hz – 3 kHz. The 700 Hz value was used so the results could be directly compared to the velocity and energy vector analysis used by Gerzon & Barton (1992) with the 3 kHz value used as a nominal value. The x-axis scale in these graphs represents either a real or synthesised Ambisonic source position in degrees. The y-axis scaling represents either the average time difference (in samples, sampled at 44.1 kHz) or the average amplitude difference, measured linearly, with an amplitude of one representing 0 dB gain.

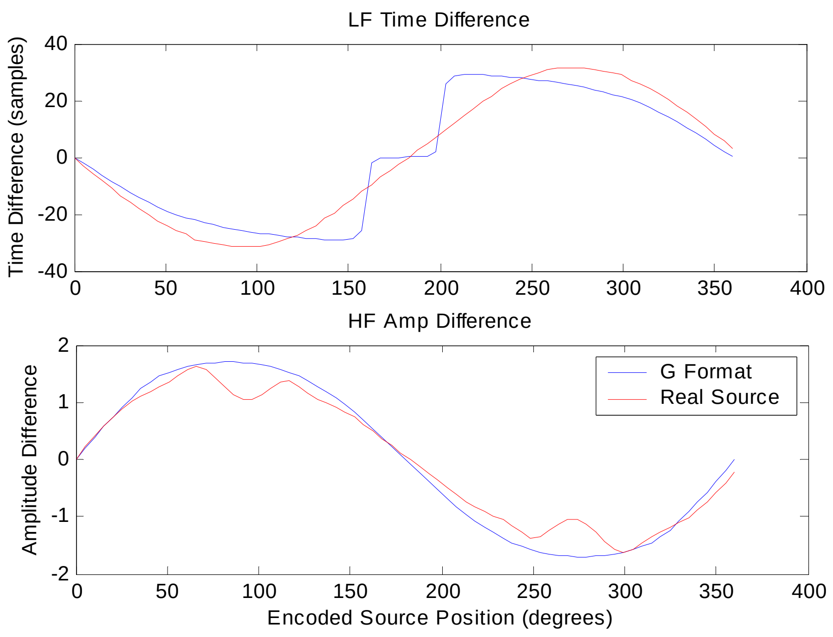


Figure 5.33 A graph showing real sources and high and low frequency decoded sources time and level differences.

This graph shows two interesting points. The low frequency, time difference, graph indicates that the decoded material is not perfect, showing a significant error around the rear of the system's decoded sound field. This is, of course, understandable as there is a speaker 'hole' of 140^0 between the two rear speakers; however, this fact is not apparent from the velocity vector analysis.

The high frequency amplitude differences are a very good fit to the real source's curve, even when a source is to be reproduced around the rear of the listener. The fact that the two vector analysis techniques perform slightly differently is not wholly unexpected, as these two ideas were taken from a number of sources and converted into part of a psychoacoustic meta-theory by Gerzon (1992c).

In order to analyse the robustness of the calculated coefficients, head rotation must be simulated. As the set of HRTFs used for the auralisation and analysis of the Ambisonic decoders are taken using a fixed head, head rotation is achieved by moving the speaker sources around the listener (which is, essentially, the same thing). This more complex relationship between the real and virtual source's localisation cues can then be observed. A well designed decoder will have localisation cues that follow the changing real cues as closely as possible, where as a decoder that does not perform as well will exhibit various artefacts, such as the virtual source moving with the listeners as they rotate their head in any one direction (in the horizontal plane in this example).

Figure 5.34 shows a graphical representation of two sets of decoder coefficients that solve the energy and velocity vector equations (as good a fitness value as possible). It can be clearly seen that the low frequency decoder (that we shall concentrate on here) has different virtual microphone responses for each of the decoders even though the decoders' performance analysis using the velocity vector gives an identical response for each coefficient set. To make a more detailed comparison between these two sets of coefficients we can use the HRTF simulation described above.

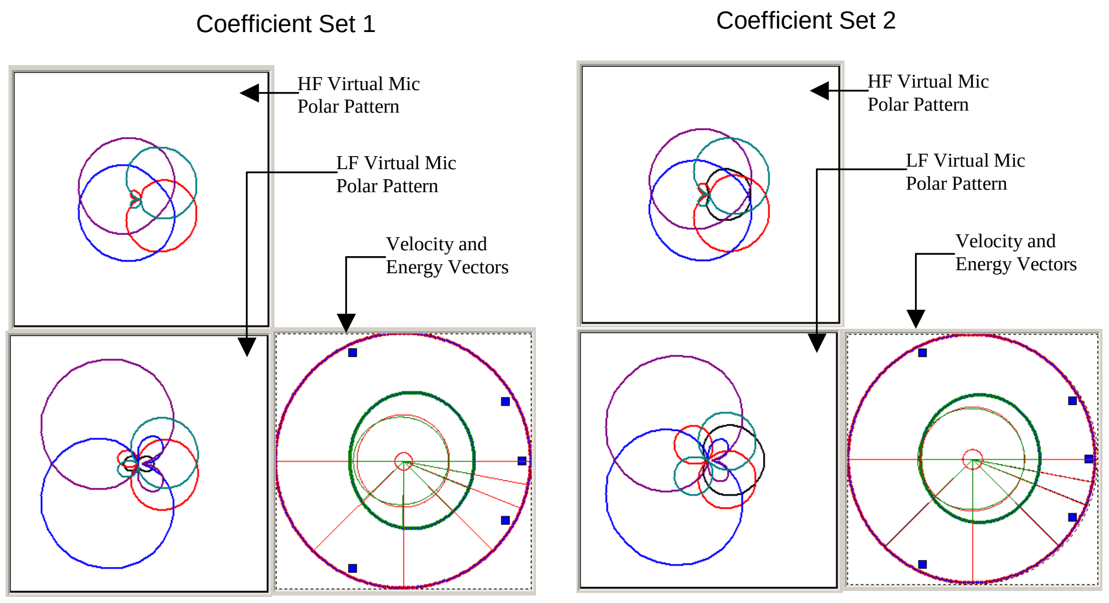

Coefficient Set 1
Coefficient Set 2
HF Virtual Mic Polar Pattern
LF Virtual Mic Polar Pattern
Velocity and Energy Vectors
HF Virtual Mic Polar Pattern
LF Virtual Mic Polar Pattern
Velocity and Energy Vectors

Figure 5.34 Graphical representation of two low/high frequency Ambisonic decoders.

Figure 5.35 shows that coefficient set 2 has a better match of the low frequency time difference parameter, when analysed using the HRTF data, than coefficient set 1. However, this does show up a shortcoming of the energy and velocity vector technique. As mentioned already, a number of solutions can be found that satisfy the energy vector equations, and a number of solutions can be found that satisfy the velocity vector equation. Once a good set of coefficients have been produced it has previously been a case of listening to the resulting decoders and subjectively deciding which one is 'best'.

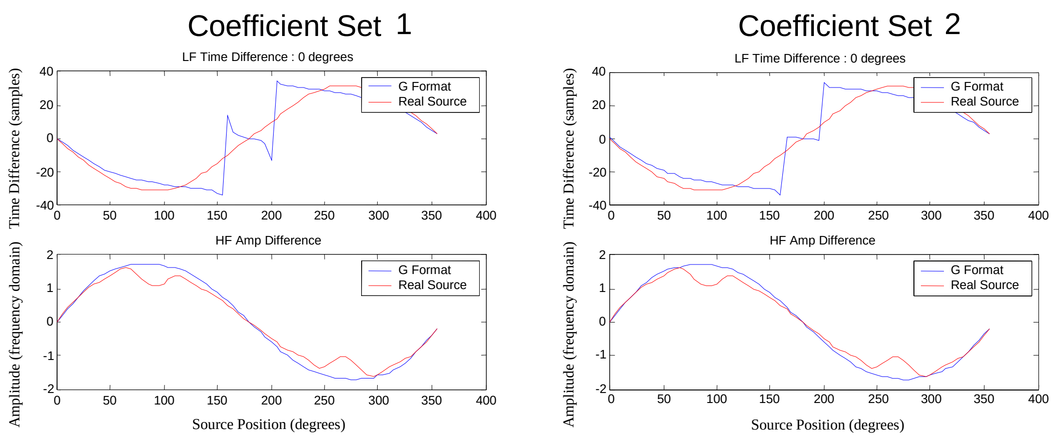


Figure 5.35 HRTF simulation of two sets of decoder.

However, if we continue the HRTF simulation, the effect that head rotation has on the reproduced sound field can be observed (see Figure 5.36). In anechoic circumstances, simulating a change of head orientation and a rotation of all the speaker positions are actually the same thing. So in order to accurately simulate head movement, all the speakers are rotated. This should have the effect of the time and amplitude difference graphs cyclically shifting when compared to Figure 5.35. Any difference in the graphs *apart* from the cyclic shift is in error with what *should* be happening (and what can always be seen in the graphs with regards to an *actual* source). Observing Figure 5.36, it can be seen that head movement introduces errors to the mean time and level differences presented to a listener in anechoic circumstances. The low frequency time difference results are similar in error, but a difference can be clearly seen. Coefficient set 1's low frequency plots stay faithful to a real source's time difference. However, the second set of coefficients does not behave as well as this. If we look at the real and virtual source shown at 0^0 on the graphs (representing where the listener is facing, which will now be an off-centre source due to the rotation of the speakers) the virtual response should follow that of a real source. That is, a source at 0^0 should now have an off-centre response as the speakers have rotated (again, which is the same as head rotation in anechoic circumstances).

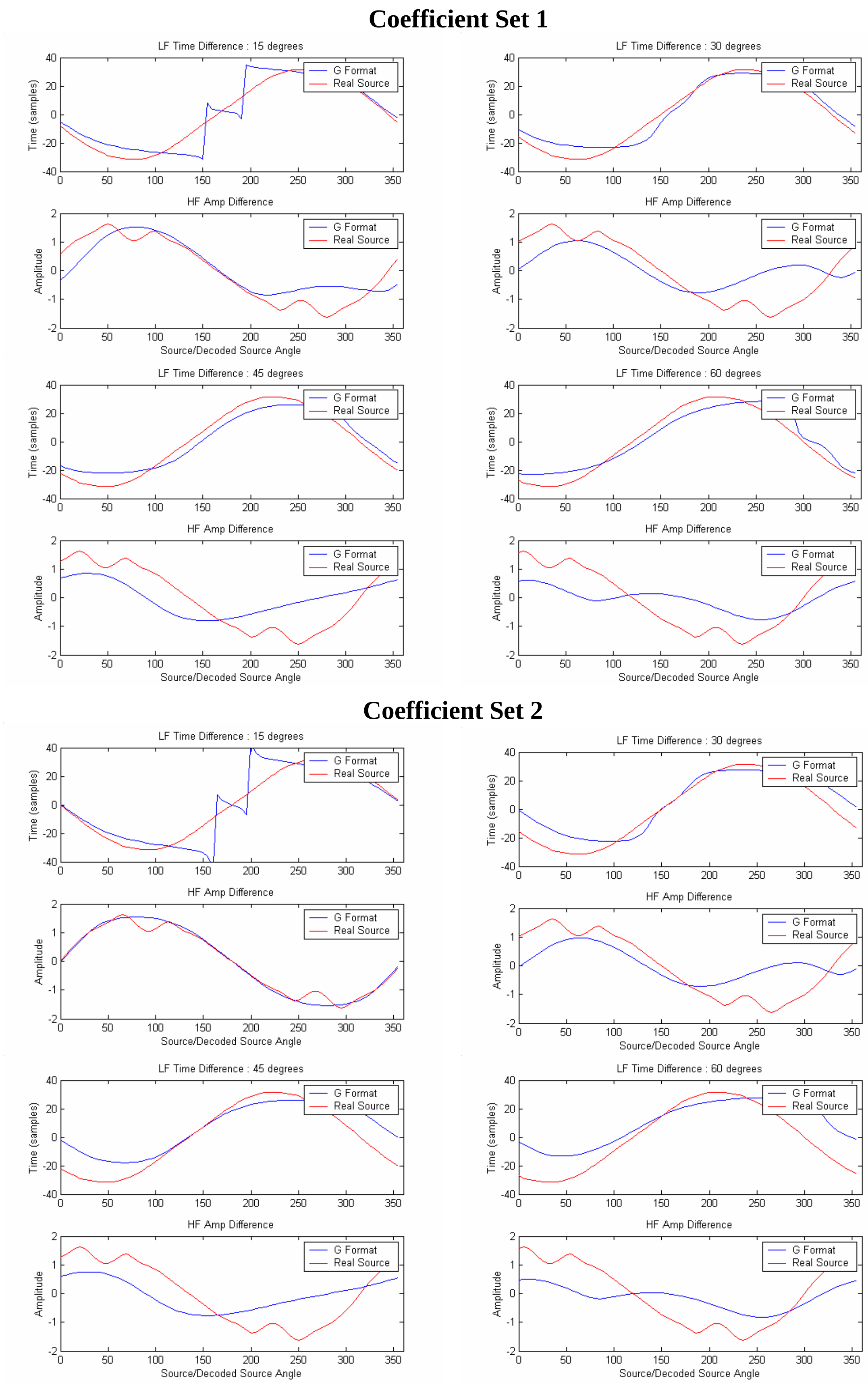


Figure 5.36 **HRTF Simulation of head movement using two sets of decoder coefficients.**

This is not the case for the 2nd set of coefficients and it can be seen that as the head is rotated, the virtual source's time difference stays at approximately 0 samples difference. This means that when the head is rotated, the virtual sound source will track with the listener, potentially making the resulting sound field confusing and unstable.

5.3.6 HRTF Decoding Technique – Low Frequency

The evidence gathered from the HRTF analysis of the decoders' performance under head movement suggests that, as far as the low frequency velocity vector is concerned, more information is needed to design a decoder that is both stable under head rotation and has accurate image localisation. However, as the velocity vector is used as an approximation to the interaural time difference, it is now possible to alter the Tabu search algorithm described in section 5.3.4 to ignore the velocity vector and deal directly with the interaural time difference present for encoded sources around the unit circle. This, on its own, may lead to potential performance increases as the interaural time difference for a listener looking straight ahead can be mapped more accurately using HRTF data, when compared to the velocity vector theory. Also, head rotations can be simulated as shown above, and these results taken into account when evaluating the fitness of a particular decoder.

So, as is immediately apparent, the actual Tabu search algorithm will remain the same (decoder still has same number of coefficients etc.) but the algorithm that supplies the Tabu search with its fitness coefficient must be altered to take advantage of this new research.

$$Fitness = \sum_{m=0}^{12}\left(\sqrt{\sum_{k=0}^{360}\left(\frac{d\phi_{ref}^{k}}{\partial\omega}-\frac{d\phi_{dec}^{k}}{\partial\omega}\right)^{2}}\right)\Bigg/13$$

(5.10)

k = Source angle

ϕ=Average Phase Response (0-700Hz)

ω=Frequency

m=Head Rotation number.

The fitness is now calculated using Equation (5.10) and then combined with the pressure level (volume) fitness given in Equation (5.8) using the root mean square value. Again, the closer this fitness value is to 0, the better the performance of the decoder coefficients. In order to take into account head movement, this equation is evaluated using speaker rotations from 0^0 to 60^0 in 5^0 increments, and then the average fitness is taken.

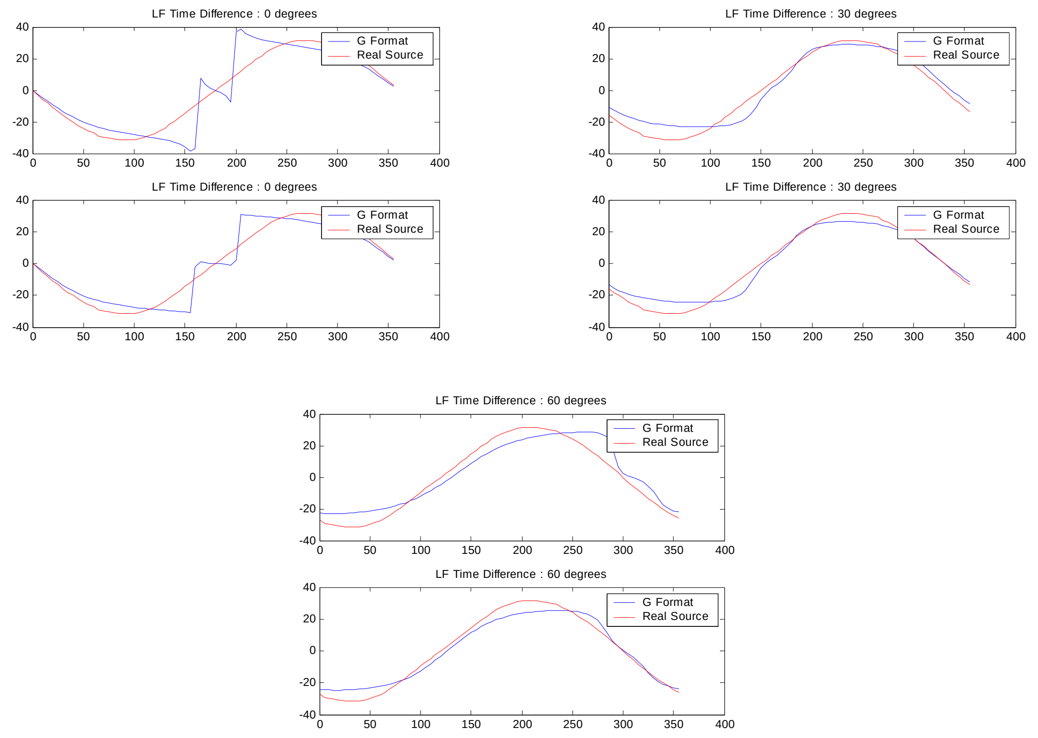


Figure 5.37 **Comparison between best velocity vector (top) and a HRTF set of coefficients (bottom).**

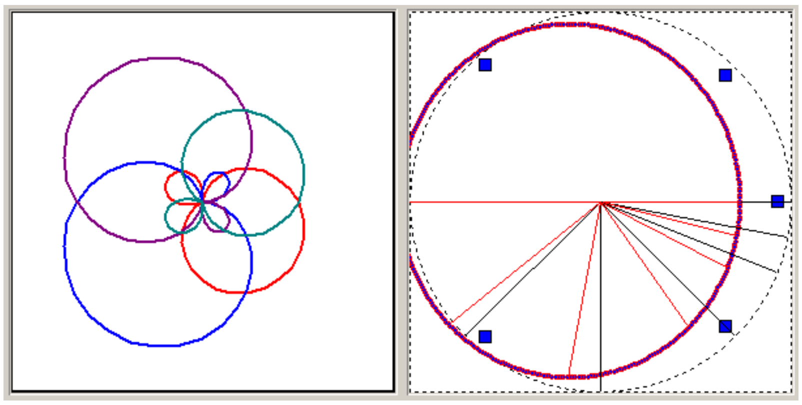


Figure 5.38 **Polar and velocity vector analysis of decoder derived from HRTF data.**

In terms of the low frequency decoders that this technique produces, there is a very high correlation between this HRTF method and the previous velocity vector analysis. That is, a decoder calculated using HRTF data produces a good velocity vector plot as shown in Figure 5.38.

However, it can be seen that, in order to maintain the image stability due to head rotations, a compromise is needed between the accuracy of the decoder's localisation (according to the velocity vector) and its image stability under head rotations. To see if this is actually the case Figure 5.37 shows the HRTF analysis of the best velocity vector decoder (as used in Figure 5.36) and a set of decoder coefficients derived using HRTF data. It can be seen that the resulting plots are almost identical for each reproduced angle and degree of head rotation (0^0,30^0 and 60^0 in this case). The HRTF derived set seems to actually have a better fit than the velocity vector analysis suggests, and a slightly better fit than the original velocity vector decoder (which was found to be the best of several found using the velocity vector technique). So, as the decoder is now calculated taking head rotation into account, every decoder now produced using this technique (as there are, again, multiple solutions) will have an analytical performance similar to that shown in Figure 5.37.

5.3.7 HRTF Decoding Technique – High Frequency

As already stated (and as can be seen in Figure 5.36), the decoder's high frequency response is much more difficult to match to that of a real source, and most decoders derived using the energy vector theory have a response to head rotations very similar to those shown in Figure 5.36. However, as is shown in the listening test later in this chapter, although decoders can be designed using HRTF data directly, taking head rotations into account, this will not *necessarily* result in decoders that perform better under head rotations than when designing a decoder using the energy vector analysis. It decoder designed using velocity and energy vectors can, clearly, still have a good response to head rotations, it is just that it is not due to the Tabu search algorithm striving for this behaviour. However, when utilising velocity/energy vector optimisations, the head rotation parameter can still be used in order to differentiate between decoders' performance as many resulting decoders are possible.

The algorithm used to calculate the fitness parameter for the higher frequency decoder actually needs to be of a slightly different nature than that of the low frequency system. This is due to the fact that after analysing the high frequency lateralisation cues of many optimum decoders (optimum, in that they were optimised using the energy/velocity vector methods, or using purely front facing HRTF optimisation) it was found that, due to the non-uniformity of the speaker layout, high frequency head turning is more catastrophic for the amplitude cue when compared to the low frequency phase cue. If the average fitness were used then the Tabu search would treat optimising the response under head rotation with the same priority as looking straight ahead, possibly resulting in a decoder that performs best when looking 30^0 to the left, for example. It makes more sense to have priority given to the decoder's output when the listener is facing straight ahead, and so a weighting term must be used. The equation used for the localisation fitness parameter is given in Equation (5.11). This resulted in HRTF decoders that performed best when the listener is facing straight ahead, as if the weighting parameter was not used, the Tabu search algorithm would converge on decoders with a poor, analytical, performance (i.e. the fitness function did not *truly* represent the fitness of the decoder as a small increase in fitness when facing off-centre made more of a difference when compared to a centrally facing listener).

$$Fitness = \sqrt{\sum_{k=0}^{360}\left(f(k)_{ref} - f(k)_{dec}\right)^2} \tag{5.11}$$

f=average magnitude response between 700-3000Hz of a real source ($_{ref}$) at k^0 from centre front, and a decoded source ($_{dec}$) located at k^0 from centre front.
k=source angle (in degrees).

5.3.8 Listening Test

5.3.8.1 Introduction

In order to try and quantify any improvement that can be attributed to the optimisation techniques described above, listening tests are needed. Although the main body of this report concentrates on the numerical analysis and optimisation of Ambisonic decoders using the lateralisation parameters and velocity and energy vectors, a number of small listening tests were developed in the hope that others will carry the work further now that a technique for optimising irregular Ambisonic decoders has been made available.

When designing the listening tests, there are two main types of material that can be presented to the listener:

1. Dry, synthetically panned material
2. A pre-recorded real event in a reverberant space.

Each one of these extremes will result in a test that will be more suited to testing for one attribute more than another. As an example, if the recent Project Verdi test is observed (Multi Media Projekt Verdi, 2002), two recordings in a reverberant space were used, with the following attributes tested in the questionnaire:

a. Subjective room size – very big to small
b. Localisation accuracy – very good, precise to bad
c. Ensemble depth – deep to flat
d. Ensemble width – wide to narrow
e. Realism of the spatial reproduction – very good, natural to bad, unnatural
f. Personal preference – very good to bad.

These are typical of the types of spatial attributes tested when listening to pre-recorded material (although others, suggested by Berg & Rumsey, 2001, could be envelopment, presence and naturalness). This type of material is hard to test, in some ways, as it does depend on what you are expecting the tested system to achieve. For example, accurate scene-capture and ‘best sounding' are not necessarily synonymous; ultimately, the personal preference parameter may be of greater importance.

Conversely, the most common form of test carried out on a dry, synthetically panned source is that of simple recognition of the angular placement of that source (again, see Moller *et al.*, 1999, Kahana *et al.*, 1997, Nielsen, 1991 and Orduna *et al.*, 1995 as typical examples). However, evaluating other attributes can often lead to a fuller picture of what a particular system is achieving. Such attributes could include:

g. Source width/focus
h. Source distance
i. Source stability (with respect to head movement).

When it comes to testing a surround sound system, the ideals of the system are easier to decide upon. The best case scenario would be a system which:

- Has small image width/good image focus
- Reproduces distance accurately
- Reproduces sources in a fixed position, regardless of listener orientation.

Also, not mentioned at this point is that the performance of any multi-speaker system in an off-centre position can also be assessed using any/all of the above points.

As far as the optimisations of the Ambisonic decoder are concerned, the direct consequences should be (with regard to an Ambisonically panned, dry source):

- Increased accuracy/matching of encoded source position to perceived source position.
- Increased image stability with respect to head turning.

Other effects of the optimisation may be (again, with regard to an Ambisonically panned, dry source):

- Change in perceived image width/focus.
- Timbral alteration due to differences between low and high frequency decoders

All of the above would also be true when listening to pre-recorded, reverberant material, with potential increase in accuracy and coherency of the lower order, lateralisation cues, resulting in improvements to the higher order spatial properties of the reproduced audio environments:

- Envelopment should be increased, that is, the sense of being in a real place, and not listening to an array of speakers.
- Spaciousness should more closely resemble that of the actual event.
- Depth perception should be more accurate.

To this end, in order to subjectively test these decoders, questions based around these attributes should be designed.

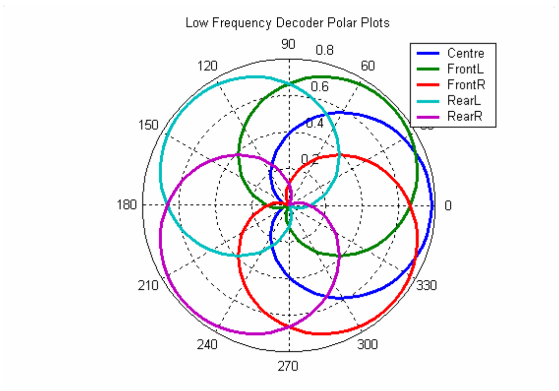
5.3.8.2 Decoders Chosen for Testing

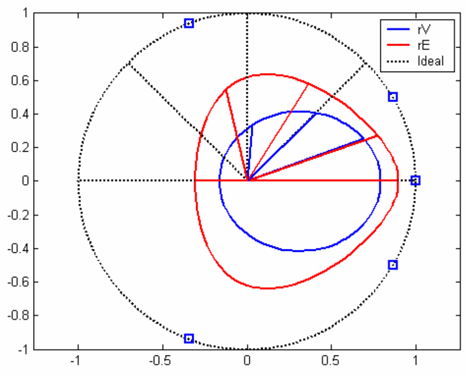
A small sample listening test was carried out to give an insight into which specific decoders worked best, and also to observe any common features with Ambisonic decoders designed for use with an ITU 5 speaker array in order to influence further listening tests to be carried out after this research.

Five decoders were chosen for this test comprising:

- One decoder using the default settings of the commercially available SoundField SP451 Surround Processor (SoundField Ltd., n.d. a).
- Two decoders optimised using the energy and velocity vector.
- Two decoders optimised using HRTF data directly

An analysis of these decoders will now follow, using both the energy and velocity vector and the HRTF decomposition methods described above.





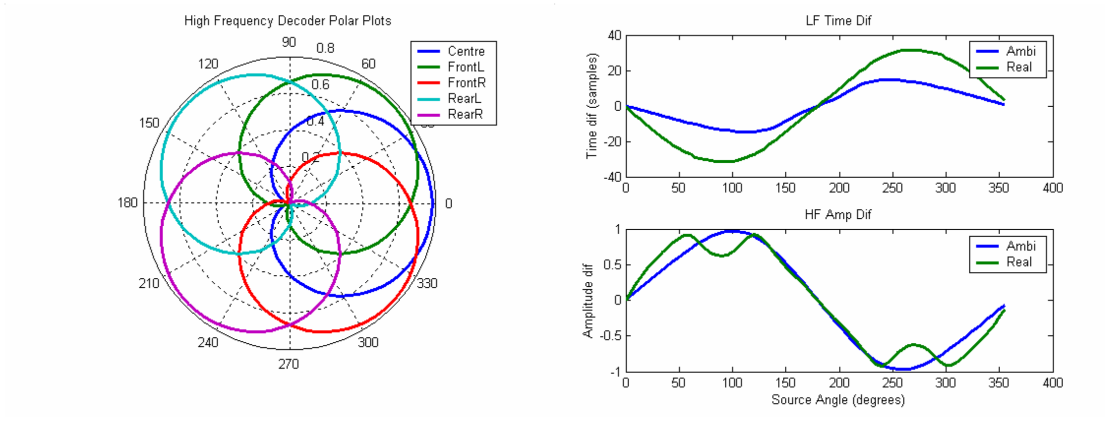
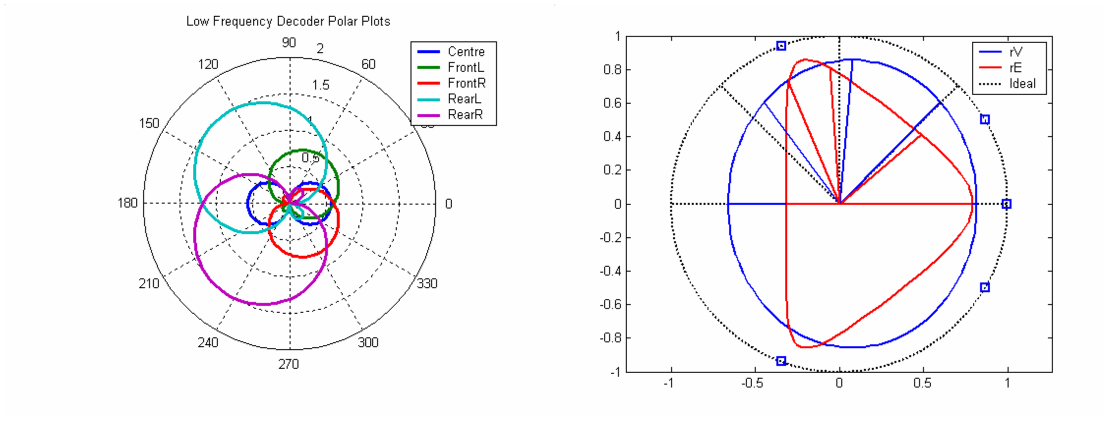


Figure 5.39 Decoder 1 – SP451 Default Settings

Figure 5.39 shows the default settings of the commercially available SP451 Surround Processor unit. This decoder is frequency independent (i.e. both high and low frequency decoders are the same), with all the virtual microphone polar patterns being of type cardioid. This leads to various problems when the decoder is viewed using energy and velocity vectors, with the resultant lengths of the vectors being suboptimal, and all of the source positions being shifted forwards (i.e. a source that should be at 45^0 will be reproduced closer to around 20^0 when decoded). However, when the resulting HRTF analysis is observed, the high frequency amplitude differences are a surprisingly good match to that of an actual source, with the low frequency time difference showing the greatest error.



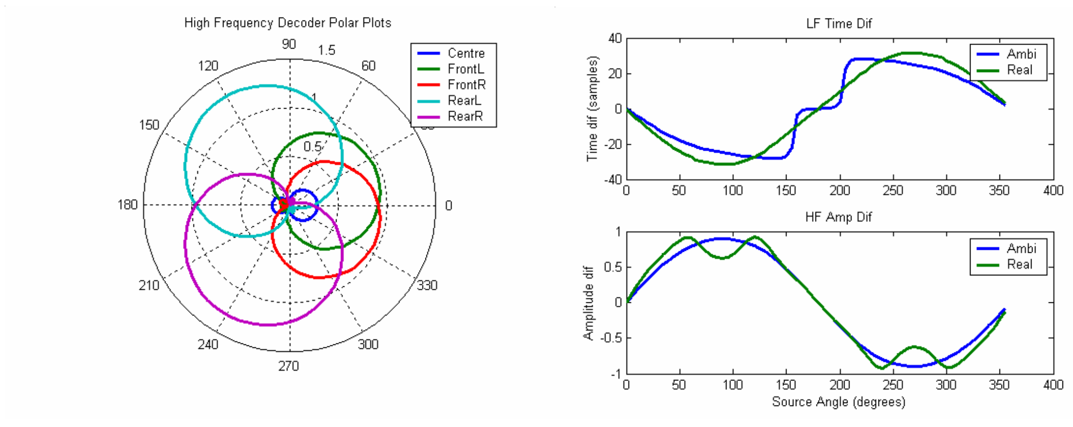


Figure 5.40 Decoder 2 – HRTF Optimised Decoder

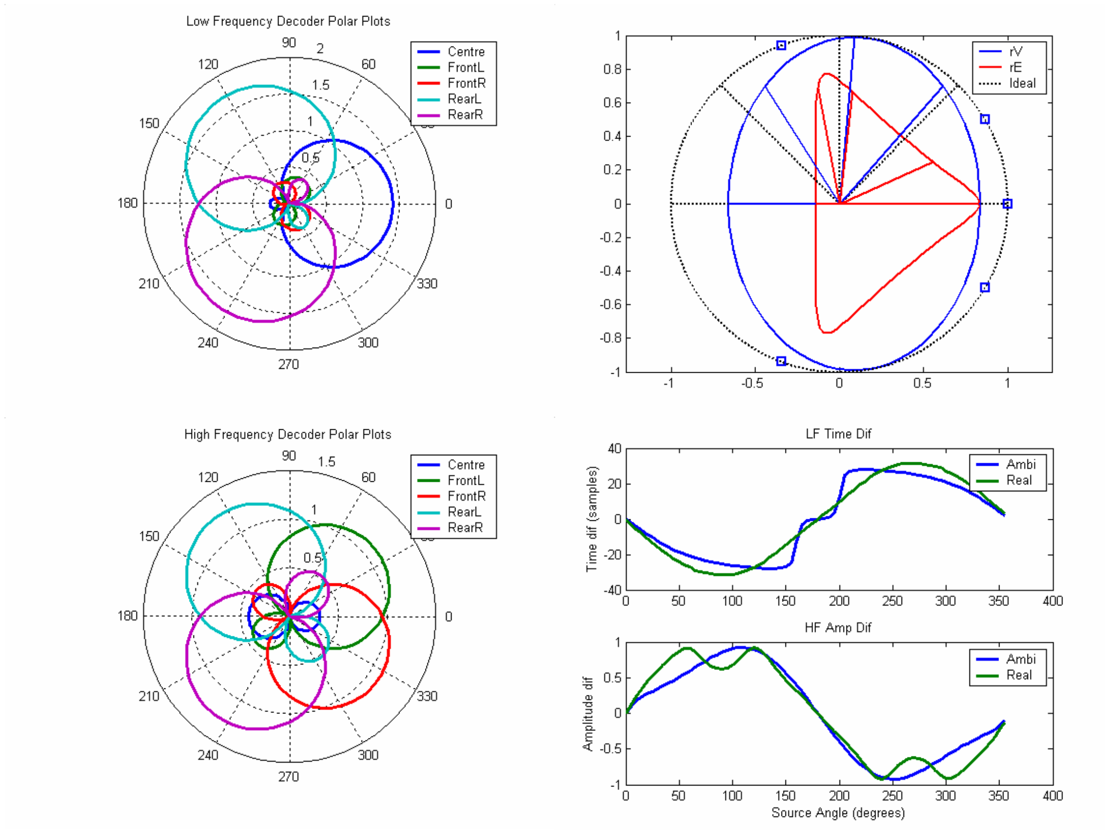
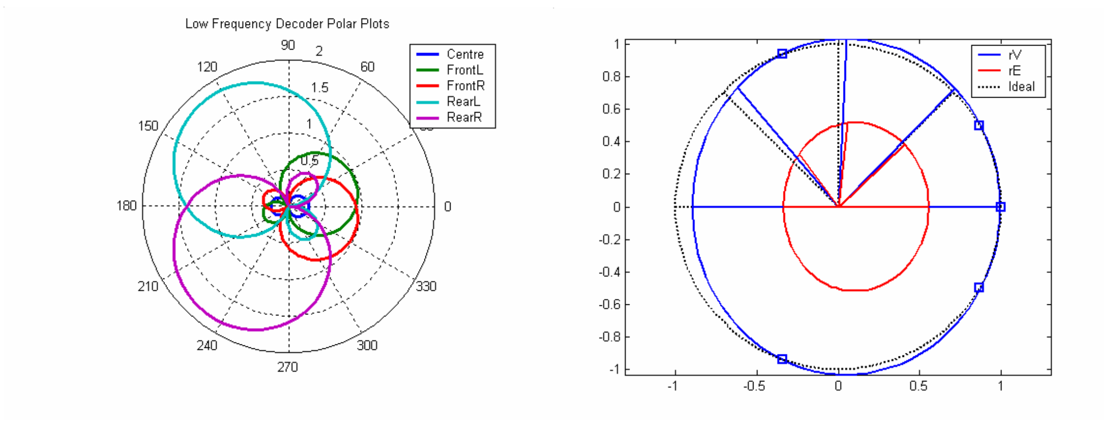


Figure 5.41 Decoder 3 – HRTF Optimised Decoder

Figure 5.40 and Figure 5.41 show two examples of decoders optimised using HRTF data directly. It can be seen that these two decoders have produced similar results when looked at using the HRTF data directly and when using the velocity and energy vector analysis, although the virtual polar patterns for both high and low frequency decoders are quite different. Also, the two types of analysis show good agreement as to the angular distortion introduced by Decoder 3, with frontal sources not producing enough level difference between the ears, and so pushing sources towards the front of the speaker array. Decoder 2 has a much better encoded/decoded source position agreement which is, again, shown in both the HRTF and velocity/energy vector analysis at high frequencies, with very similar performance, again using both forms of analysis, at low frequencies.

Figure 5.42 and Figure 5.43 show the two decoders that were designed using the velocity and energy vector theories. One thing to note, firstly, is that these decoders were optimised using rear speaker positions of +/- 115^0 instead of the usual +/- 110^0. Unfortunately this was not noticed until after the listening test was carried out, but this is why the low frequency velocity vector match is not as good as those shown in Section 5.3.4. Again, both of these decoders have quite different low frequency virtual microphone polar responses, but have near identical velocity vector responses. However, if the HRTF data is looked at, it can be seen that Decoder 4's low frequency phase differences can be seen to have significant errors around the rear of decoder's response, showing a 'flipping' of the image cues at source positions of 160^0 and 200^0. The high frequency decodes were designed using slightly different criterion, with the angular accuracy of Decoder 4's energy vector reproduced angle being given a slightly smaller weighting, resulting in a higher error in the reproduction angle for the rear of the decoder, but with the localisation quality (vector length) benefiting from this approach.



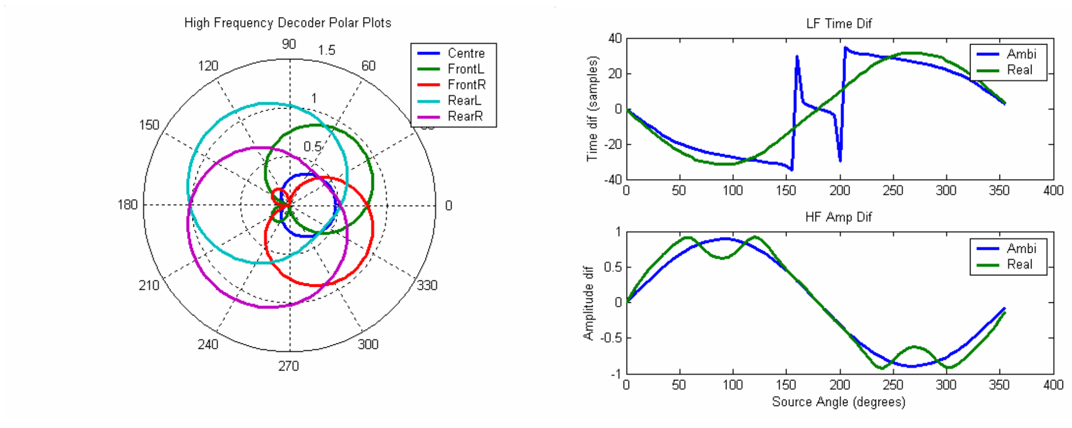


Figure 5.42 Decoder 4 – Velocity and Energy Vector Optimised Decoder

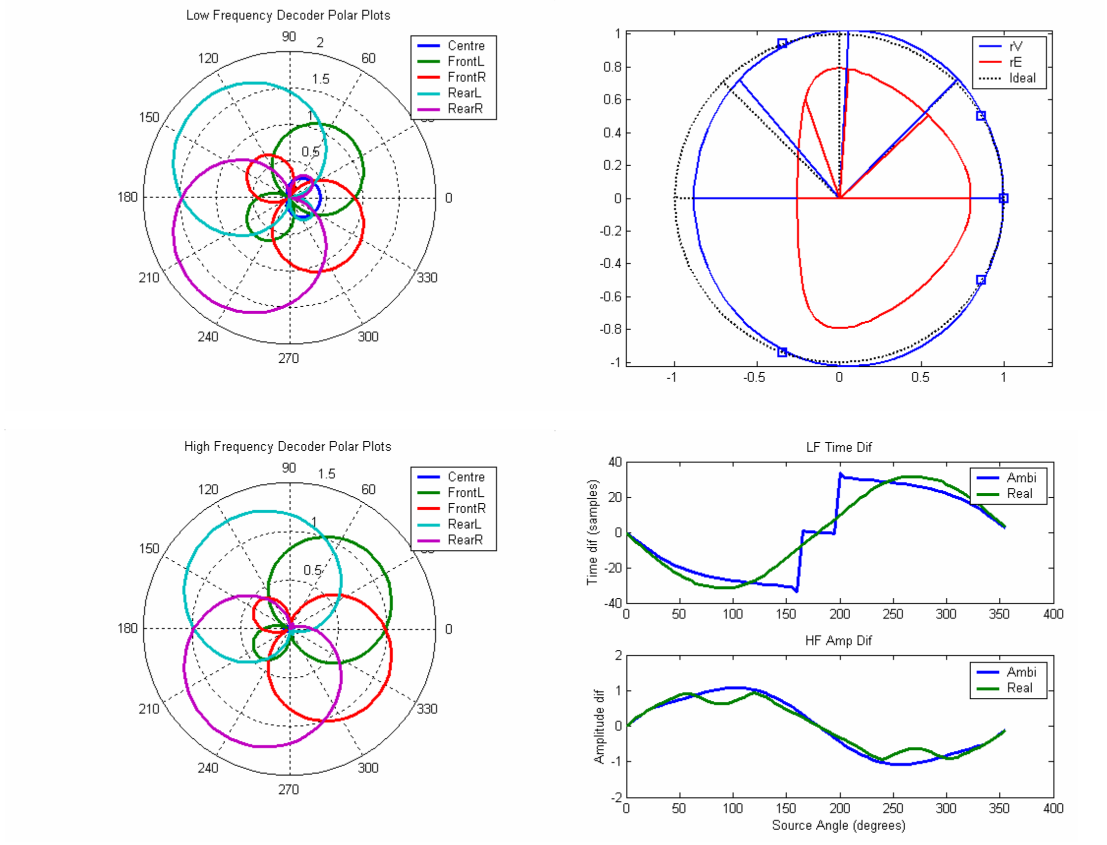


Figure 5.43 Decoder 5 - Velocity and Energy Vector Optimised Decoder

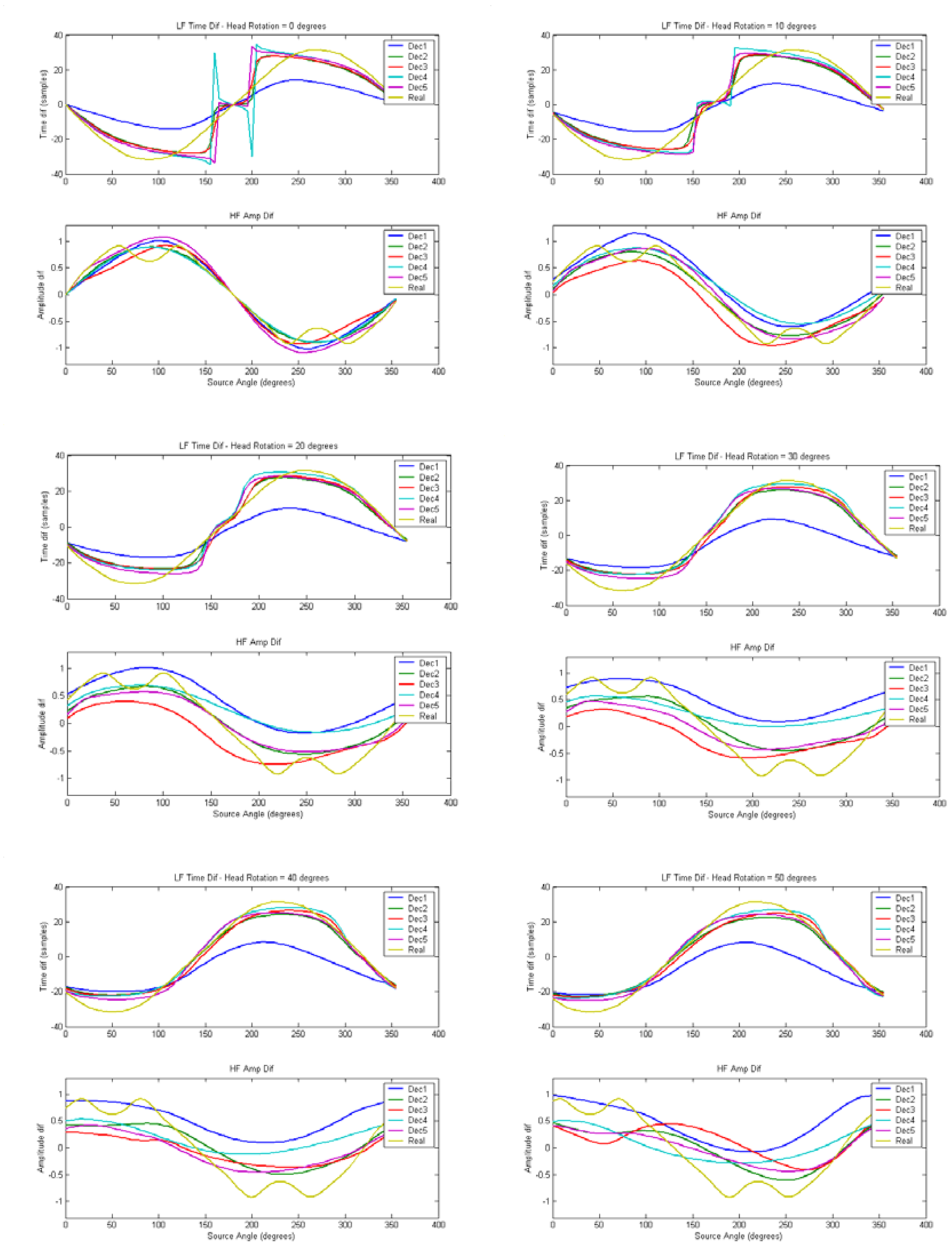


Figure 5.44 Comparison of low frequency phase and high frequency amplitude differences between the ears of a centrally seated listener using the 5 Ambisonic decoders detailed above.

Although the HRTF analysis of the various decoders has been shown, no mention has yet been made of the performance of each decoder, numerically, with respect to head turning. Figure 5.44 shows each decoder's performance, when compared to a real source, with respect to a listener turning their head from 0^0 (facing straight ahead) to 50^0. It can clearly be seen that all optimised decoders perform in a very similar manner at low frequencies, with even the unoptimised decoder performing in a coherently incorrect fashion (i.e. it does not seem to exhibit the image tracking of a frontal source, for example, as described in section 5.3.6). However, as it is to be expected, the high frequency decoders do not perform as well. Figure 5.45 shows the lateralisation cue errors as absolute error values, with Figure 5.46 showing the average error value for each decoder with respect to head turning.

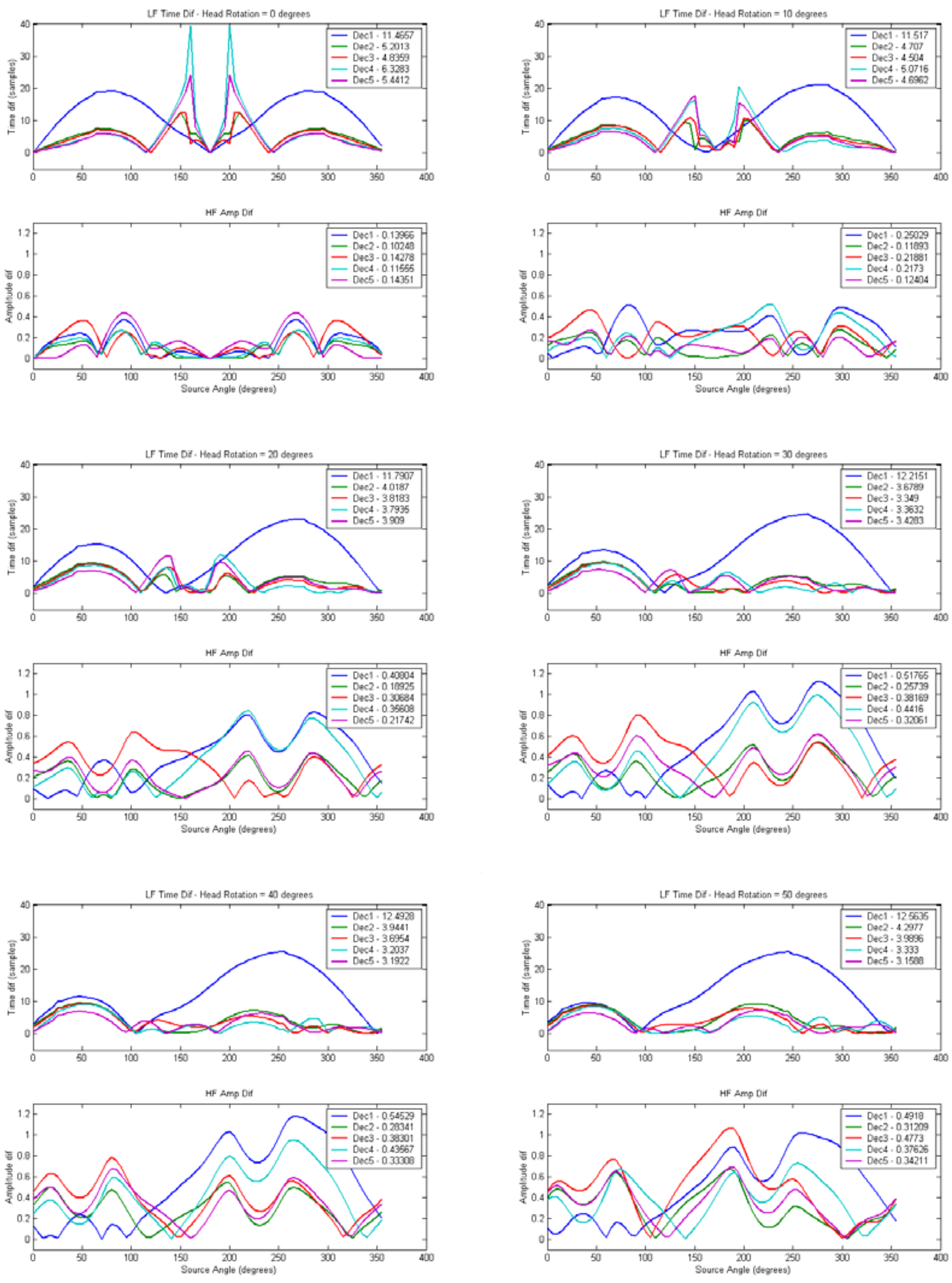


Figure 5.45 **Graphs showing absolute error of a decoder's output (phase and level differences between the ears of a centrally seated listener) compared to a real source, with respect to head movement.**

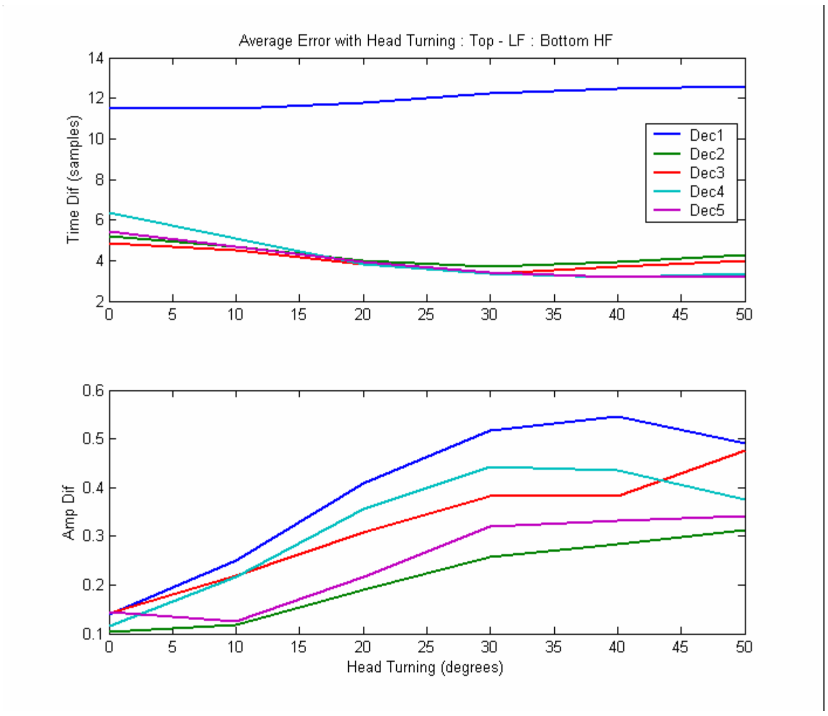


Figure 5.46 **Graph Showing the Average Time and Amplitude Difference Error with Respect to A Centrally Seated Listener's Head Orientation.**

Figure 5.46 shows, in a very simplified manner, how each decoder will perform. Using this graph as an indicator for overall performance, it can be seen that, as already mentioned, all of the decoders perform almost equally as well with respect to low frequency phase cues, with Decoder 1 having, by far the worst error, but, as already mentioned, an error that stays reasonably consistent with head turning. However, it is the high frequency plots that give more insight into the performance of any decoder, as it is the high frequency decoder that is most difficult to optimise, using either energy vector of HRTF techniques. Performing best, here, is Decoder 2, which was designed with head turning as a parameter (although, only up to 30 degrees). However, the decoder with the next best high frequency error weighting is Decoder 5 which is a decoder designed using the energy and velocity vector principles. It must also be noted that, although the decoders all seem to perform similarly (under numerical analysis), looking at the low frequency errors it can be seen that, again, decoder 5 performs very well (best, in fact), but decoder 2 at low frequencies is one of the worst performing decoders (ignoring Decoder 1). Although there are four optimised decoders tested, each low frequency and high frequency decoder was designed separately. No criteria has yet been set for deciding which low frequency decoders will complement particular high frequency decoders and so the decoders have been paired randomly (although always grouped with a decoder that was optimised in the same way, that is, using either HRTF or velocity/energy vector methods).

5.3.8.3 Listening Test Methodology

For the actual listening test, two separate testing methods were chosen:

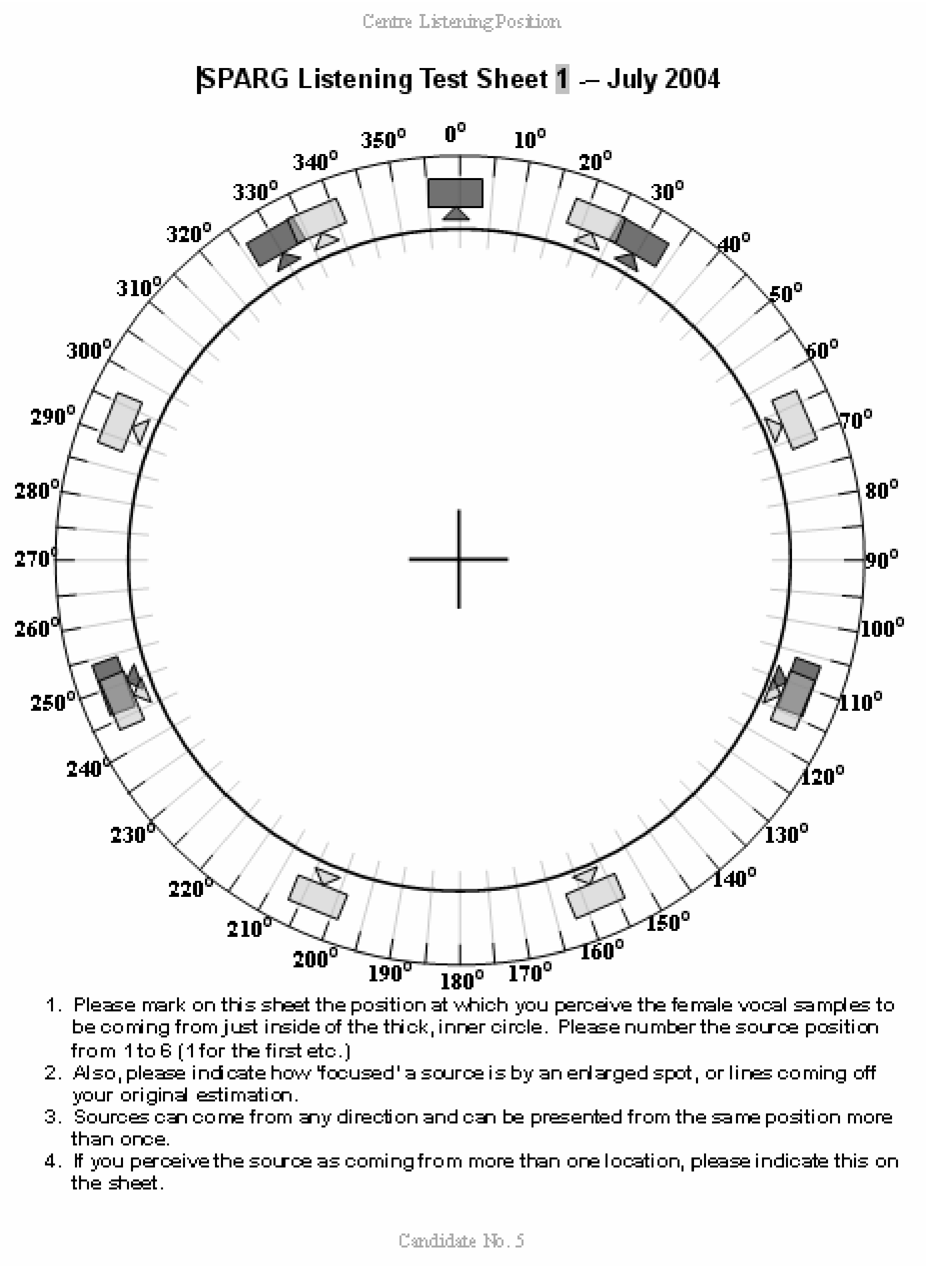
- A listening test similar to that described in section 5.2, measuring the accuracy of panned, mono sources in the decoded sound field.
- A test where users give a preference as to which decoder performs best when auditioning reverberant, recorded material.

These two styles of testing are not designed to be all-encompassing, but have produced interesting points for use in further testing methodologies.

Two sources were chosen for the listening tests to be carried out. The source that was to be synthetically panned was dry, female speech which is often used in such tests (for example, see Martin *et al.*, 2001, Kahana *et al.*, 1997, Moller *et al.*, 1999 and Neilsen, 1992) due to its wide frequency range, and reasonably un-fatiguing sound (especially when compared to band-limited noise and other such sources). For the test of a real recording where decoder preference was to be given by a 60 second excerpt from a recording made by the company, Serendipity (2000), of Rick Wakeman playing the piano in Lincoln Cathedral. It is a very reverberant recording made by a company that has had significant experience with the SoundField Microphone, particularly in the effective placing of the microphone (something that can often be overlooked when choosing recorded material).

For this small test, three listeners were used. All three were experienced listeners that had taken part in multi-channel sound system listening tests before. The first test had sources presented to them, six source positions per decoder. The source positions were identical for each decoder, but played in a pseudo-random order. The listeners were asked to indicate in which direction they thought the source was coming from and to give an indication of source width. This was to be recorded on the sheet shown in Figure 5.47

which showed the layout of speakers in the University's Multi-Channel Research Lab. In addition, to aid in the recording of source position, each speaker in the lab had a label fixed on it with its angular position relative to straight ahead. They were asked to draw the size of the source, as this method has proved to be more intuitive in these situations (Mason *et al.*, 2000).


Centre Listening Position

SPARG Listening Test Sheet 1 – July 2004

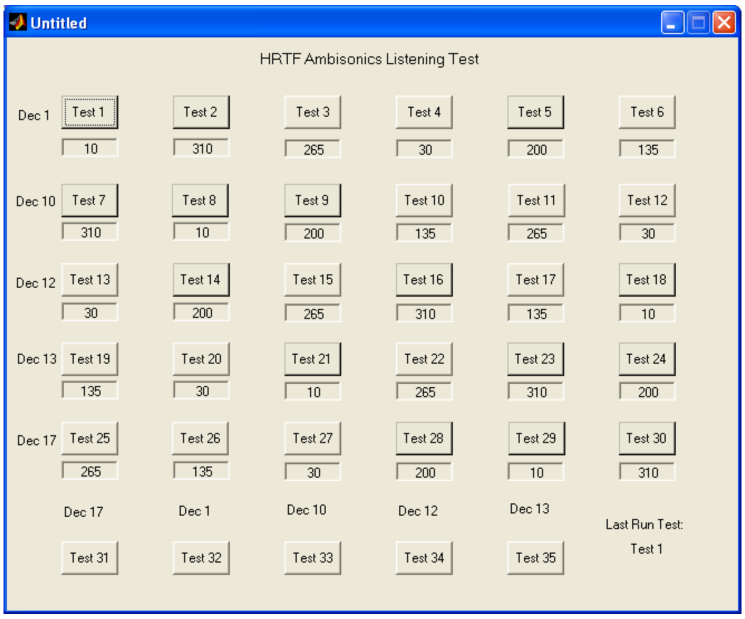
0°
10°
20°
30°
40°
50°
60°
70°
80°
90°
100°
110°
120°
130°
140°
150°
160°
170°
180°
190°
200°
210°
220°
230°
240°
250°
260°
270°
280°
290°
300°
310°
320°
330°
340°
350°

1. Please mark on this sheet the position at which you perceive the female vocal samples to be coming from just inside of the thick, inner circle. Please number the source position from 1 to 6 (1 for the first etc.)
2. Also, please indicate how 'focused' a source is by an enlarged spot, or lines coming off your original estimation.
3. Sources can come from any direction and can be presented from the same position more than once.
4. If you perceive the source as coming from more than one location, please indicate this on the sheet.

Candidate No. 5

Figure 5.47 Sheet given to listening test candidates to indicate direction and size of sound source.

The user interface for controlling the listening test was constructed in Matlab, which called Simulink models that encoded and decoded the mono sources in real-time, taking in a direction parameter that had been pre-entered. A screen shot of this user interface is shown in Figure 5.48.



HRTF Ambisonics Listening Test

Dec 1	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
	10	310	265	30	200	135
Dec 10	Test 7	Test 8	Test 9	Test 10	Test 11	Test 12
	310	10	200	135	265	30
Dec 12	Test 13	Test 14	Test 15	Test 16	Test 17	Test 18
	30	200	265	310	135	10
Dec 13	Test 19	Test 20	Test 21	Test 22	Test 23	Test 24
	135	30	10	265	310	200
Dec 17	Test 25	Test 26	Test 27	Test 28	Test 29	Test 30
	265	135	30	200	10	310
	Dec 17	Dec 1	Dec 10	Dec 12	Dec 13	Last Run Test:
	Test 31	Test 32	Test 33	Test 34	Test 35	Test 1

**Figure 5.48	Screenshot of Matlab Listening Test GUI.**

5.3.8.4 Listening Test Results

The listening test results showed reasonably subtle differences between the different decoders when tested using the synthetically panned source, and much more obvious differences when listening to a more complex, recorded, sound field.

Figure 5.49 shows the results for the three listeners. The square data points represent the recorded source position with the error bars, above and below these positions showing the recorded source size for each decoder. It is difficult to analyse these graphs directly, but it can be seen that all of the decoders seem to perform reasonably well in this test with no image flipping becoming apparent, although two sources were recorded as coming from more than one location, subject 1 – decoder 4 and subject 3 – decoder 1. Interestingly these were both at source position 225^0, which is the area where the decoders will all perform at their worst (i.e. at the rear of the sound field).

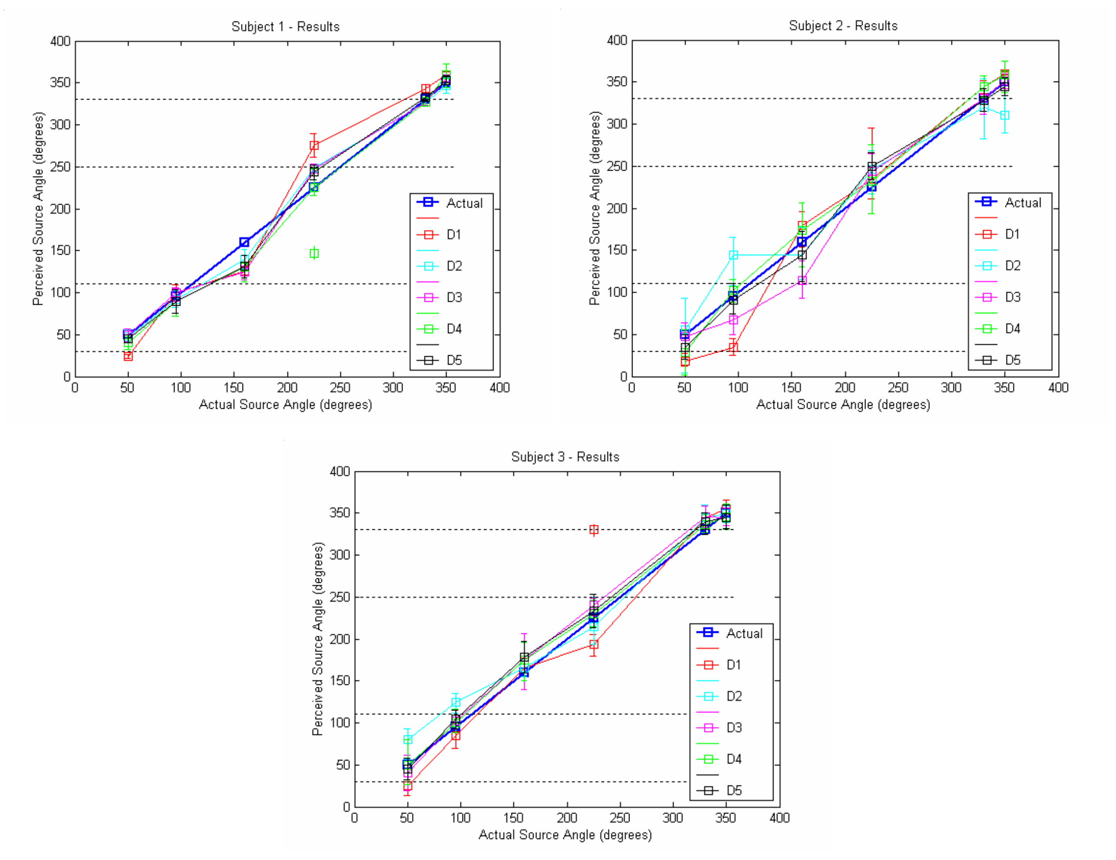


Figure 5.49 **Graphs showing the results of the panned source part of the listening test for each subject. 'Actual' shows the correct position, D1 – D5 represent decoders 1 – 5.**

In order to compare these results more equally, the average absolute angle error and image size can be seen for each subject in Figure 5.50. As is to be expected, the image source's graphical depiction of size is different for each subject (Mason *et al.*, 2000), with subject one generally recording smaller image sizes than subjects 2 & 3. It would be reasonable to insert actual source positions in order to record some form of 'calibration' size source for each listener, but this was not attempted in this small test. Another obvious result is that decoder one seems to perform worst, subjectively, according to each subject (i.e. high mean error value). This was an expected result. The other results, however, are slightly more varied from listener to listener. It was proposed in section 5.3.8.2 that decoders 5 and 2 would be expected to perform best, taking into account head turning and the average localisation error this would produce. However, only subject 1 seemed to agree with this statement in its entirety. Decoder 5 did perform consistently well throughout this phase of the test, but decoder 2 performed less favourably when the results of subjects 2 and 3 are observed.

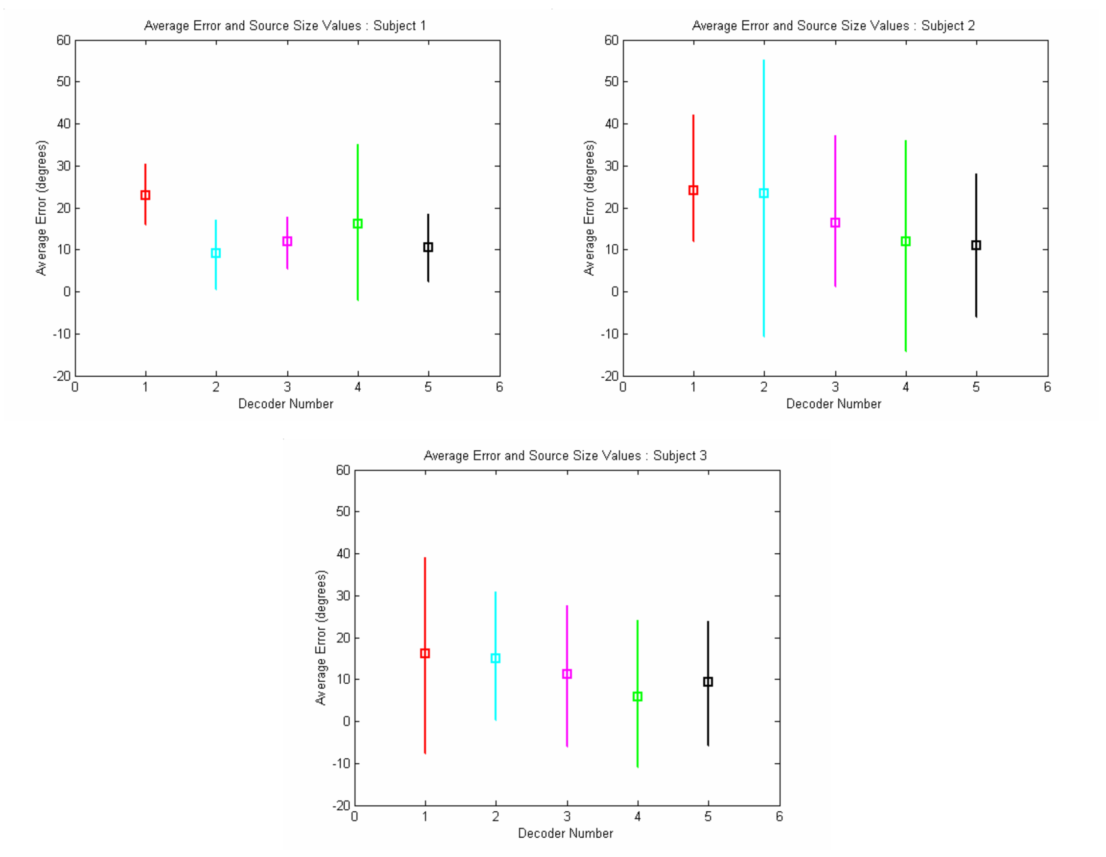


Figure 5.50 Graph showing mean absolute perceived localisation error with mean source size, against decoder number.

There are a number of potential reasons for this:

- Subject 1 was the most experienced listener in this test, and may give the most correct, or predictable results.
- Decoder 5 is located at the end of the test, and the subjects may be changing the way they are grading the results (or learning how to interpret them better) as the test continues. This may be corroborated by the general downwards slope that subjects 2 and 3 show in their average error results.
- The low and high frequency decoders interact in some more complex, non-linear way than has been simulated in the previous analysis of the decoders (i.e. the low and high frequency decoders should not be designed and analysed in isolation).

Figure 5.51 shows the average absolute error and image size for each decoder. It must be noted that, as the image size for each subject has not been normalised, the image size ratios of subject 1 (from decoder to decoder) will have less of an effect than that of subjects 2 and 3. However, the average absolute localisation will not be affected.

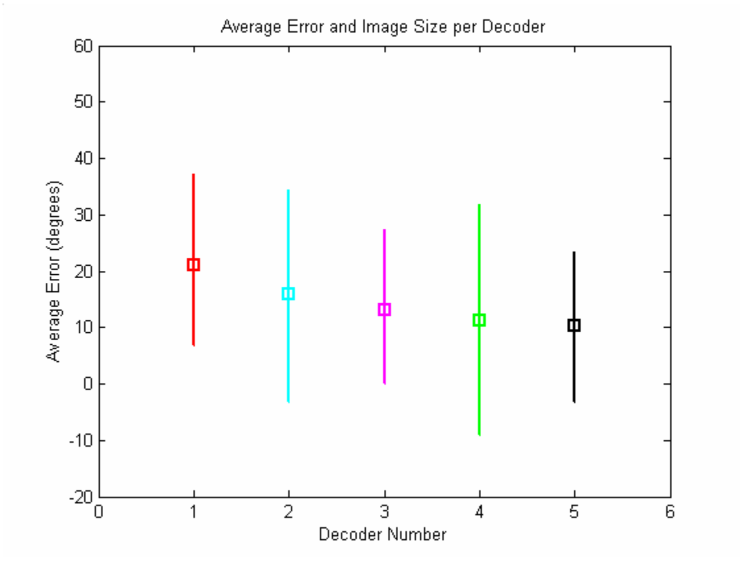


Figure 5.51 **Graph showing the mean, absolute, localisation error per decoder taking all three subjects into account.**

Figure 5.51 shows that, overall it is decoder 5 that seems to perform best in this test, with the downwards slope, starting with decoder 1, being clearly evident in this figure. Also evident is the already mentioned, relatively equal performance of all of the optimised decoders, with an average error between 10^0 and 16^0 compared to decoder 1's average error of 21^0.

Other non-recorded observations were also evident from this test, and are listed below:

- Head movement helped greatly in the localisation of sources in this experiment, and were used extensively by each listener.
- It was noted that although front and side sources were generally very stable (an impressive result by itself, when compared to amplitude panned material or the observations of Craven's higher order decoder (Craven, 2003)), rear images only performed correctly when facing forwards. That is, when the subject turned to face the source, the two rear speakers were perceivable as sources. In these cases all subjects recorded the position facing forwards.
- Front and side images were generally perceived at the same distance as the speakers, whereas rear images were perceived

much closer to the head, almost on a line joining the two rear speakers of the ITU speaker array.

The rear image problems are not wholly unexpected as it can be seen that rear images due to head turning and analysis using the velocity/energy vector methods all point to rear images performing less well. However, the fact that rear images can be formed at all, with a speaker hole of 140^0, is still an impressive result.

The 2nd part of the listening test was the auditioning of a 60 second except of a piano recording made in Lincoln Cathedral. Each listener heard each decoder's representation of this piece once and was then invited to call out which versions they wished to hear again. This was continued until a preference was given as to which decoder they thought performed best. The results of this test were as follows:

Preference		Subject 1	Subject 2	Subject 3
Best	1st	3	3	3
	2nd	5	2	5
	3rd	2	5	4
	4th	4	4	2
Worst	5th	1	1	1

Table 5.1 Table showing decoder preference when listening to a reverberant, pre-recorded piece of music.

The results showed a clear trend, showing that decoder 1 was by far the worst of the five decoders, but with decoder 3 clearly being preferred by all three listeners. This decoder, although not performing as well under head-turning analysis, is the only optimised decoder to have significant shifting of sources towards the front, when looking at Figure 5.41, as shown in both the energy vector and HRTF analysis at high frequencies. This is not the same as just using the forward dominance control as decoder 3 maintains the overall perceived volume equally from all directions. This, therefore, could be a more subjective, artistic artefact of this decoder, although comments from the subjects did indicate some of the reasons for choosing this decoder:

- Subjects 1 & 2 commented that decoders 5 & 2 (which they rated 2nd and 3rd, and 3rd and 2nd respectively) were very similar in performance, both with a slightly ‘oppressive’ sweet spot. This, interestingly, disappeared when auditioned off-centre. Decoder 3 did not suffer from this.
- Subject 1 mentioned that decoder 4 had a very wide, more diffuse image.
- All agreed that decoder 1 was very front heavy, with an obvious centre speaker, and 2 subjects mentioned that it was almost ‘in-head’ at the sweet spot, when compared to the other decoders.
- Subject 1 commented that the Piano, when reproduced using decoder 3, had a very ‘tangible’ quality to it.

5.3.8.5 Listening Test Conclusions

The listening test, although only being presented to a very small number of subjects, was a useful exercise, bringing to light a number of attributes that should be researched further. The most obvious result was that the un-optimised decoder, based on the standard settings of the commercially available B-Format decoder, clearly performed less-well in both of the tests. This shows that both optimisation methods do improve the performance of Ambisonic decoders for a five speaker irregular array. Also, the performance of decoder 5 in the first stage of the listening test (panned source) was also expected, although the differences between the decoders, overall, was more subtle than expected, and a much larger test base would be needed to gain more statistically significant results. However, the fact that the extremes of performance were shown in this small test is a very encouraging result. If this part of the test were to be carried out again a number of changes would be made to try and remove any bias from the results:

- The order of presentation of the test decoders would be randomised. This may eliminate the general downward sloping of the average localisation results observed in subjects 2 and 3.
- The test would be carried out over more than one day, testing each subject at least twice to try and measure what kind of variations each one was likely to produce.

- More source locations would be used so as to map more accurately the performance of each decoder.
- Actual sources would be played at random, so that a 'calibration' source width is available to judge better the width parameter of subject's results.
- A distinction could be made between source stability and image location by running two separate tests (and allowing separate analyses on the results):
    1. Where the subject is asked to face forwards at all times (knowing they will move their head a little, still).
    2. Where the subject is asked to face each source before recording its position.

Interestingly, the decoder that was unanimously voted as the 'best' decoder when listening to pre-recorded material was an unexpected result (however, the decoder perceived as 'worst' was not) with the middle group of decoders needing a larger base of subjects in order to gather a statistically significant result. Although this was a very simple test, with only one parameter, it did, indirectly, reveal some valuable insight into the performance of the decoders:

- Most listeners are often surprised by the amount of variation that can be achieved just by altering the decoder, with spaciousness and envelopment being altered massively (especially when compared to decoder 1).
- The sweet-spot problems with two of the four optimised decoders were particularly interesting, especially as these were, analytically, the best performing decoders. This suggests that over-optimising for a single position may, in fact, be detrimental to the performance of a decoder.
- The best sounding decoder may not be the one that is, necessarily, the most accurate.

Testing the performance of a decoder using pre-recorded material is far more difficult to grade when compared to the first test. A number of different recordings should be used and tests where the recording situation can be described by the listener and compared against later (i.e. actual source positions, size of room etc.) could be used to try to neutralise the artistic aspect of the decoder's performance, if necessary.

5.4 The Optimisation of Binaural and Transaural Surround Sound Systems.

5.4.1 Introduction

Both the Binaural and Transaural reproduction techniques are based upon HRTF technology and, for this reason, can be optimised using a similar approach. One of the main problems with synthesised (and recorded) Binaural material is that the reproduction is normally perceived as filtered. That is, the listener will not perceive the pinna filtering (and normally the microphone and headphone filtering too) present in the recording as transparent. Possible reasons for this could be that the pinna filtering on the recording does not match the listener's, or because no head tracking is used: minute head movements can not be utilised to help lateralise the sound source and so the frequency response heard is assumed to be that of the source itself by the ear/brain system. A similar effect is experienced with the use of crosstalk cancellation filters. If a 2 x 2 set of impulse responses are inverted so as to create a pair of crosstalk cancellation filters, then the frequency response of these filters *will* be perceived, both on and off-axis, even though the theory states that this response is actually compensating for a pinna filtering response. The most logical method of correcting these artefacts is to use inverse filtering techniques.

5.4.2 Inverse Filtering

Inverse filtering (which has already been touched upon in Chapter 3) is a subject that is very simple in principle, but takes a little more care and attention in practice. Inverse filtering is the creation of a filter whose response will completely equalise the response of the original signal. The general case is that of a filter that is created to force the response of a signal to that of a target response and is analogous to re-arranging an equation where the answer is already known, where the value of a variable (in this case, a filter) needs to be found. The time domain representation of this problem is given in Equation (5.12).

$$a(n) \otimes h(n) = u(n)$$
$$h(n) = \frac{u(n)}{a(n)}$$

(5.12)

where: a(n) = original response.
u(n) = target response.
h(n) = inverse filter (to be found).

In Equation (5.12) ⊗ represents polynomial multiplication (convolution) and the division represents polynomial division (deconvolution). A much more efficient approach to this problem is to process all of the data in the frequency domain using the Fast Fourier Transform algorithm. This then transforms the polynomial arithmetic into a much quicker point for point arithmetic (that is, the first value of ‘u’ is divided by the first value of ‘a’ and so on). These frequency domain equations are shown in Equation (5.13).

$$a(\omega) \times h(\omega) = u(\omega)$$
$$h(\omega) = \frac{u(\omega)}{a(\omega)}$$

(5.13)

where: ω = angular frequency.

If we were to take a head related transfer function and find the inverse filter in this way the filter shown in Figure 5.52 will be produced. There are a number of artefacts that can be observed, but first it should be noted that the magnitude response of the inverse filter already appears to be just that, the inverse response (mirror image about the 0 dB mark), as given by the equations above (an inverse filter can be thought of as inverting the magnitude and negating the phase as described in Gardner & Martin (1994)).

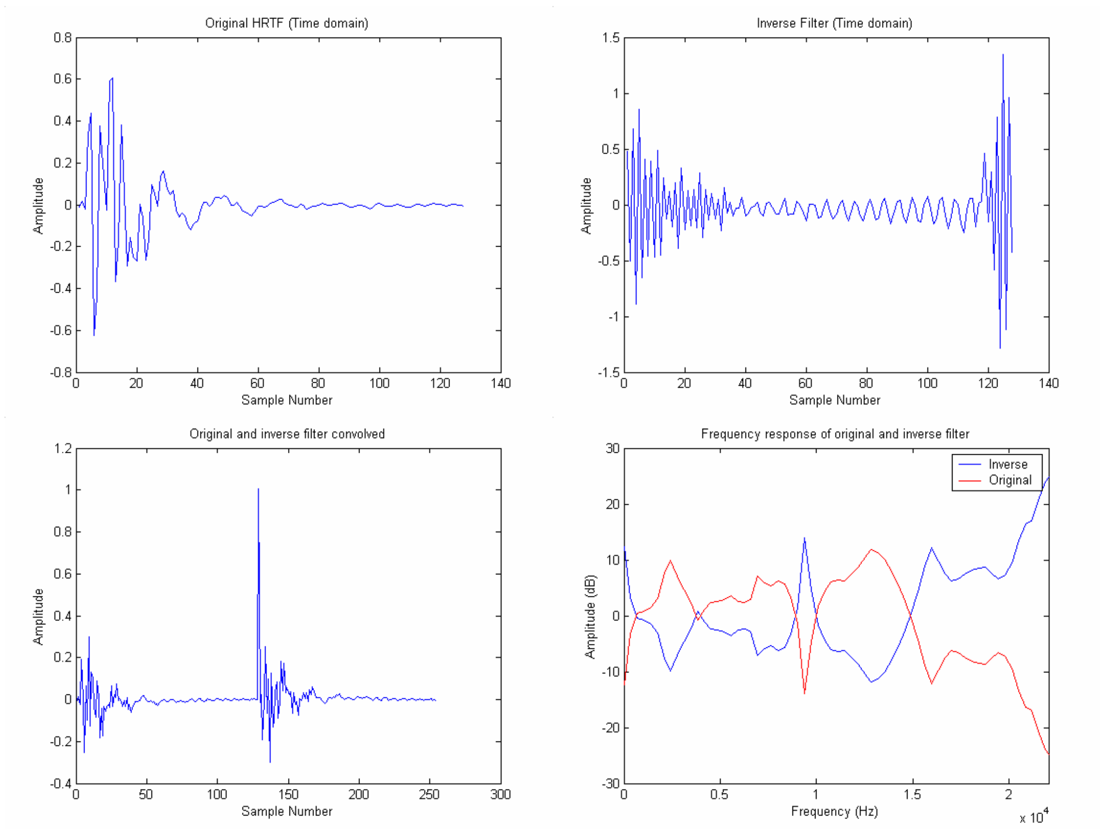


Figure 5.52 Inverse filtering using the equation shown in Equation (5.13)

Unwanted audio artefacts can be clearly seen in the time domain representation of the original and inverse signals convolved together (theoretically they should produce a perfect unit pulse if the inversion has been carried out successfully). Also, the inverse filter does not look complete in that it does not have a definite start and end point as can be observed in most filter impulses (this, on its own, however, is not necessarily an issue). The problem seen in the time domain response of the two signals convolved can be quantified if the frequency domain magnitude response is calculated at a higher resolution as shown in Figure 5.53 (the frequency domain plot in Figure 5.52 was calculated with a length equal to that of the filter). Analysis using this higher resolution shows the excessive ripple that has been introduced by this filter. This can be resolved, as in any other type of filter design, using windowing techniques (Paterson-Stephens & Bateman, 2001). However, the impulse response shown in Figure 5.52 is not yet in the correct format to have a window applied.

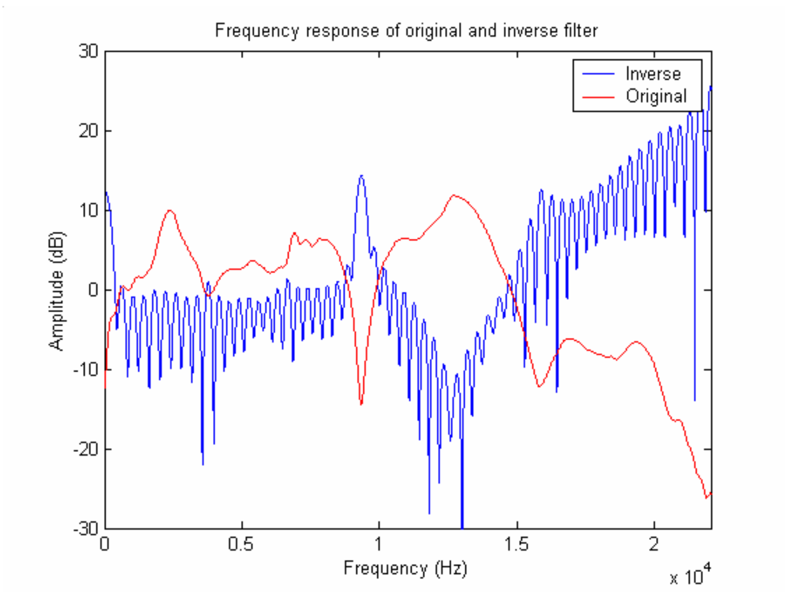


Figure 5.53 Frequency response of the original and inverse filters using an 8192 point F.F.T..

An F.I.R. filter[3] is basically a cyclic signal that will wrap around onto itself. This means that when the inverse filter is calculated, the position of the filter (in the impulse space) is not necessarily correct. For example, the envelope of the filter created in Figure 5.52 is shown in Figure 5.54 along with the ideal position of this filter.

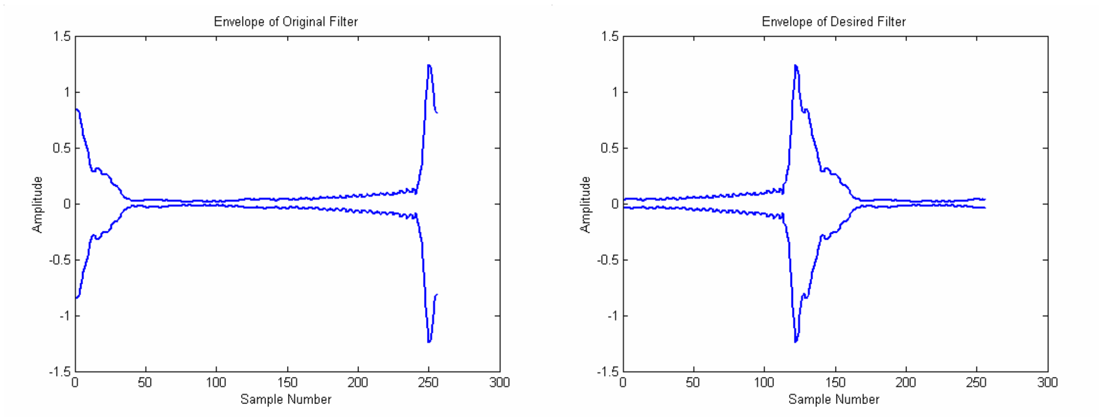


Figure 5.54 Typical envelope of an inverse filter and the envelope of the filter shown in Figure 5.52.

It can be seen in Figure 5.54 that it is desirable for the main impulse to be in the centre of the filter so as to maximise the number of samples given to pre and post delay processing for the sound. It is this main impulse that dictates

[3] Finite Impulse Response – a filter with a fixed length that is convolved (polynomial multiplication) with a signal to apply the filter's time and frequency response onto the signal.

the overall time delay introduced by the filter. As the F.I.R. filter can be treated as a continuous wrappable signal, the impulse response can be repositioned by adding a delay to the response that is to be inverted, as shown in Figure 5.54. To move the main impulse to the centre of the filter, a delay of N/2 samples must be added, where N is the length of the target filter, in samples. This technique also has the benefit of improving the frequency response of the filter, as shown in Figure 5.55 (note that due to the extra samples (zero padded) added to the shifted filter, both filters have been calculated using 256 samples).

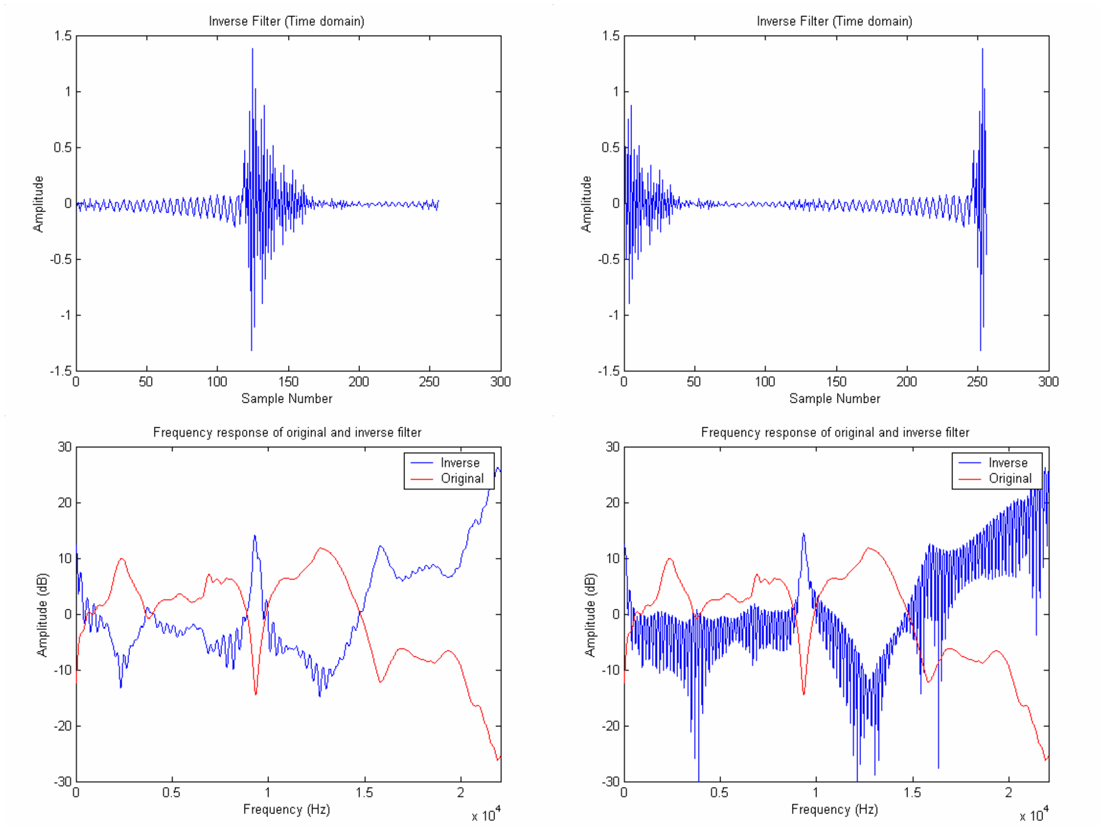


Figure 5.55 **Two F.I.R. filters containing identical samples, but the left filter's envelope has been transformed.**

It can now be seen that the frequency response of the filter has been improved and much of the rippling has been eliminated. This results in a reduction of the artefacts seen in the time domain version of the original and inverse filters convolved (as shown in Figure 5.52, bottom left plot). This is shown in Figure 5.56.

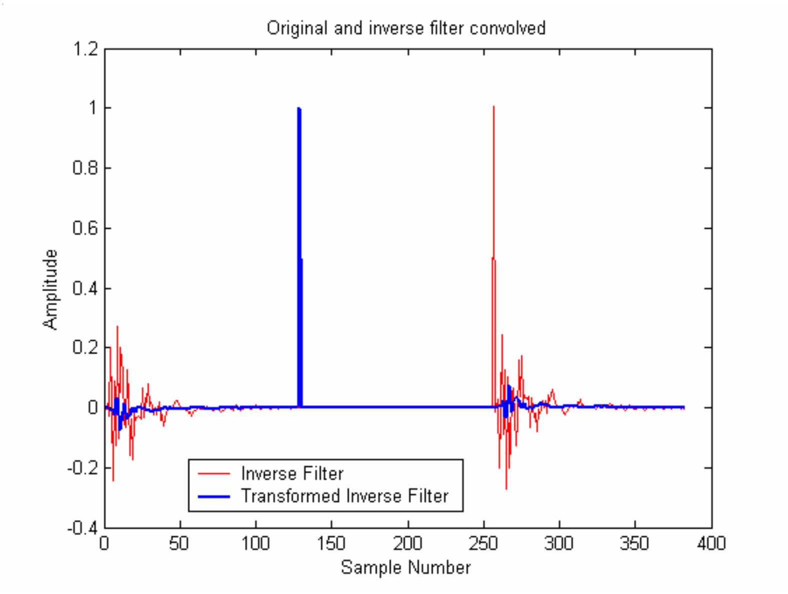


Figure 5.56 The convolution of the original filter and its inverse (both transformed and non-transformed versions from Figure 5.55).

Now that the filter is in the correct format, a window function can be applied to smooth the response still further, and help reduce these time and frequency domain artefacts. The windowed response is shown in Figure 5.57. Using a limited filter size, this is the best realisable response without using the regularisation parameter described in Chapter 3. The only method of improving this further is to create a longer response using zero-padding of the filters used to calculate the inverse. However, the resulting size of the HRTF filters must be taken into account as convolution of the inverse filter and the original HRTF filter will cause its response to increase in size. If the HRTF filter is of length ‘a’ and the inverse filter is of length ‘b’ then the resulting filter will be of a length ‘a+b-1’, and the longer the filter, the more processing power will be needed for it’s implementation. The differences between using a windowed 256-point filter and a windowed 1024-point filter are shown in Figure 5.58.

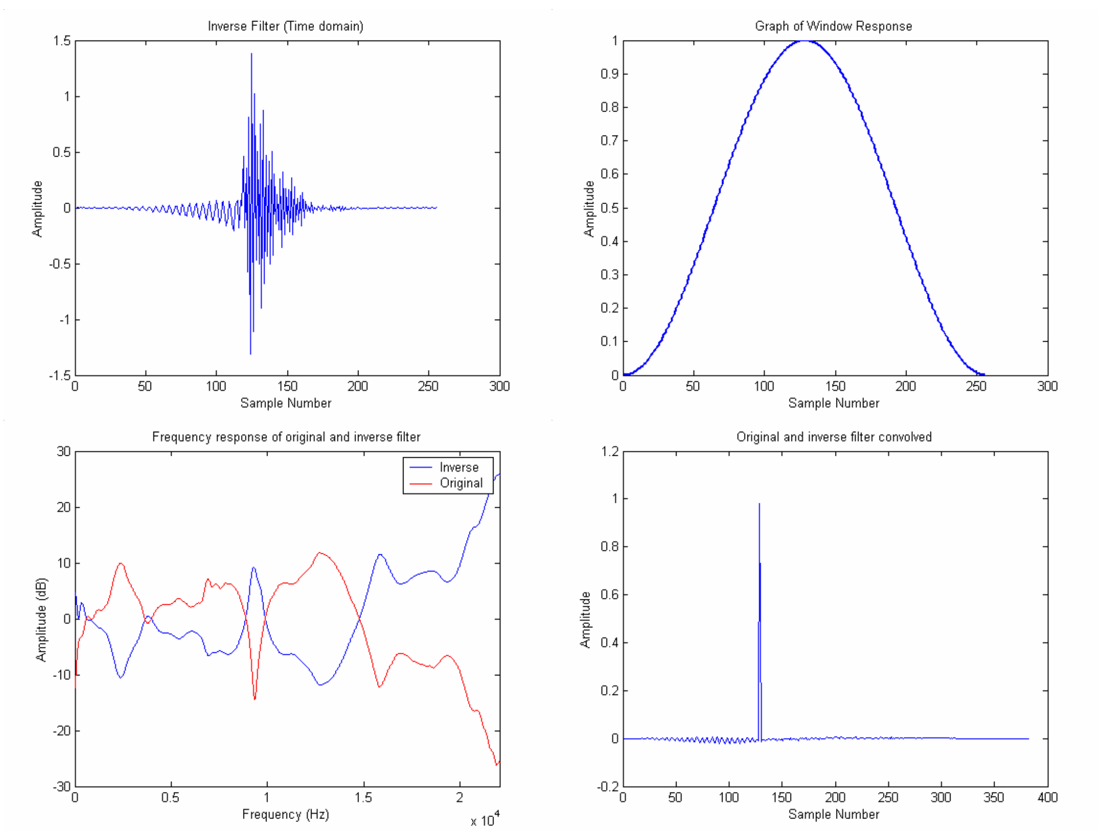


Figure 5.57 A frequency and time domain response of the filter after a hamming window has been applied.

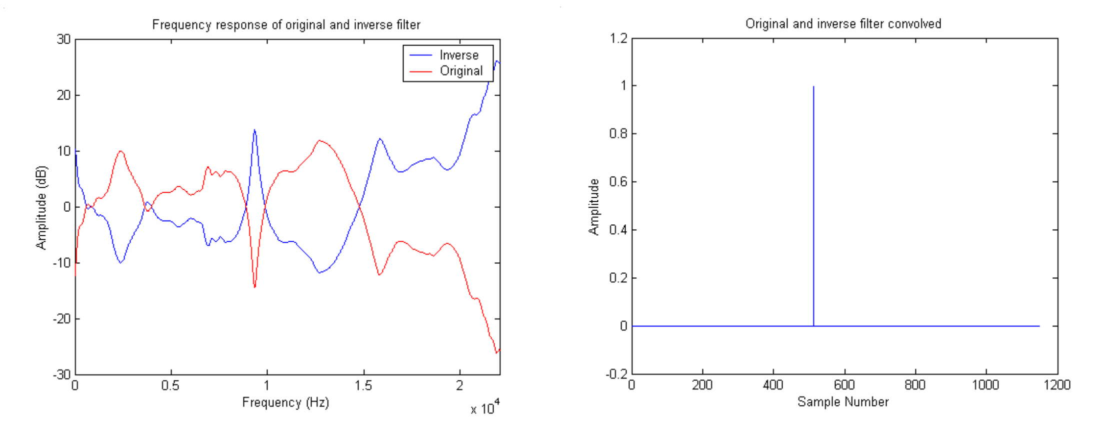


Figure 5.58 The response of a 1024-point windowed inverse filter.

5.4.3 Inverse Filtering of H.R.T.F. Data

When inverse filtering the HRTF data, the only decision that has to be made is which HRTF will be used to equalise the whole HRTF set. Two logical choices are available:

- The near ear response to a sound source at an angle of 90^0 as this will most likely be the filter with the least amount of pinna filtering affecting the response.

- The ear's response to sound directly in front of the listener so that when the sound is positioned at 0^0, the H.R.T.F. responses at the ears are identical and flat.

The 1024-point inverse filters for both of these methods are shown in Figure 5.59. Looking at this figure it can be seen that, in reality, the 0^0 HRTF is far more ill-conditioned to the inversion process when compared to the 90^0 response. Some wrapping of the resulting filter can be seen for the 0^0 response indicating that a longer filter length is desirable. This is to be expected because of the reason stated above (the 90^0 angle has less head/pinna filtering associated with it) and so it is best to use the 90^0, near ear, HRTF as the reference response.

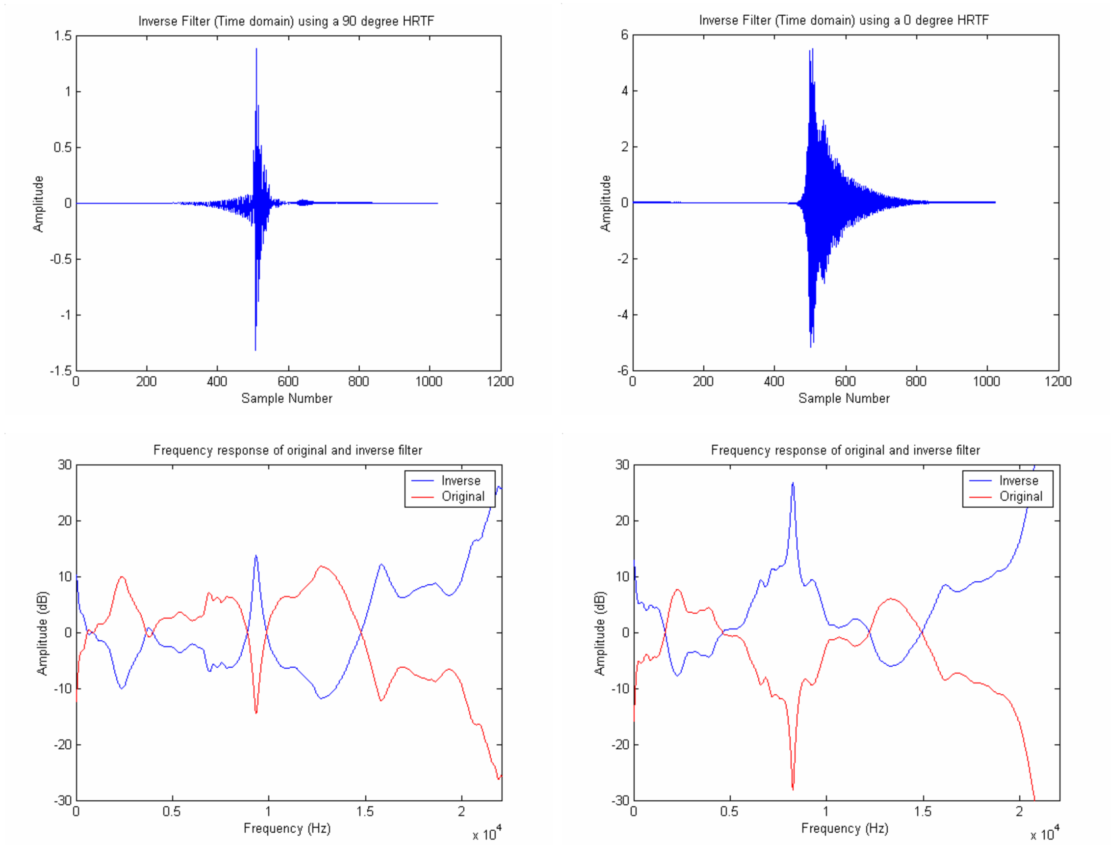


Figure 5.59 **The 1024-point inverse filters using a 90^0 and a 0^0, near ear, HRTF response as the signal to be inverted.**

As an example, a set of H.R.T.F. data has been processed in this way using an inverse filter size of 769-points (so that the convolution of the original with this inverse filter will be equal to 1024-points). Figure 5.60 shows a number of the H.R.T.F. impulses in the time and frequency domain so a comparison of them can be made both before and after inverse filtering.

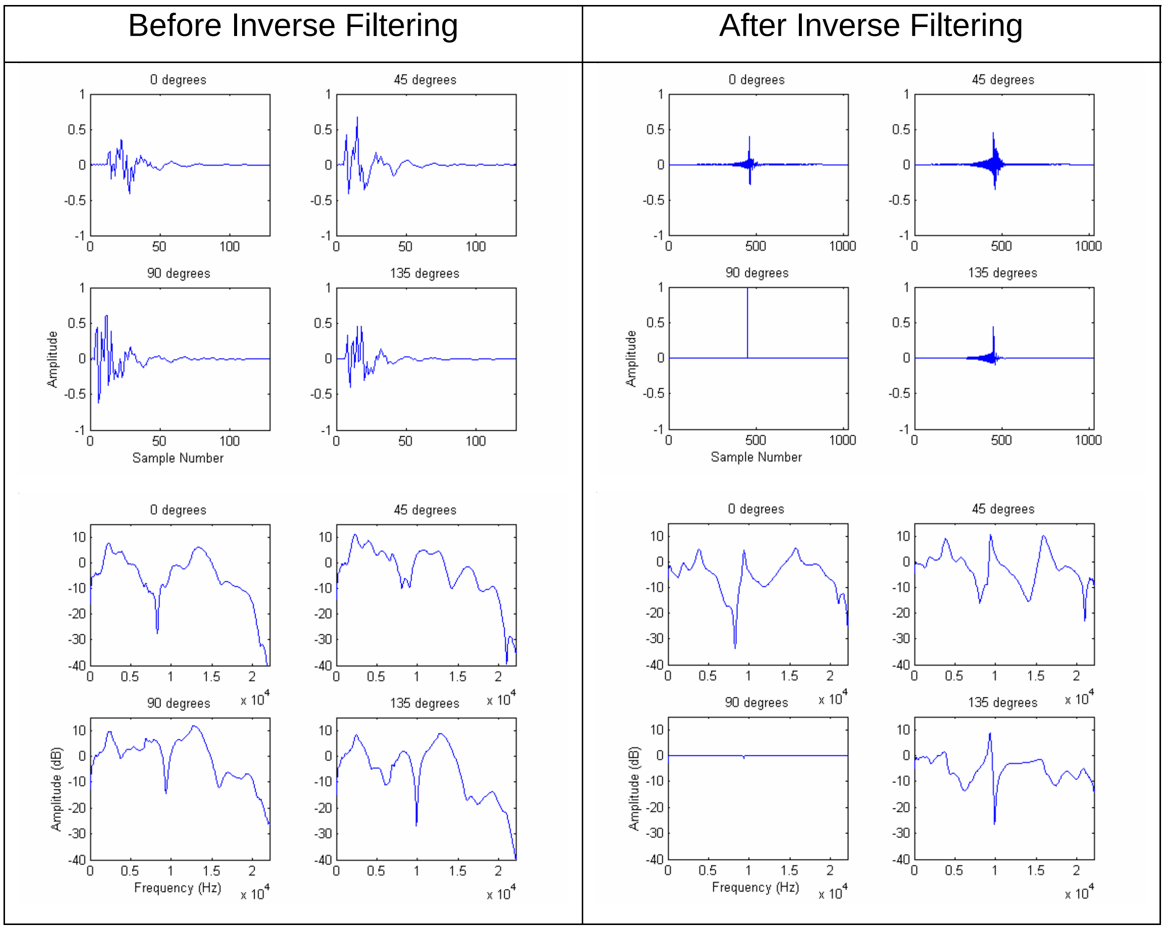


Figure 5.60 **Comparison of a HRTF data set (near ear only) before (right hand side) and after (left hand side) inverse filtering has been applied, using the 90^0, near ear, response as the reference.**

Figure 5.60 shows that although both sets of HRTFs still have a pinna filtering effect, the inverse filtered set have a larger bandwidth, in that extreme low and high frequency components of the impulse responses contain more energy, and contain peaks and troughs in the frequency response that are no larger the originals (for example, the 135 degree frequency response plots both have a notch no lower than around -27 dB). These inverse filtered HRTFs are perceived to be of a better fidelity than that of the originals (which have this response due, in some part, to the non-optimum inverse filtering of the source's response that was used to record the HRTF data in the first place (Gardner & Martin, 1994)). It can also be seen that due to the nature of these new inverse filtered HRTFs, they could also be windowed and shrunk if smaller responses were needed due to processing constraints thanks to the roughly equal amount of pre and post delay filtering (i.e. the highest amplitude parts of the filter are at the middle sample position).

5.4.4 Inverse Filtering of H.R.T.F. Data to Improve Crosstalk Cancellation Filters.

As mentioned at the start section 5.4, one of the problems of the crosstalk cancellation system is that very noticeable colouration of the reproduced sound can occur, both due to the crosstalk cancellation itself, and due to the response of the individual parts of the system (usually speaker to near ear, and speaker to far ear responses). This is why there is a difference between crosstalk cancellation in the free field and crosstalk cancellation using HRTF data. However, as discussed in Chapter 3, system inversion using frequency-dependent regularisation can be used to compensate for this, at the expense of the accuracy of the crosstalk cancellation at these frequencies. For this reason, it is desirable to minimise any potential ill-conditioning due to the response of the individual components of the system *prior* to the 2 x 2 matrix inversion process, thus resulting in the least amount of regularisation needed in order to create a useable set of filters. In this way, the inverse technique described in section 5.4.2 will be utilised in much the same way. For example, the system shown in Figure 5.61 will be used as a basis for the creation of a pair of crosstalk cancellation filters.

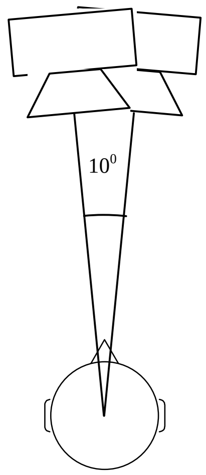


Figure 5.61 System to be matrix inverted.

This is a typical arrangement for a crosstalk cancellation system, and is based on a pair of speakers placed at +/- 5^0 in front of the listener. Using the HRTF set from M.I.T. (Gardner & Martin, 1994) this will give the responses for the near and far ears (assuming symmetry) as shown in Figure 5.61.

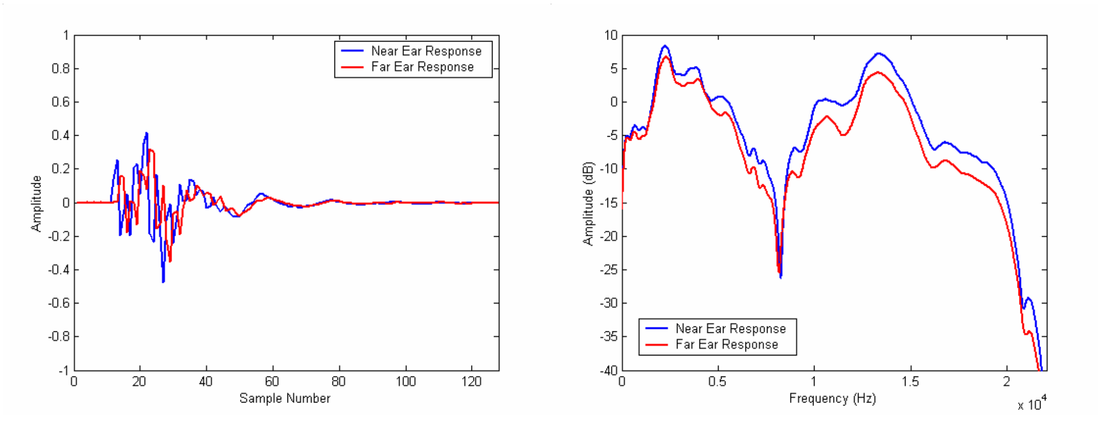


Figure 5.62 HRTF responses for the ipsilateral and contralateral ear responses to the system shown in Figure 5.61.

If a set of crosstalk cancellation filters are constructed from these two impulse responses, using the techniques described in Chapter 3, then the responses shown in Figure 5.63 are obtained (using no regularisation).

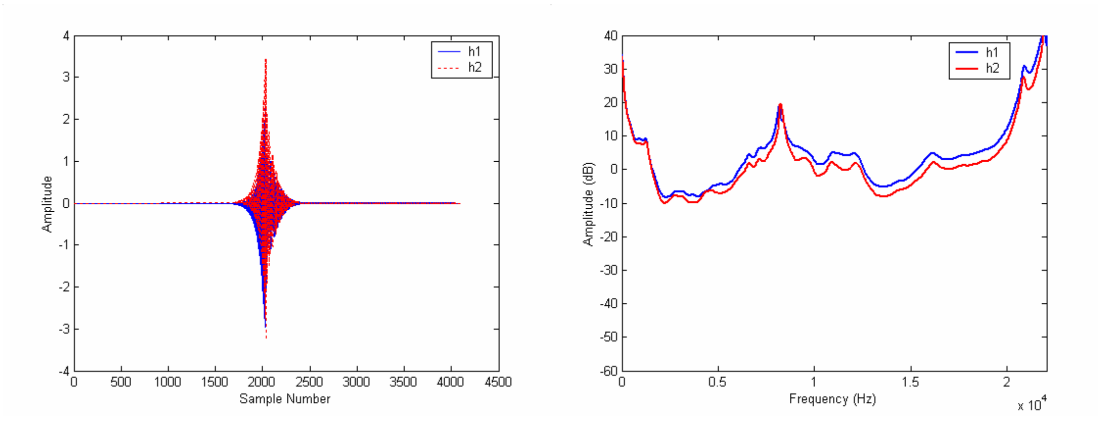


Figure 5.63 Crosstalk cancellation filters derived using the near and far ear responses from Figure 5.62.

It can be seen, from Figure 5.63, that the expected peaks are present. That is, a peak at very low frequencies due, mainly, to the close angular proximity of the speakers and the peaks at around 8 kHz and high frequencies, which appear to be due to the inversion of the responses of the near and far ear HRTFs (as seen in Figure 5.62). When this crosstalk cancelled system is auditioned, not only is a very coloured sound perceived off-axis, but a non-flat frequency response is also perceived on-axis. This is also coupled with a large loss in useable dynamic range as the amplifier and speakers have to reproduce such a large difference in frequency amplitudes. These are mainly because of the reasons stated at the start of section 5.4.1, but also because of the different pinna/head/ear responses observed for different listeners. A more general, yet correct inverse filtering method is needed to correct these problems.

If regularisation is to be avoided as a last resort, then the responses shown in Figure 5.62 must be 'flattened' using inverse filtering techniques. As it is the difference between the near and far ear responses that is important, the filtering of these two responses will have only fidelity implications so long as the same filter is applied to both the near and far ear response. Also, the least ill-conditioned of the two responses is likely to be the near ear response, as it will have been filtered less by the head and pinna, so it is this response that will be taken as the reference (although, due to the small angular displacement of the speaker, there is little difference between the two filters). The inverse filter of the near ear HRTF is shown in Figure 5.64.

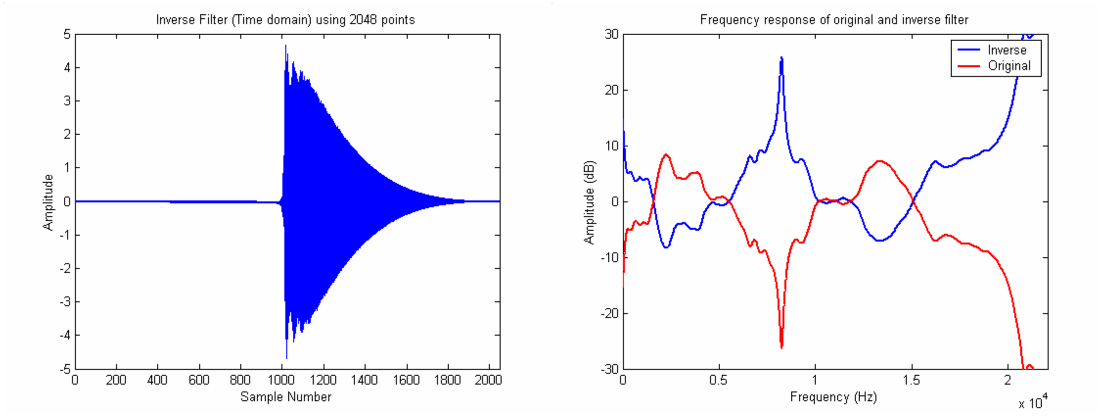


Figure 5.64 Inverse filter response using the near ear H.R.T.F. from Figure 5.62.

Applying this inverse filter to the ipsilateral and contralateral ear responses shown in Figure 5.62, gives the new ipsilateral and contralateral ear responses shown in Figure 5.65. If these filters are now used in the calculation of the crosstalk cancellation filters (using the 2 x 2 inverse filtering technique with no regularisation), then the filters shown in Figure 5.66 are obtained.

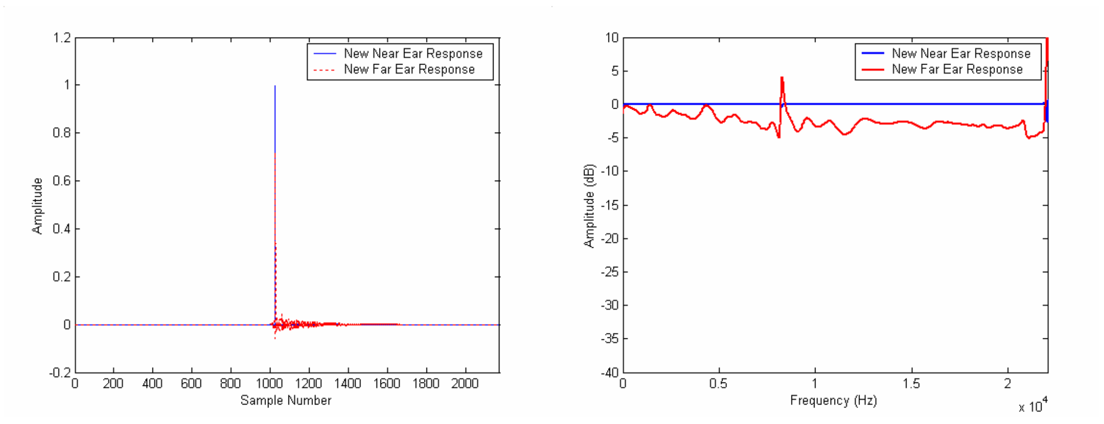


Figure 5.65 **Near and far ear responses after the application of the inverse filter shown in Figure 5.64 (frequency domain scaling identical to that of Figure 5.62).**

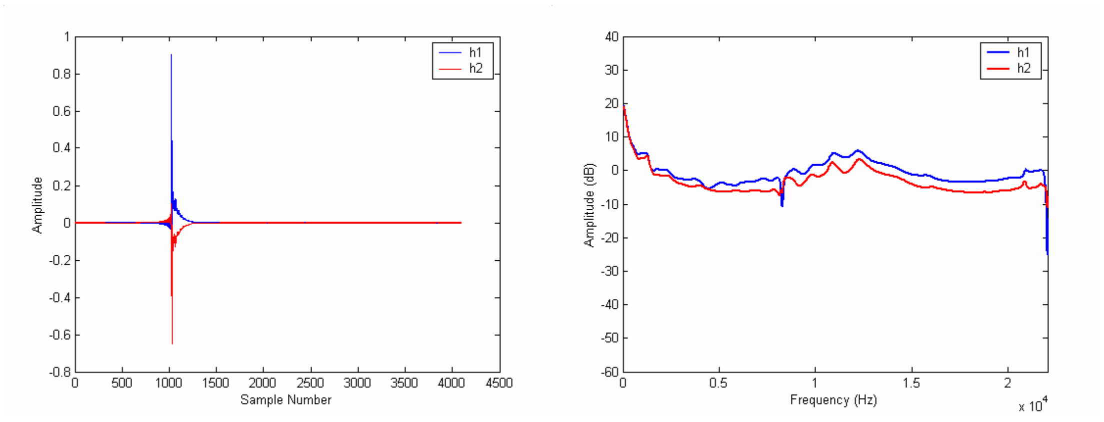


Figure 5.66 **Crosstalk cancellation filters derived using the near and far ear responses from Figure 5.65 (frequency domain scaling identical to that of Figure 5.63).**

The optimisation of these filters using inverse filtering techniques can be verified by observing the responses shown in Figure 5.66:

- The overall response of both of the filters has been flattened with the largest peak above very low frequencies now at around 6dB at around 12.5 kHz, and virtually no peak at very high frequencies, which means that regularisation is no longer needed at these frequencies.
- The peak at low frequencies is now solely due to the 2 x 2 matrix inversion and not the response of the ipsilateral and contralateral ear responses, which has reduced this peak from over 30dB to 20dB. This means that, although regularisation is still needed here, a smaller amount can be applied, making the crosstalk cancellation more accurate in this frequency range.

- The flattening of the filter responses causes the on-axis response to be perceived as much flatter (un-filtered) than before.
- The flattening of the filter responses also has the added effect of making off-axis listening seem far less filtered.
- The crosstalk cancellation filters are actually smaller in length than the originals shown in Figure 5.63, even though the contralateral and ipsilateral ear responses used to calculate them were much larger than the originals shown in Figure 5.62. This is due to the fact the new near and far responses are much less ill-conditioned for inversion (the filters do not have to 'work as hard' to achieve crosstalk cancellation).

These new crosstalk cancellation filters, although much better than filters created using the *raw* HRTF data, still need to use some regularisation, and still sound a little bass heavy. However, at this point, it is still possible to take the inverse filtering technique a step further. As always, it is the difference between the two ears that is important, especially as the pinna used in the HRTF data is not likely to be the same as that of the listener. So, using inverse filtering, it is possible to design crosstalk cancellation filters that require *no* regularisation to correct for the conditioning of the system. If the filter representing 'h1' is used as a reference, then another inverse filter can be created by inverting the response of 'h1'. If this inverse filter is convolved with both *h1* and *h2* then the h1 filter will, in theory, become the unit impulse, and h2 will then be a filter representing the difference between *h1* and *h2*. These filters are shown in Figure 5.67, and Figure 5.68.

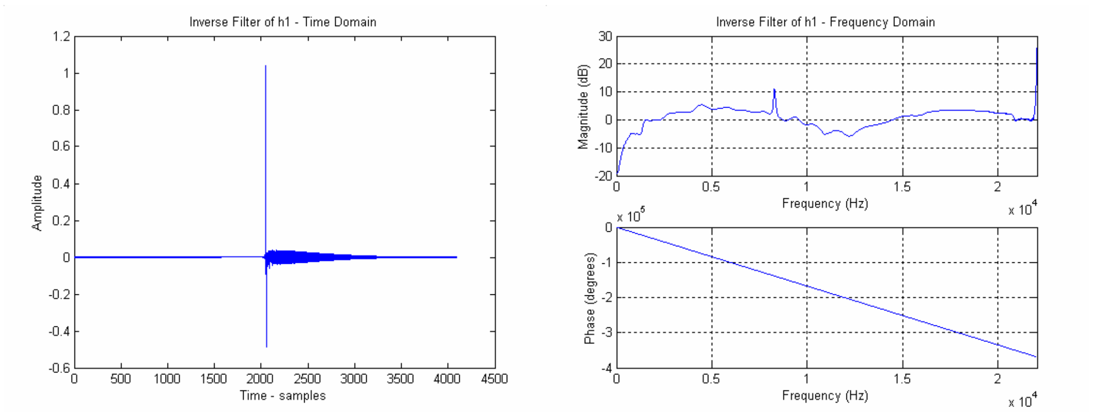


Figure 5.67 Filter representing inverse of h1, in both the time and frequency domain.

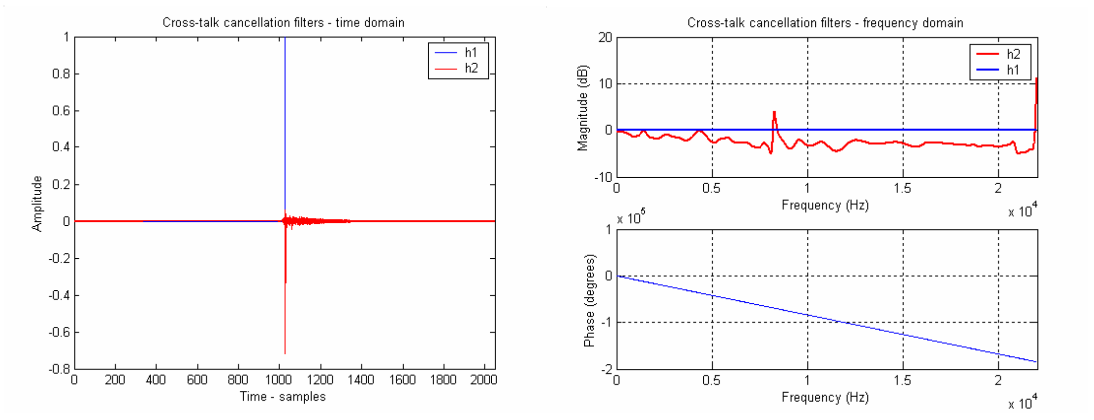


Figure 5.68 Crosstalk cancellation filters after convolution with the inverse filter shown in figure 5.51

It can be seen from Figure 5.68 above that h1 has a flat frequency response and h2 now has very little energy over the 0dB point meaning that the system needs no regularisation. These new, double inverted, filters are also perceived as performing much better than the previous crosstalk cancellation filters, with a less muffled sound and clearer imaging. One other highly useful feature of these new filters is that h1 can be approximated by a unit impulse (as this is what h1 should be, theoretically, anyway) which cuts the amount of FIR filtering in the system by a half, replacing the h1 filters with a simple delay line, as shown in the block diagram in Figure 5.69.

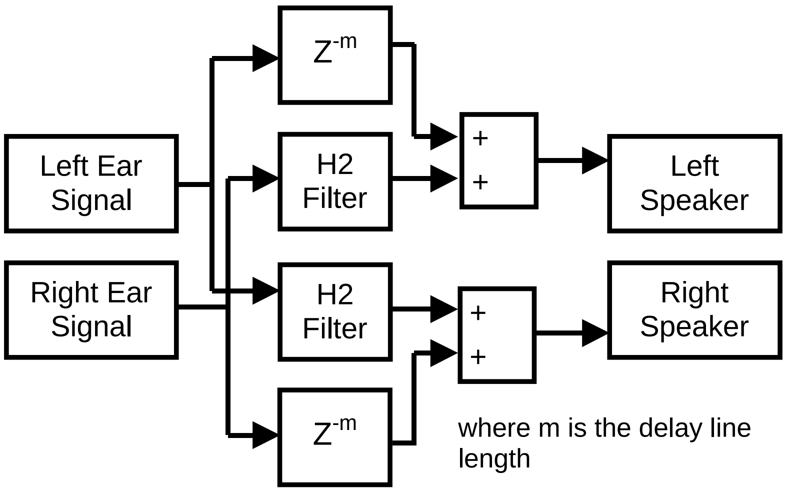


Figure 5.69 The optimised crosstalk cancellation system

However, these double inverted filters do mean that when the speakers are positioned close to each other, the response can be perceived as lacking in bass response when compared to the single inverted case (which is perceived as having a raised bass response anyway). For example, if we inject an impulse into the block diagram shown in Figure 5.69 (but replacing the delay line with the filters again) and compare the results that will arrive at the ear of a listener (although it should be noted that the analysis is using the non-optimum frequency response of the MIT HRTF data), the results shown in Figure 5.70 can be seen (note that the speakers in the University of Derby's Multi-channel research laboratory are actually placed at +/- 3^0, and so filters for this speaker arrangement is shown in Figure 5.70).

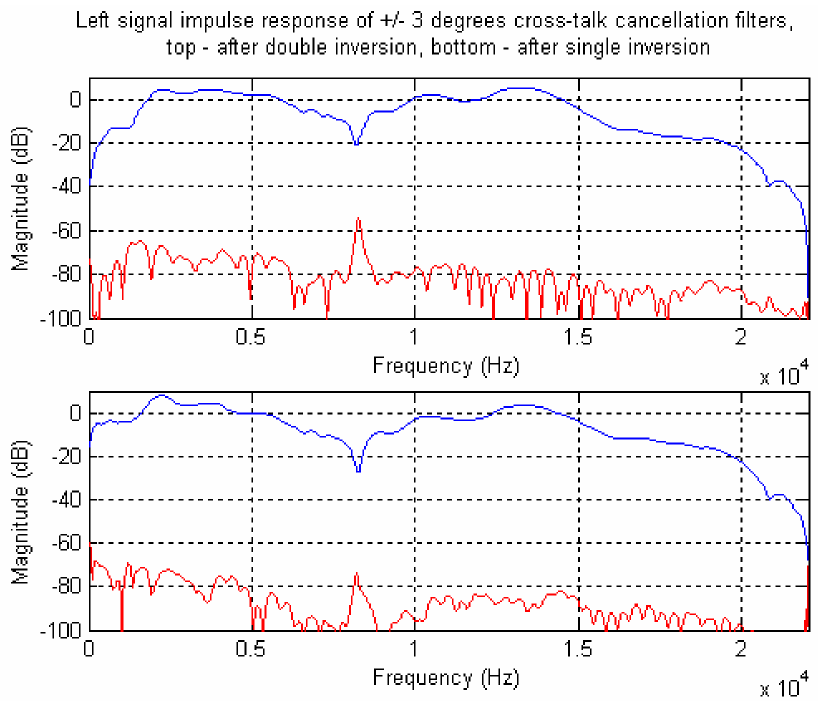


Figure 5.70 **Left Ear (blue) and Right Ear (red) responses to a single impulse injected into the left channel of double and single inverted cross talk cancellation systems.**

Both responses show a good degree of crosstalk cancellation, in the right ear response, with the single inverted system seeming to perform slightly better. The low frequency roll-off can also be noted in the left ear response of the double inverted system. However, these quantitative results cannot necessarily be taken at face value. For example, the single inverted system (lower plot) is perceived as being bass heavy, although this is not shown in these graphs as it is the non-optimum HRTF data used in this analysis. Also, the double inverted system is perceived as performing better at the higher frequencies, although this, again, is not suggested in this plot. It is also, interesting to look at the same graphs for the +/- 30^0 case, as shown in Figure 5.71.

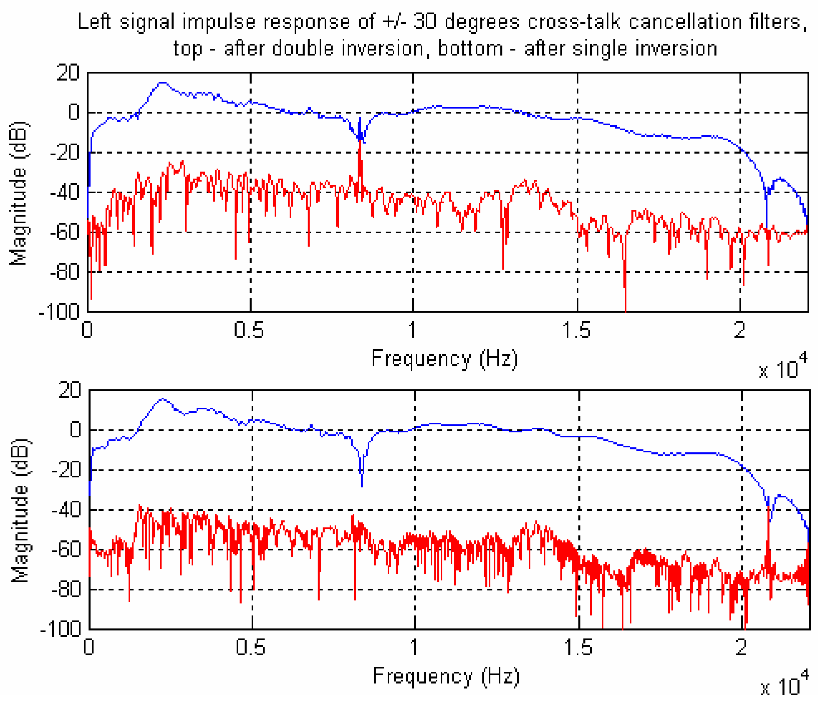


Figure 5.71 **Left Ear (blue) and Right Ear (red) responses to a single impulse injected into the left channel of a crosstalk cancellation system.**

This plot shows two significant results:

- The bass loss is no longer an issue. However this is to be expected as widening the speaker span alleviates the bass boost in the original filters which, in turn, means they do not need to be inverse filtered.
- The cancellation of the right ear signal is shown to be around 20dB worse than that shown for the +/- 3^0 case.

This second point is interesting as the crosstalk cancellation filters have been created in exactly the same way as the +/-3^0 case. This means that the same differences between the filters will be retained. The only *absolute* in the filtering process is the response due to the pinna alone, and it is this discrepancy that must be causing the problem. These two graphs suggest that the further apart the speakers, the more the pinna matching between the listener and the filters becomes important. This would explain why widening the speakers degrades the localisation quality using this system.

5.5 Conclusions

Optimisation techniques have been described, analysed and discussed in this chapter, with the main part of this section concentrating on the optimisation of the Ambisonics decoders.

5.5.1 Ambisonic Optimisations Using Heuristic Search Methods

The main problem to be tackled in this section was the derivation of Ambisonic decoders for irregular arrays, as, although Gerzon & Barton (1992) had suggested some parameters to be used in the design of these decoders, the solving of these equations was previously a lengthy and difficult process. In the analysis of the original work by Gerzon and Barton (1992 & 1998) it was found that:

- Multiple values could be chosen that would satisfy these equations, analytically performing equally well.
- The original coefficients suggested by Gerzon & Barton (1992) were actually non-ideal, with an oversight in the way in which the equations were initially solved leading to a mismatch between the low and high frequency decoders' perceived source position.

Various new methods have been devised and implemented in software to solve these problems:

- A heuristic search method, based on a Tabu search algorithm, has been developed, along with the fitness functions that need to be satisfied in order to automatically generate decoders for irregular speaker arrays. This method has the three following benefits:
  - It automatically solves the non-linear simultaneous equations in an optimal way.
  - Changing the start position for the search will generate a different set of coefficients.
  - This method solves all the parameters of the equations simultaneously which corrects for the low and high frequency decoder mismatch found in Gerzon & Barton's method (Gerzon & Barton 1992 and Gerzon & Barton 1998).
- An analysis technique based on the use of generic HRTF data has been devised to help differentiate between Ambisonic decoders designed using the above method, using head turning as an additional parameter as phase and level differences will generally be similar for each decoder.

The tabu search method has also been shown to work well on the new higher order decoder types, such as the one proposed by Craven (2003), which has far more coefficients to optimise, demonstrating that the Tabu search methodology is easily extendible to more unknowns (either a higher order, or more speakers).

The HRTF analysis technique described above was also used to validate the original work by Gerzon & Barton (1992) which then led to the creation of a heuristic search program, with corresponding fitness functions, used to design Ambisonic decoders for irregular arrays using the HRTF analysis technique first proposed in Wiggins *et al*. (2001) taking into account head turning directly, so reducing the number of decoders produced. The properties of this new technique are as follows:

- For a two-band decoder the correlation between decoders designed using the velocity/energy vector methods and HRTF methods are good.
- Using the HRTF technique a decoder could be designed using more frequency bands, which is impossible using the previous velocity/energy vector method.
- However, the HRTF decoder method is far more computationally expensive and it does take the tabu search algorithm longer to converge on an optimum result, but as this is an off-line process anyway, this is not a major issue.

A small listening test was carried out using both synthetically panned material and pre-recorded material in order to help steer future listening tests aimed at optimised Ambisonic decoders. Although only three subjects were used, the decoder that performed worst in both tests was unanimously seen as an un-optimised decoder based on the default settings of a commercially available B-format decoder for the ITU irregular speaker array. However, although many more subjects would be needed to gain statistically significant results, all the optimised decoders performed well, with the expected decoder performing best in the synthetically panned listening test. As expected, there were no great differences between decoders designed using either optimisation method, as the two systems correlate well with respect to coefficients and, in fact, slightly less optimal decoders seemed to perform well when recorded, reverberant material was auditioned by the test subjects. Also, one reported observation was that the most optimal decoders seemed to deliver a more pleasant listening experience slightly off-centre (when compared to the same decoder in the sweet spot), which is an extremely interesting result that needs to be investigated further.

In summary, the use of the Tabu search algorithm has resulted in a vast simplification of the process of designing Ambisonic decoders, allowing for the Vienna equations (Gerzon & Barton, 1992 & 1998) to be solved correctly for irregular speaker arrangements (although the software concentrates on a typical five speaker horizontal arrangement). This has then been taken a step further through the use of the HRTF data directly.

5.5.2 Further Work for Ambisonic Decoder Optimisation.

Now that the decoder design algorithm can directly use HRTF data the obvious next step is to increase the number of frequency bands. When taking this method to its extreme, this will mean that instead of using cross-over filters, a W, X and Y filter will be created for each of the speaker pairs (or 1 set for the centre speaker). In this way it should be possible to maximise the correctness of both the level and time differences simultaneously for many frequency bands improving the performance of the decoder still further for a centrally seated listener. The software could also be extended to take into account off-centre listening positions which could, potentially, lead to a control over the sweet spot size, trading the performance at the centre, for the performance around this spot. This may well be beneficial, not only to create a ‘volume solution’, but to also circumvent the problems noticed in the listening test with respect to the more optimum decoders, analytically speaking, giving a slightly uncomfortable, obtrusive listening experience directly in the sweet spot.

5.5.3 Binaural and Transaural Optimisations Using Inverse Filtering.

The use of inverse filtering techniques on HRTF data has proved an invaluable tool in the optimisation of both Binaural and Transaural reproduction. An improvement in the frequency response of the crosstalk cancellation filters has been demonstrated which is apparent both on and off axis from the cancellation position. This reduces the need to use the frequency dependant regularisation function; although at the extreme upper frequencies (where little energy in the HRTF data is present) it is still advisable to use regularisation to stop the excessive boost of these frequencies.

It has also been shown how moving the speakers closer together has the effect of improving the analytical crosstalk cancellation figure between the ears of a listener in the sweet spot. This has to be a feature of the pinna filtering mismatches as the differences between the creation and analysis HRTF filters were kept constant, with only the monaural pinna filtering having changed (all the work was based around the same set of HRTF filters and pinna differences between the ears are kept constant).

5.5.4 Further Work for Binaural and Transaural Optimisations.

A method to control the amount of inverse filtering that is carried out on the crosstalk cancellation filters must be used as the single inverted filters sound bass heavy, and the double inverted filters are bass light. This can be done by carrying out the following steps:

- Create the inverse filter in the frequency domain and split into magnitude and phase.
- Create a unit impulse, delayed by half the length of the inverse filter, in the frequency domain and split into magnitude and phase.
- Crossfade the magnitude responses of the two filters using the desired ratio, and use the phase from the unit impulse.
- Mix the magnitude and phase of this filter back into its complex form and inverse FFT into the time domain.

- This will result in a filter that has a linear phase response (that is, pure delay) and a magnitude response can be chosen from flat to the magnitude response of the inverse filter.
- Use the above filter as the 2nd inversion filter in the creation process of the crosstalk cancellation filters.

Once the above steps have been carried out, listening tests can be carried out to determine which filters are perceived as having the flattest response.

5.5.5 Conversion of Ambisonics to Binaural to Transaural Reproduction

Although the conversion from the base format of Ambisonics has been described in Chapter 4, there are still some ongoing issues that have meant that listening tests on this part of the project have not taken place. During this project all of the systems have been looked at separately with main optimisation work carried out on the Ambisonics decodes and the crosstalk cancellation systems.

The conversion of Ambisonics to binaural is now well documented (see Noisetering *et al.*, 2003 for the most recent overview) and this, coupled with the inverse filtering techniques described in section 5.4 works well. Similarly, playing a standard binaural recording over the two speaker crosstalk cancelled system described in the same section also works well, with the inverse filtering techniques resulting in a much flatter, un-filtered sound when compared to a crosstalk cancelled system using raw HRTF data. However, when combining these two steps and attempting to reproduce an Ambisonic decode over either a two or four speaker crosstalk cancelled array, sub-optimal results are experienced with heavily filtered results perceived. Further work is needed in this area to bring this conversion process up to an acceptable level. However, for further work the following avenues will be investigated:

- The use of Bumlein's shuffling technique in order to convert a coincident recording into a spaced one at low frequencies will be attempted as this will remove the need for Ambisonic to binaural conversion step, and will reduce some of the filtering applied to the system.

- The crosstalk cancellation and Ambisonic to binaural conversion steps are taken in isolation; however, the filtering and calculation of crosstalk cancellation filters can be combined by using the Ambisonic to binaural decode function shown in equation (4.3), as the target function for the crosstalk cancellation inversion equation shown in equation (3.13). This will mean that inverse filtering is not needed as the filters response to pinna should, to some extent, cancel each other out, resulting in a less filtered system.

Chapter 6 - Implementation of a Hierarchical Surround Sound System.

While carrying out this research it became apparent that although the Matlab/Simulink platform was very useful in the auditioning and simulation of surround sound systems, more efficient results (with regards to processor loading) could be achieved, particularly when FIR filtering, if custom programs were written for the Windows platform using the Win32 API.

In this chapter the various signal processing algorithms and implementation details will be discussed, so as to build up a library of functions to be used in multi-channel audio applications.

The platform specific code will then be investigated so that an audio base class can be constructed, and it is this class that will form the basis for audio applications.

Once the necessary background information and techniques have been discussed, an example application based upon the surround sound system described in Chapter 4 will be covered.

6.1 Introduction

At the beginning of this research it was assumed that the best platform for the implementation of a system that relied on digital signal processing techniques was one based around a digital signal processor. However, this seemingly logical assumption has now been challenged (Lopez & Gonzalez, 2001).

Around ten years ago D.S.P. devices were far faster than home computers processors (Intel, IBM, etc.), but whereas D.S.P. core speeds have been increasing at a steady rate (approximately doubling every two years), the rate of increase of core speed of a P.C. processor is now doubling every year. This has resulted in the processing power available on fast PCs now being greater than that available on more expensive D.S.P. chips (Lopez & Gonzalez, 2001). As much of the testing and algorithm development was already taking place on a PC platform (using Matlab® and Simulink®) it soon became apparent that this platform would be suitable for the final implementation of the system and, in some ways, be far more suited than a dedicated D.S.P. platform.

Using the PC as a signal processing platform is not a new idea (Lopez & Gonzalez, 2001; Farina *et al.*, 2001), but has not been viable for surround sound until fairly recently. This is mainly due to the fact that reasonably priced, multi-channel cards (16 or more channels) are now readily available and are not only the perfect test platform for this surround sound project, but also, once the technology is in place, they provide a perfect platform to actually develop surround sound software. It is, of course, also due to the fact that Intel's Pentium and AMD's Athlon processors are now very powerful and can easily process over 32-channels of audio in real-time. Therefore, convolving long filters with just a few channels of audio (as in crosstalk cancellation) is not a problem for today's PCs (assuming efficient algorithms are used, see later in this chapter). So, when it comes to developing such a system, what options are available?

- Home PC computer (Host Signal Processing).
- Digital Signal Processor Platform.
- Hybrid of the two.

Each of the systems described above have their pros and cons and each of these methods have been utilised, at some point, during this project. A description of each will be given.

6.1.1 Digital Signal Processing Platform

A Digital Signal Processor is basically a fast micro-processor that has been designed and optimised with signal processing applications in mind from the outset (Paterson-Stephens & Bateman, 2001). This means that it generally has a more complex memory structure when compared to a 'normal' micro-processor and a more specialised command set. An example of a memory structure used by D.S.P.s is a system is known as dual-Harvard architecture. A standard micro-processor is normally designed around the von Neumann architecture (Paterson-Stephens & Bateman, 2001), and although a thorough investigation into these techniques is not part of the scope of this project, a brief explanation will be given to help differentiate between D.S.P.s and PC micro-processors.

Von Neumann architecture is reasonably straightforward, having one memory space, one internal data bus and one internal address bus. All of these components are used in the reading and writing of data to and from memory locations etc.. A diagrammatic view of von Neumann architecture is shown in Figure 6.1. Basically the Internal Address Bus selects what data is to be read/written, and then this is sent to the C.P.U. or A.L.U. for processing along the internal data bus.

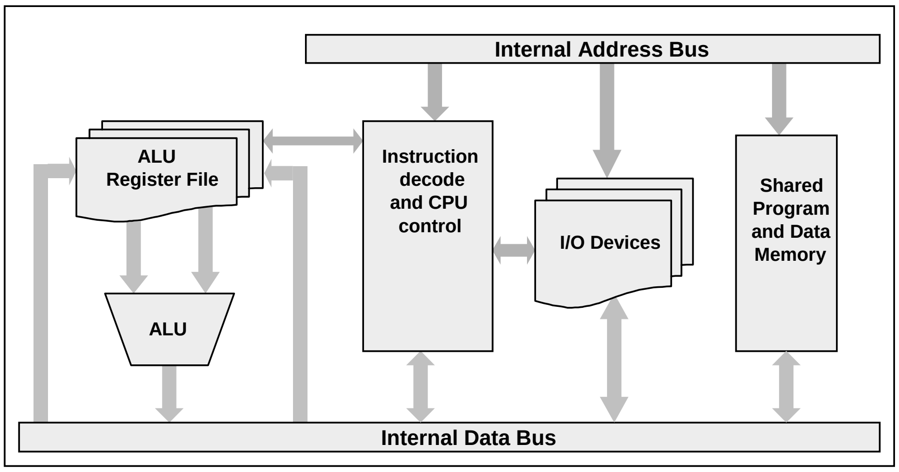


Figure 6.1 A Von Neumann Architecture.

A Harvard architecture (see Figure 6.2) based micro-processor (common in D.S.P. devices) has a very similar layout to the von Neumann architecture, except that three memory spaces, three address buses and three data buses are used as follows: one address bus, memory space, and data bus for program memory, one for X data memory and one for Y data memory. This means that the D.S.P. device can access memory more efficiently, being able to read/write up to three memory locations per clock cycle, as opposed to one using Von Neumann architecture. Also, a more complex Address Generation Unit (A.G.U.) is normally included that can handle such things as modulo address (circular buffering) and bit-reversed addressing (used in Fast Fourier Transforms). This is another task that is taken away from the main processor incurring no extra processor overhead.

As explained above, it is mainly the architecture of the system that differentiates between a D.S.P. and a PC micro-processor. However, another difference between a D.S.P. and a PC is that a D.S.P. has no ‘operating system’ as such (although specialised real-time operating systems can be employed). That is, each D.S.P. platform is configured for optimal performance using whatever peripherals are used with it. It is not a general, ‘jack of all trades’ with flexibility being the key feature, like a PC. The advantages of not having an operating system will become more apparent when discussing the PC platform. The D.S.P. platform is designed for real-time processing, that is, processing containing no perceivable delay.

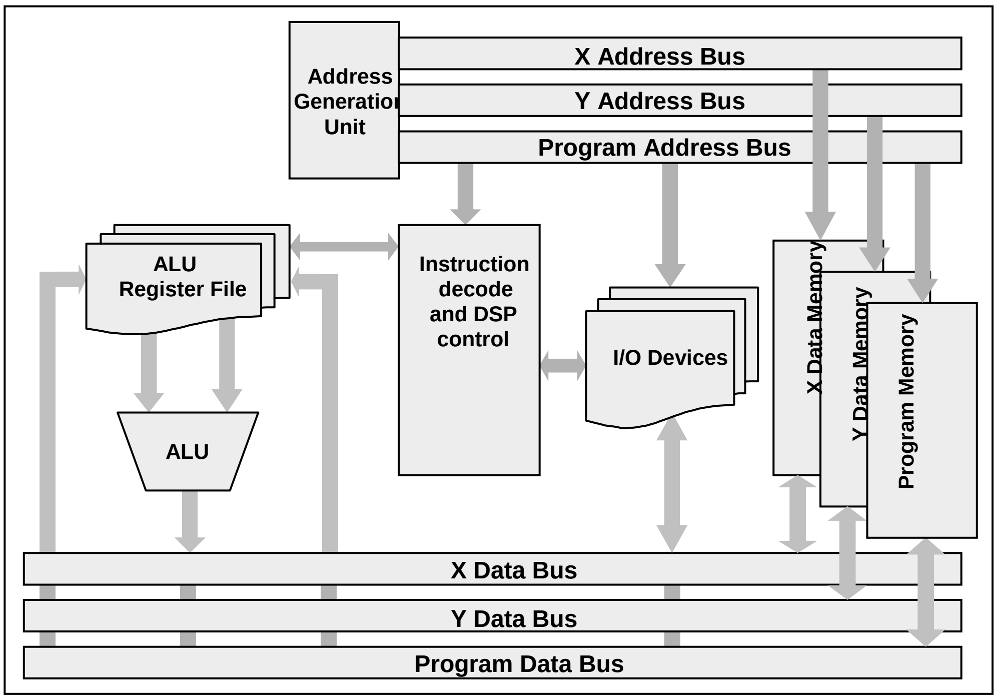


Figure 6.2 Diagram of a Harvard Architecture

6.1.2 Host Signal Processing Platform (home computer).

A PC (or Apple Macintosh) can be used as a system for carrying out digital signal processing. This is now a viable solution because processors for these platforms are now becoming very fast and the distinctions between the micro-processor and D.S.P. are becoming more blurred as the PC has more low-level optimisations for signal processing applications (such as streamed music and video, via the World Wide Web). One of the PC’s biggest assets and potentially largest limiting factors is its operating system. In this project the

Windows 2000 operating system was used. This operating system was chosen as it is more stable than Windows 98, is compatible with more software than Windows NT and uses fewer resources than Windows XP. In any case, all these Microsoft platforms use the same API, namely, Win32. Firstly, the reason that the operating system is the PC's greatest asset is that it's A.P.I. simplifies many operations on the PC and makes programming graphical user interfaces relatively straightforward (as opposed to generating code to run, say, a separate LCD display). Also, the operating system handles all the calls to peripherals using a standard function set. This means that the programmer does not need to know exactly what hardware is in the machine, but can just quiz Windows as to whether the hardware meets the requirements needed (e.g. it has the correct number of channels available). The operating system also has disadvantages for similar reasons. Windows is a graphical user environment, that is, it is geared towards graphical applications. Audio, of course, is very well supported, but must be accessed using the Windows A.P.I., that is, direct access of the underlying hardware is not possible under Windows. When using this, it is soon noticed that considerable latency can be introduced by both taking audio as an input and passing it out as an output, and although this latency can be specified (within limits), the lower the latency, the more unstable the system. This will be explained in more detail later in this Chapter.

6.1.3 Hybrid System

The most user-friendly technique for developing such a system is by using a hybrid system comprising of the two systems mentioned above. This system would not only be a very easy system to develop, but would also be very cost effective as a product, as half of the hardware platform (i.e. the PC) would already be in place. It would include the positive aspects of both of the above systems, with a graphic user interface being programmed and realised on the host PC system, but with the actual processing of the audio stream being handled by the D.S.P. card, meaning that latency is no longer a problem, and tried and tested G.U.I. techniques can be utilised on the P.C. side. Such a system can be devoid of any noticeable latency as the P.C. side is used to just update a few parameters on the D.S.P. card. For example, if a three-dimensional panning algorithm was to be implemented, then the D.S.P. card would handle all of the audio passing through the system, mixing the audio signals together, and passing the sounds to the correct speakers, at the correct levels. The P.C. would be passing just the co-ordinates of where the virtual sources are to be panned to. This also has the benefit of taking some of the processing load off the D.S.P. card, as the P.C. can be used to calculate coefficients, etc. that may rely on computationally expensive floating point calculations, such as square roots and trigonometric functions, with the results passed to the D.S.P. card for use.

6.2 Hierarchical Surround Sound System – Implementation

Although, as mentioned above, the hybrid system is the ideal solution for the development of the hierarchical surround sound system, it was not a practical solution for this particular project, mainly due to the cost of the D.S.P. development boards with true multi-channel capability (although such an affordable multi-channel board has now become available from Analogue Devices®). Thus, as much of the testing and investigative work was carried out using a P.C. with a multi-channel sound card (using Matlab, Simulink and a Soundscape Mixtreme, 16-channel sound card), it was decided that this would be the platform used for the realisation of the project's software. For the explanation of the software application developed as part of this project, this section will be split into two main sub-sections:

- The techniques and algorithms needed for the successful implementation of the system described in chapters 3, 4 and 5.
- An explanation of the Windows platform, its associated A.P.I.s, and considerations and techniques acquired for this platform specific programming task.

6.2.1 System To Be Implemented.

Figure 6.3 shows a simplified block diagram of the proposed hierarchical surround sound system.

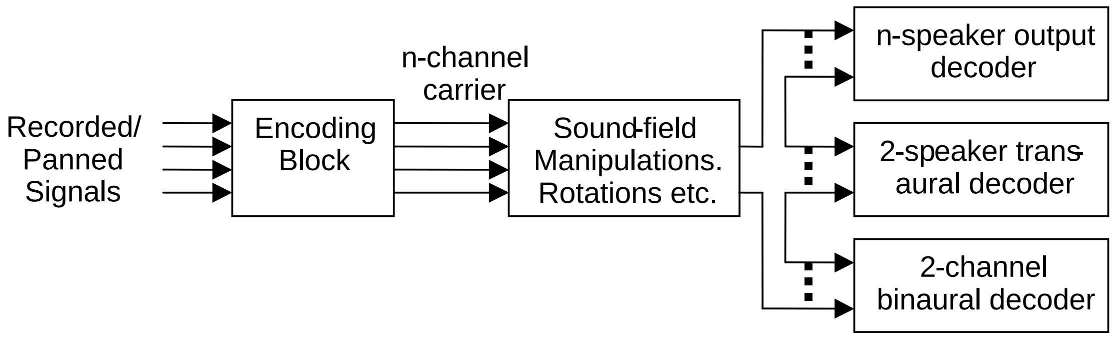

Recorded/ Panned Signals
Encoding Block
n-channel carrier
Sound-field Manipulations. Rotations etc.
n-speaker output decoder
2-speaker trans-aural decoder
2-channel binaural decoder

Figure 6.3 The hierarchical surround sound system to be implemented.

It can be seen from this block diagram that the proposed system has a number of distinct sections that consist of:

- Recording of input signals, which will be in 1st Order B-format, in this example.
- Sounds will be able to be manipulated internally (rotated, for example) while in B-Format.
- These four-channel B-Format signals will then be decoded in one of three ways:
  - Multi-speaker panned output.
  - 2 or 4 speaker transaural output.
  - 2-channel binaural output.

In order to describe how these functions will be implemented in a C++ environment it is necessary to understand how the Windows operating system will pass the data.

- The sound data will be presented in buffers of a fixed size (a size that is fixed by the application itself).
- The sound data will initially be passed to a buffer as an 8-bit unsigned (char), although the application will always be dealing with 16-bit signed integers (short) on the input and output sections.
- All intermediate processing will then take place at 32-bit floating point precision.
- The application will use 8-channels in and 8-channels out from a single sound card.

6.2.2 Fast Convolution

One of the most processor intensive functions needed in the hierarchical surround sound software is that of convolution which is needed for the binaural and transaural reproduction systems. Also, for accuracy it is desirable for the cross-over filtering, needed in the Ambisonic decoders, to be carried out using F.I.R. filters, as these possess linear phase responses in the pass band (that is, pure delay), and so will cause the least distortion to the audio when the two separate signals are mixed back together (as long as the filter length, and therefore delay, is the same for each of the filters). F.I.R. filters are simple to implement in the time domain (they are the same as polynomial multiplication) but are very computationally expensive algorithms to perform. Filtering of this kind is much more efficiently handled in the frequency domain, thanks to the Fast Fourier Transform algorithm. However, convolving two signals together in the frequency domain is slightly more complex, when compared to its time domain equivalent.

To understand why other considerations must be taken into account for frequency domain convolution let us first consider the time domain version of the convolution algorithm. If we have two signals, c and h, where c is the signal to be convolved and h is the impulse response that we will convolve the signal with, the convolution of these two signals is given by Equation (6.1).

$$y = c \otimes h$$
$$y(n) = \sum_{i=1}^{128} c(n-i)h(i)$$

(6.1)

where y = result
n = sample number
i = index into impulse response

In the above case, the impulse that is to be convolved with the signal is 128 samples long, and it can be seen that the convolution process works on the past 128 samples of the signal. In programming terms this suggest that this algorithm can be implemented using a circular buffer that is set to store the current sample, and the preceding 128 samples before the current sample. If the impulse is stored in another circular buffer, then the implementation of this algorithm will follow the block diagram shown in Figure 6.4.

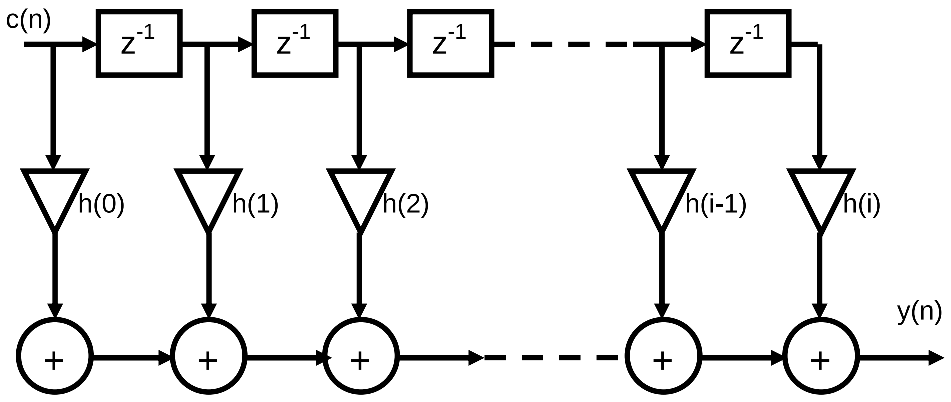


Figure 6.4 Time domain convolution function.

From Figure 6.4 it can be seen that this algorithm will take ‘i’ multiplies and additions per sample which, considering 128 samples represents an impulse response length of 0.003 seconds at a sampling rate of 44.1kHz, would not be suitable for longer impulses. So, how can this algorithm be transferred to the frequency domain? It has already be noted that time domain polynomial multiplication is the same as frequency domain point for point multiplication (i.e. time domain convolution is the same as frequency domain multiplication), and this fact can be used to improve the speed of this algorithm. Taking this into account for a fixed length signal is relatively straightforward. If your original signal is 256 samples long, and the impulse is 128 samples, as long as the F.F.T. size used is longer than the final length of these convolved signals (256+128-1), then both the signals can be transferred into the frequency domain, multiplied, point for point (note that this is the multiplication of complex numbers), and then an inverse-F.F.T. applied. However, if the incoming signal needs to be monitored as it is being fed into the system (such as in a real-time system) then, obviously, we cannot wait to find out the length of the signal in question, the incoming signal must be split up into slices (which is what happens in a computer, anyway). Furthermore, once the signal has been split up, this simple frequency domain convolution will not work correctly, that is, you cannot just multiply a slice by the frequency domain impulse and inverse F.F.T. it again, as the slice has increased in size. Therefore, some form of overlap-add scheme must be used (Paterson-

Stephens & Bateman, 2001). A block diagram showing this process is shown in Figure 6.5.

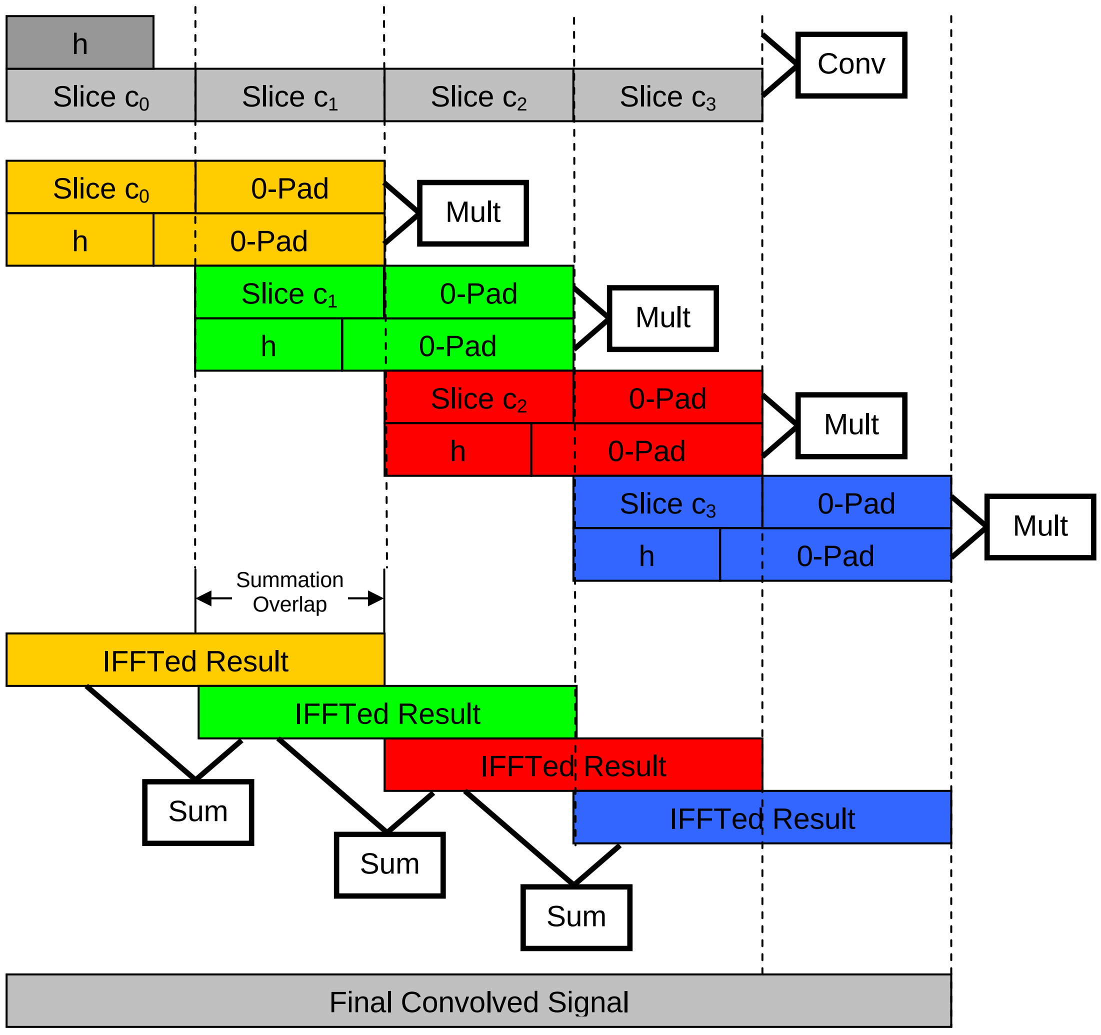


Figure 6.5 Fast convolution algorithm.

The example shown in Figure 6.5 uses a slice length of 128 samples, an impulse length of 100 samples, and a zero-padded F.F.T. length of 256 samples (as 128+100-1 = 227 samples, and 256 is the next power of 2 higher than this). This system means that the minimum latency achievable by this method is measured by the slice size. This example is a specific example of the overlap add system, but shows perhaps the simplest overlap relationship between the multiplied segments. A more general relationship between the length of the slice, and the overlap for summation is given in Equation (6.2).

```
Summation Overlap = (FFT Length) – (Length of Slice).
where:
(Length of Slice) + (Length of Impulse) – 1 <= FFT Length.
```

(6.2)

So, for this example, if the slice length is equal to 225 and the impulse length is 32, then the F.F.T. size could still be 256 (225+32-1=256), and the summation overlap would be 31 (256-225=31). This is a useful parameter to know so the length of the input slice can be maximised when compared to the F.F.T. size to increase the efficiency of the program (make more multiplies count, so to speak). For example, if an F.F.T. size of 256 samples was to be used and the impulse had a length of 32 samples, then a slice size of 225 should be used so as to minimise the summation overlap, and minimise the number of slices that the sound should be divided into (and, hence, the number of times the algorithm must be carried out). Due to the number of specific function calls and number types that are needed for this algorithm, it will be described in C later, when disscussing the more platform specific parts of the application. However, as an example, the Matlab code for such an algorithm is given in Table 6.1.

```
slicesize=225;
impsize=32;

fftsize=256;

if slicesize+impsize-1>fftsize
    error('FFT size must be GREATER or EQUAL to slicesize+impsize-1')
end

%Load signal and impulse
ht=wavread('h0e045a.wav');
ct=wavread('Test.wav');

%Convert Stereo files to a mono array
c=ct(:,2)';
h=ht(1:impsize,2)';

%create frequency domain impulse
fh=fft(h,fftsize);
%clear temp storage for summation block
told=zeros(1,fftsize);

%zero pad signal, if not an exact multiple of the
%slice size
if length(c)/slicesize~=ceil(length(c)/slicesize)
   c(length(c)+1:slicesize*ceil(length(c)/slicesize))=0;
end

for i=1:slicesize:length(c)
```

```
    %create frequency domain slice
    fc=fft(c(i:i+slicesize-1),fftsize);
    %multiply with impulse
    fr=fh.*fc;
    %IFFT result
    r = real(ifft(fr,fftsize));
    %Summation of result (res) with portion of last result (told)
    res(i:i+slicesize-1) = r(1:slicesize) + told(1:slicesize);
    %update using last result ready for summation next time.
    told=zeros(1,fftsize);
    told(1:fftsize-slicesize) = r(slicesize+1:fftsize);
end
```

Table 6.1 Matlab code used for the fast convolution of two wave files.

6.2.3 Decoding Algorithms

The crux of the algorithmic work carried out during this research is concerned with the decoding of the B-format (1st or 2nd order) signal, and it is these algorithms that will be discussed here. As all of the decoders (apart from the simplest multi-speaker decoders) rely on filtering techniques, they will be utilising the frequency domain filtering techniques discussed in section 6.2.2.

The first step in all of the decoding schemes is to decode the Ambisonics audio to multiple speakers, as it was originally intended. As discussed in Chapter 5, for the most psychoacoustically correct decoding methods, cross-over filtering must be used. So far, it has been established that the samples will arrive for processing, and be passed back into a 2-dimensional array, as this is the most flexible system of holding multi-channel audio data in memory. These Ambisonic audio streams will normally consist of 3, 5, 4 or 9 channels of audio data (1st order horizontal only, 2nd order horizontal only, full 1st order, or full 2nd order, respectively). The actual derivation of the coefficients needed for this process was covered in Chapter 5 and so will not be repeated here. All of the speaker feeds in an Ambisonic system are derived using combinations of the various channels available. To this end, it can be useful to specify an Ambisonic structure specifically so as to simplify writing audio applications later on. The structure used to represent an Ambisonic (1st or 2nd order) carrier will comprise:

- Nine pointers to floats.
- An integer length parameter.
- A Boolean flag indicating a 1st or 2nd order stream.

The decision as whether to make the Ambi variable a structure or a class was taken early on in this research, where a structure was decided upon. This was mainly because any functions using this Ambi variable would have to be made global functions, and so not associated with any Ambi structure in particular, and this was thought to be a less confusing system when dealing with more than one Ambisonic stream. However, in hindsight, it would have made little difference either way. The code for an Ambi structure is given in Table 6.2.

```
#define FIRSTORDER 0
#define SECONDORDER 1
struct Ambi
{
      float *W,*X,*Y,*Z,*R,*S,*T,*U,*V;
      int Length;
      bool Order;
};

void AllocateAmbi(Ambi *aSig, const int iLen, bool bAllocChannels,
bool bOrder)
{
      aSig->Length = iLen;
      aSig->Order = bOrder;
      if(bAllocChannels)
      {
            aSig->W = new float[iLen];
            aSig->X = new float[iLen];
            aSig->Y = new float[iLen];
            aSig->Z = new float[iLen];
            if(bOrder==SECONDORDER)
            {
                  aSig->R = new float[iLen];
                  aSig->S = new float[iLen];
                  aSig->T = new float[iLen];
                  aSig->U = new float[iLen];
                  aSig->V = new float[iLen];
            }
      }
}
```

Table 6.2 Ambi Structure.

Included in Table 6.2 is a function for allocating memory dynamically and setting the other flags for the Ambi structure. A choice of whether to allocate memory is necessary as two situations are possible:

- The sources are entering the system as mono signals that are to be panned. The extra channels needed for an Ambisonic signal must be allocated.

- A B-format signal (1st or 2nd order) is entering the system. These channels can be used directly by assigning pointers directly to these channels.

As described in Chapter 5, there are two methods of decoding to an Ambisonic array. There is decoding to a regular array, and decoding to an irregular array. Of course, the decoding for a regular array is really just a special case of the irregular decoding (all of the speakers have the virtual response pointing in the same directions, with just the polar pattern altering for different frequency bands), and it has also been observed that for particularly large arrays, even simpler decoding should be used (Malham, 1998), limiting the amount of out of phase signal emanating from the speakers opposite the desired virtual source position. Let us first take the regular array case, as this is the simplest. A simple block diagram of this system is shown in Figure 6.9.

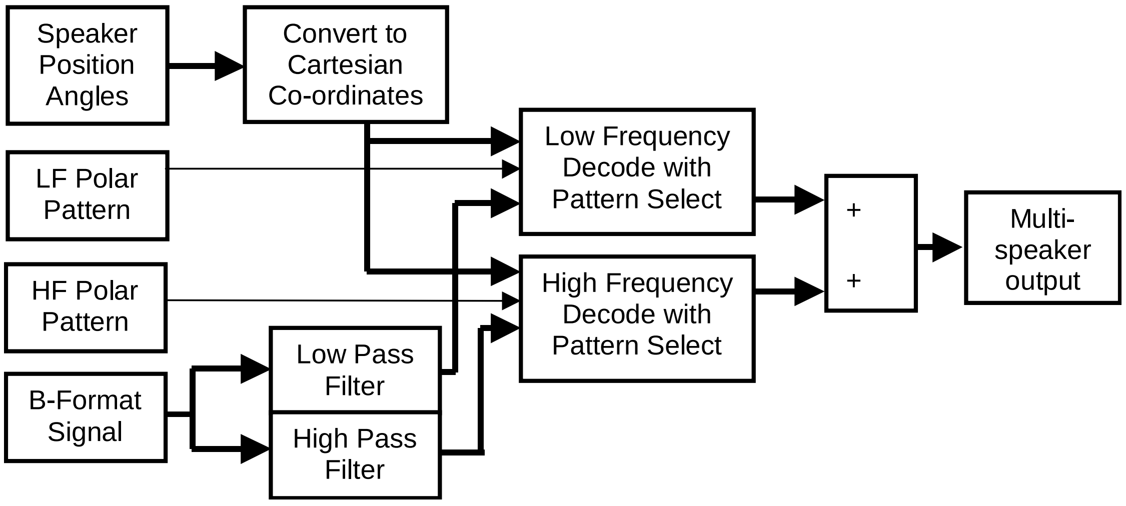


Figure 6.6 The regular array decoding problem.

Figure 6.6 shows that several parameters and settings are needed for the decoder to act upon:

- Angular position of the speakers, converted to Cartesian co-ordinates using the Ambisonic decoding equations given in equation 3.4.
- Both a low frequency and a high frequency directivity factor, as shown in Equation (3.4). It is these two parameters that set the frequency dependent decoding. For frequency independent decoding, set both parameters to the same setting (0 – 2 = omni – figure of eight).

Several functions are needed to fulfil decoding in order to minimise processing at run time. Mainly, this is carried out by the speaker position function. As the speakers are unlikely to move during system usage the Cartesian co-ordinates of the polar patterns routed to the speakers can be fixed. This means that all of the sine and cosine function calls can be made before the real-time part of the application is to be run (sine and cosine functions are very computationally expensive). A function used to calculate these decoding coefficients is shown in Table 6..

```
float ** DecoderCalc(float *fAzim, float *fElev,
                              const int NoOfSpeakers, bool Order)
{
      float **Result;
      //If 2nd Order decoder needed, 9 Rows
      if(Order)
            Result = 2DAlloc(9,NoOfSpeakers);
      //if 1st Order decoder needed, 4 Rows
      else
            Result = 2DAlloc(4,NoOfSpeakers);

      for(int i=0;i<NoOfSpeakers)
      {
            Result[0][i] = sqrt(2); //take off W offset of 0.707
            Result[1][i] = cos(fAzim[i])*cos(fElev[i]);//X
            Result[2][i] = sin(fAzim[i])*cos(fElev[i]);//Y
            Result[3][i] = sin(fElev[i]);//Z
            if(Order)
            {
                  Result[4][i] = 1.5f*sin(fElev[i])*sin(fElev[i]);//R
                  Result[5][i] = cos(fAzim[i])*sin(2*fElev[i]);//S
                  Result[6][i] = sin(fAzim[i])*sin(2*fElev[i]);//T
                  Result[7][i] = cos(2*fAzim[i])*cos(fElev[i])
                               *cos(fElev[i]);//U
                  Result[8][i] = sin(2*fAzim[i])*cos(fElev[i])
                               *cos(fElev[i]);//V
            }
      }
      //Return pointer to a two-dimensional array
      return (Result);
}
```

Table 6.3 Function used to calculate a speaker's Cartesian co-ordinates which are used in the Ambisonic decoding equations.

If the coefficients calculated in Table 6.3 are used directly then each speaker will have a cardioid response, meaning that no out-of-phase material is produced from any of the speakers (assuming a perfect, non-reverberant, B-format input captured from a perfect point source). However, it has been shown (see Chapter 5) that it can be beneficial to alter this polar response in order to make the decoder more psychoacoustically correct at different frequencies. For this, the equation shown in Equation (6.3), and discussed in Chapters 3 & 5 can be used for the final decoding.

$$S = 0.5 \times \left[(2-d) g_w W + d\left(g_x X + g_y Y + g_z Z \right) \right]$$

(6.3)

where: g_x, g_y, g_z & g_w are the speaker coefficients calculated using Table 6.3.

d is the pattern selector coefficient (from 0 – 2, omni – figure of eight).

As can be seen from Equation (6.3), it is a simple matter to include this equation in the final decoding function as it only involves a few extra multiplies per speaker, and does not use any computationally expensive sine or cosine functions. However, the decoding function is complicated slightly as a cross-over needs to be implemented using the fast convolution function given in section 6.2.2 (although, strictly speaking only phase aligned ‘shelving’ filters are actually needed, the cross-over technique using FIR filters can be used for both regular and irregular decoders, whereas the shelving filters can only be used for regular decoders). A function for carrying out an Ambisonic cross-over is shown in Table 6.4.

```
#define BLen 2049
float WOldLP[BLen],XOldLP[BLen],YOldLP[BLen];//etc.
float WOldHP[BLen],XOldHP[BLen],YOldHP[BLen];//etc.
void AmbiXOver(Ambi *Source, Ambi *Dest, SCplx *LP, SCplx *HP,
               const int order)
{
      //This exmample takes Source as the source, stores the LP
      //signal in Source, the HP signal in Dest, and takes LP and HP
      //as the frequency domain filter coefficients.
      //These original filters must be one sample less in length than
      //the buffer size

      const int Len = Source->Length;
      //copy samples
      memcopy(Source->W,Dest->W,Source->Length*4);
      memcopy(Source->X,Dest->X,Source->Length*4);
      memcopy(Source->Y,Dest->Y,Source->Length*4);
      memcopy(Source->Z,Dest->Z,Source->Length*4);
      if(Source->Order)
      {
            memcopy(Source->R,Dest->R,Source->Length*4);
            memcopy(Source->S,Dest->S,Source->Length*4);
            memcopy(Source->T,Dest->T,Source->Length*4);
            memcopy(Source->U,Dest->U,Source->Length*4);
            memcopy(Source->V,Dest->V,Source->Length*4);

            //Do second order Low pass
            OverAddFir(Source->R,LP,Len,Len-1,order,ROldLP);
            OverAddFir(Source->S,LP,Len,Len-1,order,SOldLP);
            OverAddFir(Source->T,LP,Len,Len-1,order,TOldLP);
            OverAddFir(Source->U,LP,Len,Len-1,order,UOldLP);
            OverAddFir(Source->V,LP,Len,Len-1,order,VOldLP);

            //Do second order High pass
            OverAddFir(Dest->R,HP,Len,Len-1,order,ROldHP);
            OverAddFir(Dest->S,HP,Len,Len-1,order,SOldHP);
            OverAddFir(Dest->T,HP,Len,Len-1,order,TOldHP);
            OverAddFir(Dest->U,HP,Len,Len-1,order,UOldHP);
            OverAddFir(Dest->V,HP,Len,Len-1,order,VOldHP);
      }
      //Do First order Low pass
      OverAddFir(Source->W,LP,Len,Len-1,order,WOldLP);
      OverAddFir(Source->X,LP,Len,Len-1,order,XOldLP);
      OverAddFir(Source->Y,LP,Len,Len-1,order,YOldLP);
      OverAddFir(Source->Z,LP,Len,Len-1,order,ZOldLP);

      //Do First order High pass
      OverAddFir(Dest->W,HP,Len,Len-1,order,WOldHP);
      OverAddFir(Dest->X,HP,Len,Len-1,order,XOldHP);
      OverAddFir(Dest->Y,HP,Len,Len-1,order,YOldHP);
      OverAddFir(Dest->Z,HP,Len,Len-1,order,ZOldHP);
}
```

Table 6.4 Ambisonic cross-over function

This is the comprehensive version of this function, but it can be changed depending on the application. For example, the 2nd order checking and Z signal functions can be removed for a 1st order, horizontal only, application as this will save some processing time. Now that the crossover function has been given, a regular decoding function can be developed, and is shown in Table 6.5.

```
void B2SpeakersReg(Ambi *Signal, float **Samples, float **Sp
      ,int NoOfSpeakers ,int NoOfChannels,float LPPattern
      ,float HPPattern)
{
      static float WGainLP,XGainLP,YGainLP,ZGainLP;
      static float WGainHP,XGainHP,YGainHP,ZGainHP;

      //Do XOver using global Ambi variable Signal2
      AmbiXOver(Signal, Signal2, LPCoefs, HPCoefs,Signal->Order);

      //Do loop check for both number of speakers, and number of
      //channels available on system, for testing on systems with
      //only a stereo sound card available
      for(int j=0;j<NoOfSpeakers && j<NoOfChannels;i++)
      {
            //Take pattern calculations out of loop
            //Calculate only once for each speaker
            //per buffer.
            WGainLP = 0.5f * (2-LPPattern) * Sp[0][j];
            WGainHP = 0.5f * (2-HPPattern) * Sp[0][j];
            XGainLP = 0.5f * LPPattern * Sp[1][j];
            XGainHP = 0.5f * HPPattern * Sp[1][j];
            YGainLP = 0.5f * LPPattern * Sp[2][j];
            YGainHP = 0.5f * HPPattern * Sp[2][j];
            ZGainLP = 0.5f * LPPattern * Sp[3][j];
            ZGainHP = 0.5f * HPPattern * Sp[3][j];

            for(int i=0;i<Signal->Length;i++)
            {
                  //Do Low frequency pattern adjustment and decode
                  Samples[j][i] = WGainLP * Signal->W[i]
                        + XGainLP * Signal->X[i]
                        + YGainLP * Signal->Y[i]
                        + ZGainLP * Signal->Z[i];

                  //Do High frequency pattern adjustment and decode
                  Samples[j][i] = WGainHP * Signal2->W[i]
                        + XGainHP * Signal2->X[i]
                        + YGainHP * Signal2->Y[i]
                        + ZGainHP * Signal2->Z[i];
            }
      }
}
```

Table 6.5 Function used to decode an Ambisonic signal to a regular array.

For simplicity, Table 6.5 shows only a first order example, but this function could easily be extended to include second order functionality. The two-dimensional 'Samples' array is now ready to be de-interlaced and passed back to the sound card for output.

When it comes to the decoding of an irregular array two approaches can be taken:

- Let each speaker (or speaker pair) have a user-definable pattern, decoding angle and level.
- Have each speaker use decoding coefficients directly. That is, they are supplied *after* the pattern, decoding angle and level have been taken into account.

Both of these methods are acceptable, with the first being most suited to optimising a decoder by ear and the second being most suited to using coefficients calculated using the heuristic HRTF decoding program described in Chapter 5. The latter will be slightly more efficient (although the program used to pre-calculate the coefficients could be changed to output the pattern, angle and level instead of the decoding coefficients directly).

As all of the coefficients used for decoding to irregular arrays were calculated off-line in this project (using the Tabu search algorithm described in Chapter

5), the second approach was used. The code used for this irregular decoder function is shown in Table 6.6.

```
void B2SpeakerIrreg(Ambi *Signal, float **Samples, float **SpL,
      float **SpH, int NoOfSpeakers, int NoOfChannels)
{
      static float WGainLP,XGainLP,YGainLP,ZGainLP;
      static float WGainHP,XGainHP,YGainHP,ZGainHP;
      //Do XOver using global Ambi variable Signal2
      AmbiXOver(Signal, Signal2, LPCoefs, HPCoefs, Signal->Order);

      for (int j=0;j<NoOfSpeakers && j<NoOfChannels;j++ )
      {
            //Use SpL & SpH decoding coefficients directly
            WGainLP = SpL[0][j];
            WGainHP = SpH[0][j];
            XGainLP = SpL[1][j];
            XGainHP = SpH[1][j];
            YGainLP = SpL[2][j];
            YGainHP = SpH[2][j];
            ZGainLP = SpL[3][j];
            ZGainHP = SpH[3][j];

            for (int i=0;i<Signal->Length;i++)
            {
                  //Do Low frequency pattern adjustment and decode
                  Samples[j][i] = WGainLP * Signal->W[i]
                        + XGainLP * Signal->X[i]
                        + YGainLP * Signal->Y[i]
                        + ZGainLP * Signal->Z[i];

                  //Do High frequency pattern adjustment and decode
                  Samples[j][i] = WGainHP * Signal2->W[i]
                        + XGainHP * Signal2->X[i]
                        + YGainHP * Signal2->Y[i]
                        + ZGainHP * Signal2->Z[i];
            }
      }
}
```

Table 6.6 Function used to decode an Ambisonic signal to an irregular array.

This function is very similar to the one shown in Table 6.5, except that two separate sets of speaker coefficients must be provided since they are potentially very different (not just different in polar pattern, as in a regular speaker array).

The multi-speaker array given above is possibly the most complex form of decoding as the other types (transaural multi-speaker and headphone) are based upon binaural technology and, to this end, will only need to be set up once for optimal reproduction.

As discussed in Chapter 4, in order to reproduce an Ambisonic system binaurally the separate speaker coefficients can be easily represented as a set of HRTFs with one HRTF for each of the Ambisonic signals (that is, W, X, Y etc.), or two if the rig-room-head combination are not taken to be left/right symmetrical. So, for example, a second order, horizontal only decode would be replayed binaurally using the equation shown in Equation (6.4).

$$
\begin{aligned}
Left &= (W \otimes W_{hrtf}) + (X \otimes X_{hrtf}) + (Y \otimes Y_{hrtf}) + (U \otimes U_{hrtf}) + (V \otimes V_{hrtf}) \\
Right &= (W \otimes W_{hrtf}) + (X \otimes X_{hrtf}) - (Y \otimes Y_{hrtf}) + (U \otimes U_{hrtf}) - (V \otimes V_{hrtf})
\end{aligned}
\tag{6.4}
$$

where: W, X, Y, U & V are the Ambisonic signals.

$_{hrtf}$ denotes a HRTF filter response for a particular channel.

$\otimes$ denotes convolution.

What is possibly not apparent on first inspection is that, when compared to an optimised speaker decode, a binaural simulation of an Ambisonic decoder actually requires *less* convolutions if left/right symmetry is assumed (half as many, in fact) and the same amount of convolutions if left/right symmetry is not assumed. This is due to the fact that both the crossovers and differing levels/polar patterns can be taken into account at the design time of the Ambisonic signal filters. A function used to decode a horizontal 1st order Ambisonic signal is shown in Table 6.7.

```
#define BLen 2049
#define Order 12  //FFT Length 2^12=4096
float WOld[BLen],XOld[BLen],YOld[BLen];

//Function assumes impulse length is 1 sample less than
//buffer length (i.e. 2048)
void B2Headphones(Ambi *Signal, float **Samples,
                  SCplx *WFilt, SCplx *XFilt, SCplx *Yfilt,
                  int NoOfChannels)
{
      const int Len = Signal->Length;
      OverAddFir(Signal->W,WFilt,Len,Len-1,Order,WOld);
      OverAddFir(Signal->X,XFilt,Len,Len-1,Order,XOld);
      OverAddFir(Signal->Y,YFilt,Len,Len-1,Order,YOld);

      for(int i=0;i<Len;i++)
      {
            //Left Signal
            Samples[0][i]=Signal->W[i] + Signal->X[i] + Signal->Y[i];
            //Right Signal
            Samples[1][i]=Signal->W[i] + Signal->X[i] - Signal->Y[i];
```

```
        }
        //If more than two channels were inputted and are to be
        //outputted (i.e. took B-format signal in from live
        //input) then other channels must be cleared.
        for(int i=2;i<NoOfChannels;i++)
        {
                for(int j=0;j<Len;j++)
                        Samples[i][j] = 0;
        }
}
```

Table 6.7 **Function used to decode a horizontal only, 1st order, Ambisonic signal to headphones.**

From the B2Headphones function given above, it is easy to see how this function can be extended to a two-speaker transaural representation. The block diagram for a two-speaker transaural reproduction is given in Figure 6.7.

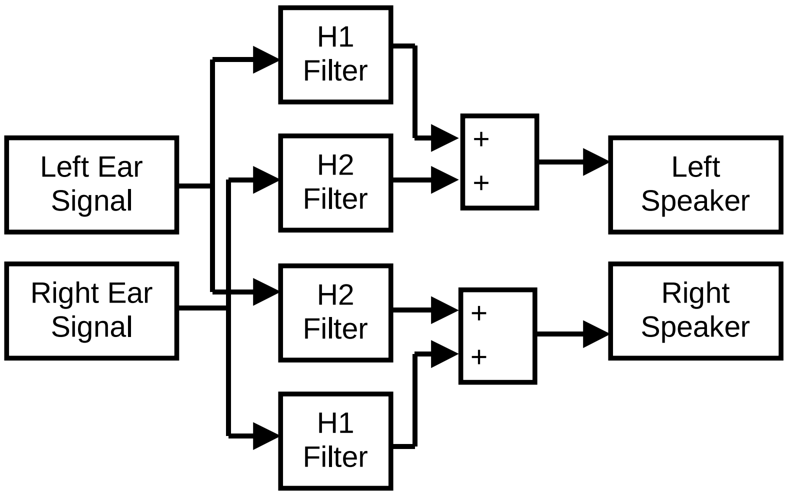


Figure 6.7 **A two-speaker transaural reproduction system.**

The method for calculating and optimising the filters needed for this arrangement were discussed in Chapter 5.

For the four-speaker version of the crosstalk cancellation not only is the above algorithm (shown in Figure 6.7) needed to be run twice, but also four signals must be provided (front left and right, and rear left and right ear signals). These can be calculated using a system very similar to the one shown in Equation (6.4), except that the front left and right HRTF filters (for the conversion to binaural) will only be taken using the gains from the front speakers, and the rear left and right HRTFs will be calculated using the gains from the rear speakers. Example sets of HRTFs for this purpose are shown in Figure 6.8 (simple, cardioid decoding, with no cross-over filtering present). These graphs show that, although the decoder is not taken as a whole, as long as the front and rear portions of the speaker rig are left/right symmetric, the same binaural simplification can be used where only one HRTF is needed for each of the Ambisonic channels. A block diagram of this four-channel crosstalk cancellation system is shown in Figure 6.9. The coding for this section is an extension of the B2Headphones function given in Table 6.7, with an extra call to a transaural function, B2Trans, given in Table 6.7.

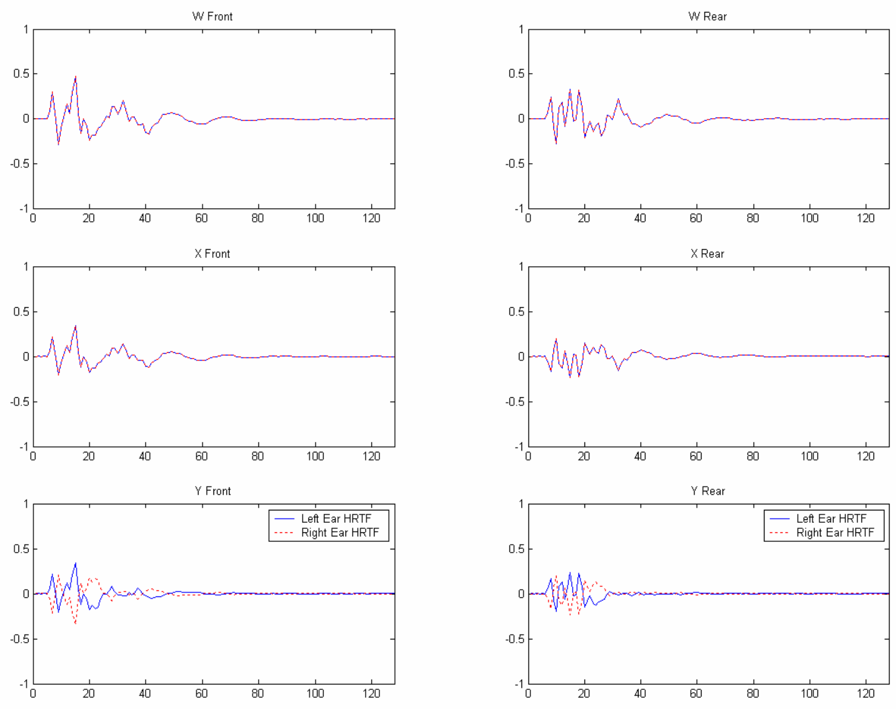


Figure 6.8 **Bank of HRTFs used for a four-channel binauralisation of an Ambisonic signal.**

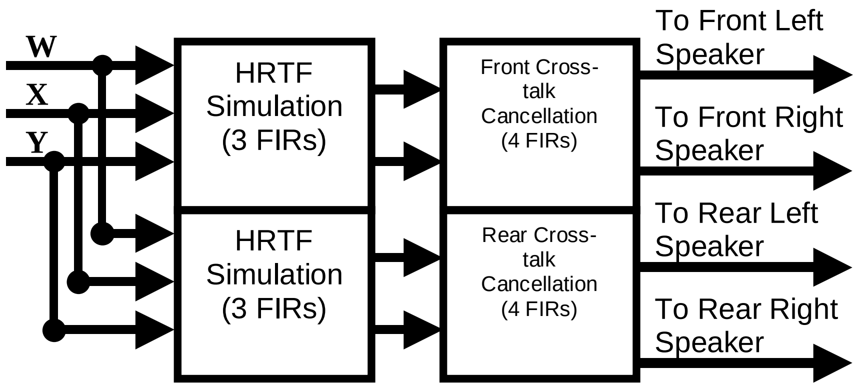


Figure 6.9 **Block digram of a four-speaker crosstalk cancellation system.**

```
#define BLen 2049
//Flag that is set for 2 and 4 speakers
//transarual reproduction.
bool Trans4;
float FLOld[BLen],FROld[BLen],FLCOld[BLen],FRCOld[BLen];
float RLOld[BLen],RROld[BLen],RLCOld[BLen],RRCOld[BLen];
```

```
void BToTrans(float **Samples,SCplx *h1, SCplx *h2,
      const int BufferLength, const int NoOfChannels)

{
      //Samples should be housing up to four channels,
      //front left, front right
      //back left, and back right binaural signals.
      static float FLCopy[BLen];
      static float FRCopy[BLen];
      memcpy(FLCopy,Samples[0],BufferLength*4);
      memcpy(FRCopy,Samples[1],BufferLength*4);

      int ChUsed=2;

      //Do 2 Speaker Transaural
      OverAddFir(Samples[0],h1,Len,Len-1,Order,FLOld);
      OverAddFir(Samples[1],h1,Len,Len-1,Order,FROld);
      OverAddFir(FLCopy,h2,Len,Len-1,Order,FLCOld);
      OverAddFir(FRCopy,h2,Len,Len-1,Order,FRCOld);

      float FL,FR;
      for (int i=0;i<BufferLength;i++)
      {
            FL = Samples[0][i];
            FR = Samples[1][i];
            Samples[0][i] = FL + FRCopy[i];
            Samples[1][i] = FR + FLCopy[i];
      }

      //Do 4 speaker transaural if flag says true
      if(Trans4 && NoOfChannels>=4)
      {
            static float RLCopy[BLen];
            static float RRCopy[BLen];
            memcpy(RLCopy,Samples[2],BufferLength*4);
            memcpy(RRCopy,Samples[3],BufferLength*4);

            OverAddFir(Samples[2],h1,Len,Len-1,Order,RLOld);
            OverAddFir(Samples[3],h1,Len,Len-1,Order,RROld);
            OverAddFir(RLCopy,h2,Len,Len-1,Order,RLCOld);
            OverAddFir(RRCopy,h2,Len,Len-1,Order,RRCOld);

            float RL,RR;
            for (int i=0;i<BufferLength;i++)
            {
                  RL = Samples[2][i];
                  RR = Samples[3][i];
                  Samples[2][i] = RL + RRCopy[i];
                  Samples[3][i] = RR + RLCopy[i];
            }
            ChUsed=4;
      }

      //Clear other output channels, ready for outputting
      for(int i=ChUsed;i<NoOfChannels;i++)
      {
            for(int j=0;j<Len;j++)
                  Samples[i][j] = 0;
      }
}
```

Table 6.8 Code used for 2 and 4 speaker transaural reproduction.

6.3 Implementation - Platform Specifics

All of the algorithmic work discussed so far in this project has been platform independent, that is, all of the functions could be implemented on any platform that supports floating point operations and standard C. However, there has to come a point where a specific platform must be chosen, and then more specialised functions are usually needed depending on the hardware/operating system used. In this project the Microsoft Windows™ operating system was used, which possesses a number of APIs for interfacing with the sound system via Windows:

- Waveform Audio (windows multi-media system)
- Direct Sound (part of the Direct X API)
- ASIO (Steinberg's sound API).

The system used in this project was the standard waveform audio system. There were a number of reasons for this:

- Waveform audio had easy support for multi-channel sound.
- *All* windows compatible sound cards had good support for this API.

Although information about the Waveform Audio API is reasonably widespread (for example, see Kientzle (1997) and Petzold (1998) Chapter 22) none give a comprehensive guide to setting up a software engine for signal processing (that is, capturing some audio live or from wave files, processing it, and outputting the processed audio). For this reason, This section of the report will give an in depth summary of how the software used in this project was structured and implemented so it can be used as a starting reference for further research to be carried out.

So, what is the Waveform Audio API? The Waveform Audio API is a layer of functions that sits between the programmer and the sound card. This means that the function calls necessary to set up and successfully run an audio application will be the same no matter what make or model of sound card the computer possesses. In this system the input and the output ports of the soundcard work seemingly independently, and so each must be taken as a separate entity and programmed for accordingly. For example, just because the output device has been set up as a 44.1 kHz, 16-bit sample stream, this does not mean that the input device will automatically take these settings when it is started. Any device activated (be it input or output) using the waveform audio API must have a number of parameters set and structures available for use. Firstly, let us examine the parameters that must be set before an output device can be started:

- Data type (for example, fixed or floating point).
- Number of Channels (for example, 1 – mono, 2 – stereo, 4, 8).
- Sample rate in Hz. (for example, 44100 or 48000).
- Bits per sample (for example, 8, 16).
- Block align – the alignment of the samples in memory (i.e. the size of the data for one sample for all of the channels, in bytes).
- Average bytes per second.
- Buffer size in bytes.

Using all of the above data, the Waveform audio API is almost ready to set up the input/output devices, however, let us first look at the block diagram of the waveform audio system as shown in Figure 6.10.

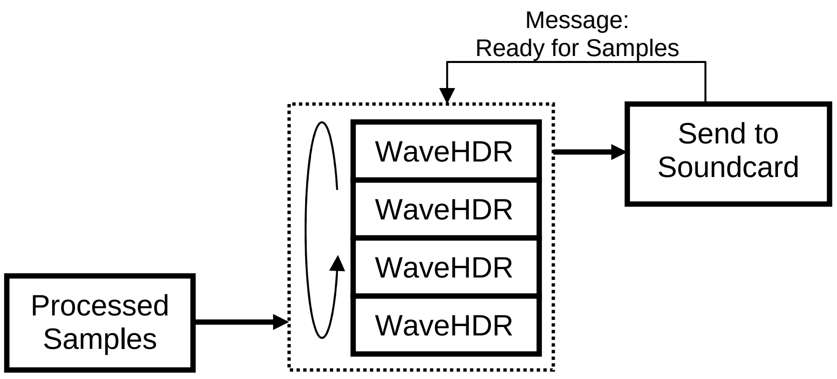


Figure 6.10 Waveform audio block diagram – Wave out.

As can be seen from this diagram, the soundcard actually informs the program when it has finished with the last buffer and is ready for the next one. This is because Windows is a message based operating system. That is, the application either passes messages, or waits to receive messages from the Windows operating system. These work in much the same way as software interrupts on a D.S.P. device, and mean that the application does not have to run in a loop, but process and send the appropriate messages in order to keep the program running. A WaveHDR is a structure that represents a buffer of audio samples, along with a few other parameters. A WaveHDR is arranged as shown in Table 6.9.

```
/* wave data block header */
typedef struct wavehdr_tag {
    LPSTR       lpData;          /* pointer to locked data buffer */
    DWORD       dwBufferLength;  /* length of data buffer */
    DWORD       dwBytesRecorded; /* used for input only */
    DWORD       dwUser;          /* for client's use */
    DWORD       dwFlags;         /* assorted flags (see defines) */
    DWORD       dwLoops;         /* loop control counter */
    struct wavehdr_tag FAR *lpNext;  /* reserved for driver */
    DWORD       reserved;        /* reserved for driver */
} WAVEHDR, *PWAVEHDR, NEAR *NPWAVEHDR, FAR *LPWAVEHDR;
```

Table 6.9 WaveHDR structure.

Of all of the various parameters available from a WaveHDR structure, only a few of them are of importance for this application. These are:

- lpData – Pointer to an array of bytes used for the storage of samples.
- dwBufferLength – Holds the length of the buffer (in bytes).
- dwFlags – Holds flags signifying that the buffer is finished with, prepared etc..

At least two of these wave headers need to be sent to either the input or output device in order for seamless audio to be heard or captured. If only one is used then an audible gap will be heard as the buffer is refilled and sent back to the device (in the case of an output device). However, as many buffers as is desired can be sent to the device, which windows will automatically store in a queue.

The other major structure that is used by the waveform audio API is WaveformatEX. This structure is used to hold nearly all of the data that must be presented to Windows in order to successfully open a device. The format of the WaveformatEX structure is given in Table 6.10.

```
/*
 *  extended waveform format structure used for all non-PCM formats.
 *  this structure is common to all non-PCM formats.
 */
typedef struct tWAVEFORMATEX
{
    WORD        wFormatTag;        /* format type */
    WORD        nChannels;         /* number of channels (i.e. mono,
                                      stereo...) */
    DWORD       nSamplesPerSec;    /* sample rate */
    DWORD        nAvgBytesPerSec;    /* for buffer estimation */
    WORD         nBlockAlign;        /* block size of data */
    WORD         wBitsPerSample;     /* number of bits per sample of
                                        mono data */
    WORD         cbSize;             /* the count in bytes of the size
                                        of extra information (after
                                        cbSize) */
} WAVEFORMATEX, *PWAVEFORMATEX, NEAR *NPWAVEFORMATEX, FAR
*LPWAVEFORMATEX;
```

Table 6.10 WaveformatEX structure.

As can be seen by the comments in Table 6.9 and Table 6.10, all of the necessary information is now potentially available for any device that is to be opened, be it an input, or an output device.

Various functions are used in the initialisation and running of a Wave device and the structures given in Table 6.9 and Table 6.10 are relied upon to provide the necessary information and memory allocation needed. Example code used to initialise a wave out device is shown in Table 6.11.

```
WAVEHDR WOutHdr[2];
WAVEFORMATEX wf;
HWAVEOUT hWaveOut;

void InitialiseWaveOut( unsigned int Device,
                        unsigned short usNoOfChannels,
                        unsigned short usSRate,
                        unsigned short usBLength)
{
      //Pass WAVEFORMATEX structure necessary data
      wf.wFormatTag      =     WAVE_FORMAT_PCM;
      wf.nChannels       =     usNoOfChannels;
      wf.nSamplesPerSec =      usSRate;
      wf.wBitsPerSample =      16;
      wf.nBlockAlign     =     wf.nChannels * wf.wBitsPerSample / 8;
      wf.nAvgBytesPerSec=      wf.nSamplesPerSec * wf.nBlockAlign;
      wf.cbSize          =     0;

      if(Device==0)
            //let windows choose device
            Device=WAVE_MAPPER;
      else
            //else, use specified device
            Device--;
      //Open wave device, specifying callback function
      //used to catch windows messages from device
      waveOutOpen(&hWaveOut,Device,&wf,(DWORD)WOCallback,
                        (DWORD)this,CALLBACK_FUNCTION);
      waveOutPause(hWaveOut);

      //Allocate memory for 2 buffers, and pass them to wave device
      for(int i=0;i<2;i++)
      {
            WOutHdr[i].dwBufferLength = usBLength * wf.wBitsPerSample
                                              * wf.nChannels/8;
            WOutHdr[i].lpData = new char[WOutHdr[i].dwBufferLength];
            WOutHdr[i].dwFlags = 0;
            WOutHdr[i].dwLoops = 0;

      waveOutPrepareHeader(hWaveOut,&WOutHdr[i],sizeof(WOutHdr[i]));
            waveOutWrite(hWaveOut,&WOutHdr[i],sizeof(WOutHdr[i]));
      }
      //Start wave out device
      waveOutRestart(hWaveOut);
}
//-------------------------------------------------------------------
void CALLBACK WaveOutCallback(HWAVEOUT hwo, UINT uMsg,
            DWORD dwInstance, DWORD dwParam1, DWORD dwParam2)
{
      switch(uMsg)
      {
            case WOM_DONE:
            {
                  //If WOM_DONE, call function used to fill buffer
                  //WAVEHDR buffer passed in to callback function
                  //as dwParam1
                  WaveOutFunc((WAVEHDR *)dwParam1);
                  break;
            }
            default:
                  break;
      }
}
```

Table 6.11 Initialisation code used to set up and start an output wave device.

As shown in Table 6.11, a call-back function must be specified in order to process the Windows' messages that are passed by the waveform audio system. For the output device the most important message is WOM_DONE. This message is passed to the call-back function every time the wave out device has finished with the WAVEHDR buffer, where a function can then be called that fills the buffer with processed samples using the processing techniques shown in Chapter 6.2 (in this case, the WaveOutFunc function is called, passing with it a WaveHdr structure).

The Wave In device is configured in much the same way by the Windows operating system, although it is interesting to note that the input and output devices are both taken to be two separate devices. To this end, no automatic connection between the two devices exists and it is the programmer that must store the input samples and then pass them to the output device (this is, of course, assuming that both input and output devices have been initialised at the same frequency, bit rate and channel numbers).

In Windows, many audio devices can be opened simultaneously, which is necessary as most multi-channel sound cards default to being configured as a number of stereo devices. However, for true multi-channel sound reproduction it is necessary to have a card that can be configured as one multi-channel device. This is due to the fact that Windows cannot open and start multiple devices at exactly the same time and, although some sound card manufacturers quote that the drivers will synchronise multiple devices, this has not been found to be the case when using their standard wave drivers. This can potentially cause problems when using such a card to feed an array of speakers used for multi-channel surround sound, as the time alignment of the output channels is assumed to be perfect. Although this artefact is not readily noticeable, it is obviously more desirable to start with a system that is as theoretically perfect as possible and so a single multi-channel device should be used, if possible. Having one multi-channel device also simplifies the processing as multiple call-back functions are not used. This effect was discovered using the Matlab add-on, Simulink. The block arrangement used to document this feature is shown in Figure 6.11.

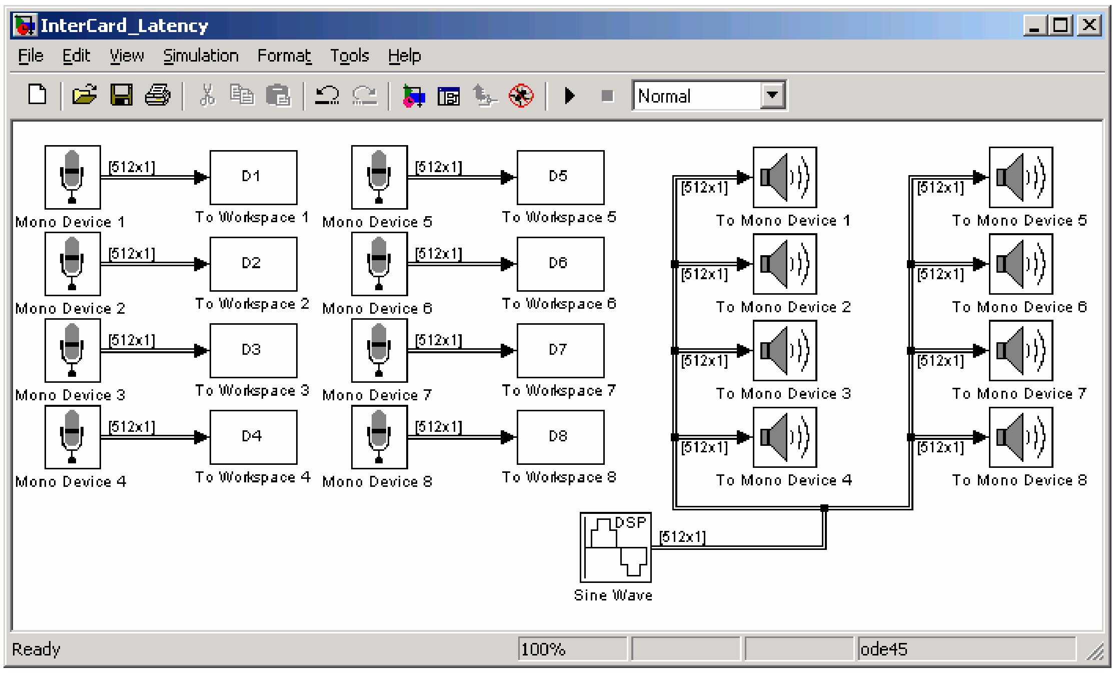


Figure 6.11 Simulink model used to measure inter-device delays

This system was used to test the latency of various devices a number of times and not only was the inter-device latency apparent, but it also changed between test runs. An example plot is shown in Figure 6.12, showing just four devices, to make the graph more readable. This variable device latency means that it is almost impossible to correct, and so a single device should be used.

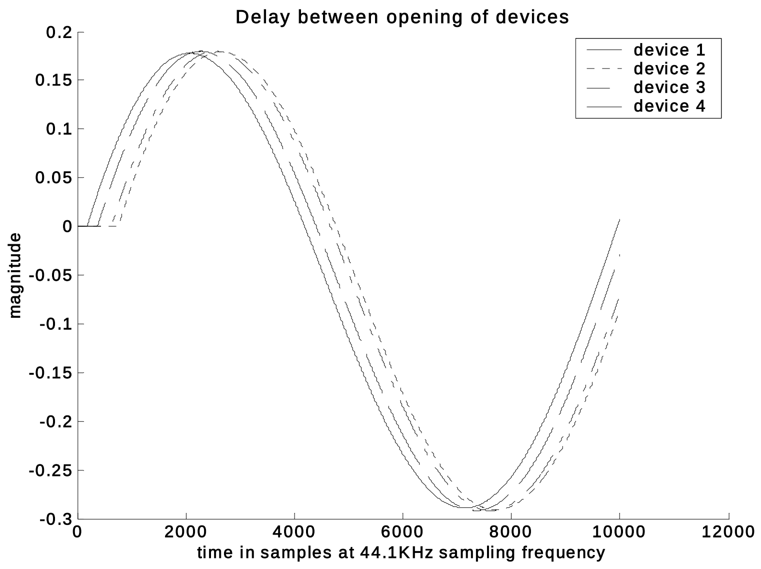


Figure 6.12 Graphical plot of the output from 4 audio devices using the Waveform audio API.

In order to successfully close an audio device, a number of API calls must be made. This is shown (for the output device) in Table 6.12.

```
void CloseDevice(UINT Device)
{
      //Reset Wave Device
      waveOutReset(hWaveOut);
      //Unlock and delete dynamic memory allocated for WAVEHDRs
      for(UINT i=0;i<NoOfBuffers;i++)
      {
            waveOutUnprepareHeader(hWaveOut,&WaveHeadersOut[i],
                        sizeof(WaveHeadersOut[i]));
            if(WaveHeadersOut[i].lpData)
                  delete [] WaveHeadersOut[i].lpData;

      }
      //Close Wave Device
      waveOutClose(hWaveOut);
}
```

Table 6.12 Closing a Wave Device

Both the opening and closing of an input wave device is identical to that of an output wave device, with the only difference being the message passed to the call-back function.

As all of this coding is Windows dependent (that is, it will never be needed for any other system), the wave device functions were encapsulated within a class. This meant that a basic 'pass-through' application could be coded, that did no processing. A new class could then be created, inheriting from this first class, but with the processing functions being redeclared so that minimal extra coding is needed for every new sound processing application that is to be written.

In order for this first class to be as flexible as possible, a signal processing function for both incoming and outgoing samples has been written. This means that the signal can be monitored (or processed) just after the input and just before the output of the audio to the soundcard.

A block diagram of the structure of this class is shown in Figure 6.13.

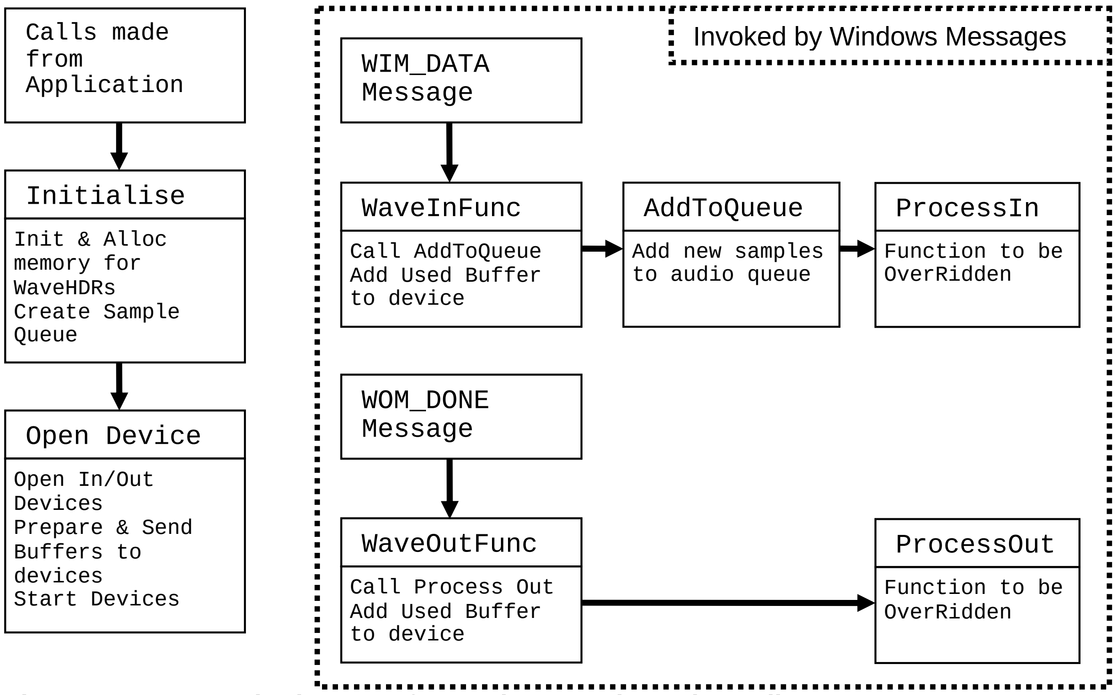

Calls made from Application
Initialise
Init & Alloc memory for WaveHDRs Create Sample Queue
Open Device
Open In/Out Devices Prepare & Send Buffers to devices Start Devices
Invoked by Windows Messages
WIM_DATA Message
WaveInFunc
Call AddToQueue Add Used Buffer to device
AddToQueue
Add new samples to audio queue
ProcessIn
Function to be OverRidden
WOM_DONE Message
WaveOutFunc
Call Process Out Add Used Buffer to device
ProcessOut
Function to be OverRidden

Figure 6.13 Block Diagram of Generic 'pass-through' Audio Template Class

It can be seen from Figure 6.13 that, apart from the initialisation and opening of the audio devices, the whole of the audio subsystem is driven by messages. The WIM_DATA message signalling that an audio buffer is ready for use (i.e. full) causes the WaveInFunc to call a function that adds this audio data to a data queue. Then, when the WOM_DONE message has been received signalling that an output buffer is ready to be filled again, the ProcessOut function is called, which is where the audio processing will be carried out on the data at the end of the audio queue, and then passed to the empty output device. An example of the overridden ProcessOut function is shown in Table 6.13. Example code for the whole of this base class can be found in the Appendix.

```
void ProcessAudio(WAVEHDR *pWaveHeader,
                  unsigned short usNoOfChannels,
                  unsigned short usBufferLengthPerChannel)
{
      //Output Callback
      //Grab pointers to in and out buffers
      short *inPtr = (short *)ReadBuffer->lpData;
      short *outPtr = (short *)pWaveHeader->lpData;

      float yn;

      for(
      unsigned int i=0;i<usBufferLengthPerChannel*usNoOfChannels;
      i+=usNoOfChannels)
      {
            //Left Channel
            yn  = (float)inPtr[i];
            //Processing Here
            outPtr[i] = (short)yn;

            //Right Channel
            yn  = (float)inPtr[i+1];
            //Processing Here
            outPtr[i+1] = (short)yn;
      }
}
```

Table 6.13 Example implementation of the ProcessAudio function for a Stereo Application.

6.4 Example Application

Using the signal processing and wave API code given above, it is now a relatively simple task to build an example signal processing application. In this research project the programming environment of Borland C++ Builder was used (Borland Software Corporation, 2003). This environment has the advantage of drag and drop development of graphical user interfaces using standard Windows components, Borland's own components or custom components based on one of Borland's component templates. This greatly simplifies the GUI creation process meaning that working, flexible applications can be coded quickly, which then makes the use of a powerful, high level language, such as C++, a valuable signal processing prototyping tool.

As stated above, applications written for the Windows operating system can be programmed using the C++ programming language. The object oriented approach lends itself well to audio programming, particularly when filtering is involved (which it generally is). This is because for each signal that needs to be filtered, separate memory locations are needed for that particular signal's feedback, feedforward, or delay line features. When coding filters in C it is the developer that must group all of this memory together, which can be cumbersome at times with different types of filters needing different memory requirements. For example, the fast convolution algorithm described in section 6.2.2 needs an additional amount of memory for each channel filtered. The size of this memory must be the same size as the FFT window size, that is, it must be larger than the size of the incoming signal. Once other types of filter are also introduced the subsequent memory requirement would soon become complicated and difficult to follow. This, on its own, is not a large problem, but means that all the memory requirements for a filter function must be clearly documented using comments, and strictly adhered to by the developer. However, in C++ a filter 'object' can be created. Inside this object, all the extra memory requirements can be hidden from the programmer with as many filter objects created as needed. This means that each filter object can be imagined as one filter device in a studio, operating on one audio stream. Initially, all the same memory requirements must be taken care of, but once implemented inside a C++ class this can then be used as a template where the developer only has access to, perhaps, a 'processaudio' function. A simple template for such a class is shown in Table 6.14.

```
class AllPass
{
private:
      float fs,fc,alpha,*Buffer;
      float ff,fb,in,out;
      const int BufLen;
      void DoAllPass(float *signal, int iLen, float aval);
public:
      AllPass(int iLen);
      ~AllPass();
      void SetCutOff(float fcut, float fsam);
      void ProcessAudio(float *signal, float dBLP, float dBHP,
                        bool dBdummy);
      void ProcessAudio(float *signal, float LinLP, float LinHP);
};
```

Table 6.14 C++ Class definition file for an allpass based shelving equalisation unit.

An object of type *AllPass* can now be initialised in the normal way in the application. However, due to the fact that the private variable *BufLen*, representing the length of an audio buffer, has been declared *constant*, this object must be initialised with an integer length (see the constructors and destructors, AllPass(int iLen) & ~AllPass()). This means that, unless the application has a fixed buffer length, the object must be declared dynamically at run time.

Looking at this object definition file further it can be seen that the developer only has access to five functions; a constructor and a destructor that are called automatically when a new AllPass object is created or destroyed, a *SetCutOff* function, and two *ProcessAudio* functions. The latter have been created in order to give this class improved flexibility, with one function making use of linear gain values, and the other making use of dB gain values. As the same function names need to have some difference in their passed values, a dummy, unused variable has been included in one of the functions to indicate that dB gains are used. Also, it can be noted that all of the variables associated with this class are declared *private*, meaning that the calling object has no access to these variables, protecting them from potential wrong doing. All of these variables are updated, as needed, by the underlying code in the class, either at initialisation, or by a public member function. This ensures that the filter is secure and as intuitive to use as possible, with the developer only having access to the functions needed, and no more.

This method was also used for the fast convolution filter, greatly simplifying the knowledge needed by the developer to use this function. The definition file is shown in Table 6.15.

```
class FastFilter
{
private:
        int order,fftsize,siglen,implen;
        float *OldArray,*Signal,*tconv,*h;
        SCplx *fh,*fSig,*fconv;
public:
        FastFilter(int FFTOrder,AnsiString *FName,int FLength);
        ~FastFilter();
        void ReLoadFilter(AnsiString *FName,int FLength);
        void OverAddFir(float *signal);
};
```

Table 6.15 C++ class definition file for the fast convolution algorithm

Again, a system very similar to that shown in the *AllPass* filter class definition file can be seen. However, if the constructor of this class is shown, it can be seen how much work is taken away from the developer when using this class, as shown in Table 6.16.

```
FastFilter::FastFilter(int FFTOrder,AnsiString *FName,int FLength)
{
        order = FFTOrder;
        fftsize = pow(2,order);
        siglen = (fftsize/2) + 1;
        implen = fftsize/2;

        OldArray = new float[fftsize];
        Signal = new float[fftsize];
        tconv = new float[fftsize];
        h = new float[fftsize];

        fh = new SCplx[fftsize];
        fSig = new SCplx[fftsize];
        fconv = new SCplx[fftsize];

        ReLoadFilter(FName,FLength);

        nspsRealFftNip(NULL,NULL,order,NSP_Init);
        nspsRealFftNip(h,fh,order,NSP_Forw);
}
```

Table 6.16 Constructor for the FastFilter class

As is immediately evident, the memory requirements of this class are complicated, with a number of memory spaces of two variable types (representing data in both the time and frequency domain) needing to be dynamically created and destroyed when necessary. Also, the size of the coefficients used in FIR filters can be large, meaning that entering them into the code is unfeasible. So, this class actually takes in a filename that contains the list of numbers used in the filter, in single precision format. This means that the filters can be quickly designed and saved to a file format in Matlab, and then tested quickly using a C++ Windows application without the need for any changes in the code of the application, meaning that recompilation is not necessary. The Matlab code used to create these files and the C++ code used to read them are shown in Table 6.17 and Table 6.18 respectively.

```
function count = savearray(array, fname);
%save array to .dat file for reading in a c program
%   for example count = savearray(array,'c:\coefs.dat');

fid = fopen(fname, 'w');
count = fwrite(fid,array,'float');
fclose(fid);
```

Table 6.17 Matlab function used to write FIR coefficients to a file.

```
#include <fstream.h>
void FastFilter::ReLoadFilter(AnsiString *FName,int FLength)
{
        FILE *f;
        int c;

        memset(OldArray,0,sizeof(float)*fftsize);
        memset(Signal,0,sizeof(float)*fftsize);
        memset(tconv,0,sizeof(float)*fftsize);
        memset(h,0,sizeof(float)*fftsize);

        memset(fh,0,sizeof(SCplx)*fftsize);
        memset(fSig,0,sizeof(SCplx)*fftsize);
        memset(fconv,0,sizeof(SCplx)*fftsize);

        f = fopen(FName->c_str(),"rb");
        if(f)
        {
                c = fread(h,sizeof(float),FLength,f);
                if(c!=FLength)
                        MessageBox(NULL,"Filter Length Error",
                                    "Filter Length Error", NULL);
                fclose(f);
        }
        else
                MessageBox(NULL,"Cannot open file",
                                    "Cannot open file", NULL);
}
```

Table 6.18 C++ code used to read in the FIR coefficients from a file.

Now the main signal processing classes have been constructed, the application can be designed. This example application was designed to test a number of the optimisation techniques discussed in Chapter 5. However, the irregular Ambisonic array testing was carried out in Simulink, and is not implemented in this application in order to keep things a little simpler. It will be capable of taking in a first order B-format signal (comprised of four wave files, as this is how most of our B-format material is archived), or one mono wave file for panning into a B-format signal. If a mono source is used, then this can be panned using a rotary dial, and if a B-format signal is used, then the sound field can be rotated using a rotary dial. The user is able to choose from four different decoding methods:

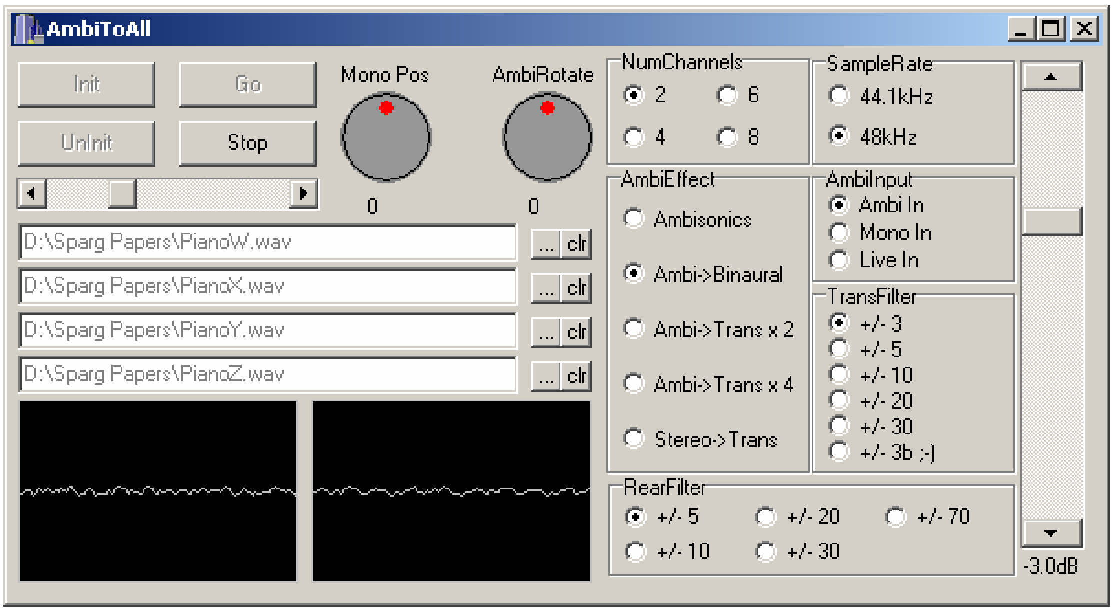
- Optimised eight speaker regular Ambisonics (using the allpass filters described above).
- Ambisonics to binaural transform (based on an eight speaker array).
- Ambisonics to two speaker transaural with speaker placements at:
    - o +/- 3^0
    - o +/- 5^0
    - o +/- 10^0
    - o +/- 20^0

    - +/- 30^0

- Ambisonics to four speaker transaural with front speaker placements as above, and rear speaker placements at:
    - +/- 5^0
    - +/- 10^0
    - +/- 20^0
    - +/- 30^0
    - +/- 70^0

In addition to these modes of reproduction, a source from the line input can also be used so that the transaural filters (two speaker algorithm) can be tested with CD material (both binaural and normal stereo). In order to utilise all of the transforms discussed above, a total of fifty six filters must be made available to the application as there must be two versions of each filter. One sampled at 44.1 kHz and another sampled at 48 kHz. This is another reason why writing these to separate data files saves time and programming effort.

To facilitate the above formats, a GUI was constructed as shown in Figure 6.14. All of the controls used are standard Windows controls, apart from the two rotary controls used for altering the mono source panning and b-format rotation. The code for the creation of the rotary controls will not be discussed here, however, but can be found in the Appendix.


AmbiToAll

Init Go Unlnit Stop

Mono Pos 0 AmbiRotate 0

D:\Sparg Papers\PianoW.wav ... clr
D:\Sparg Papers\PianoX.wav ... clr
D:\Sparg Papers\PianoY.wav ... clr
D:\Sparg Papers\PianoZ.wav ... clr

NumChannels: [✓ 2] 6 4 8

SampleRate: 44.1kHz [✓ 48kHz]

AmbiEffect: Ambisonics [✓ Ambi->Binaural] Ambi->Trans x 2 Ambi->Trans x 4 Stereo->Trans

AmbiInput: [✓ Ambi In] Mono In Live In

TransFilter: [✓ +/- 3] +/- 5 +/- 10 +/- 20 +/- 30 +/- 3b ;-)

RearFilter: [✓ +/- 5] +/- 20 +/- 70 +/- 10 +/- 30

-3.0dB

Figure 6.14 Screen shot of simple audio processing application GUI.

In the audio subsystem class, there are two main tasks to be carried out:

- Initialisation/deinitialisaton of filter structures and graphical oscilloscope.
- Process audio function.

In order for to avoid storing fifty six FIR filters in memory at once (and, for that matter, having to manage fifty six FIR filter structures in the program code), only the filters currently available for use will be stored in memory. These are:

- 3 Allpass filters for the eight speaker Ambisonic decoder.
- 3 FIR filters for Ambi to two ear binaural processing
- 6 FIR filters for Ambi to four ear binaural processing
- 4 FIR filters for binaural to two speaker transaural processing
- 4 FIR filters for binaural to four speaker transaural processing (8 used in this algorithm in total).

It is only the crosstalk cancellation filters that need to be updated in real time, and so, in order to facilitate this, the GUI sets a flag to true whenever a filter needs changing (that is, the transfilter and rear filter radio boxes are changed). The audio subsystem checks this flag at the start of every audio buffer and, if set, reloads the appropriate filter from disk.

A block diagram of the audio function for this application is shown in Figure 6.15.

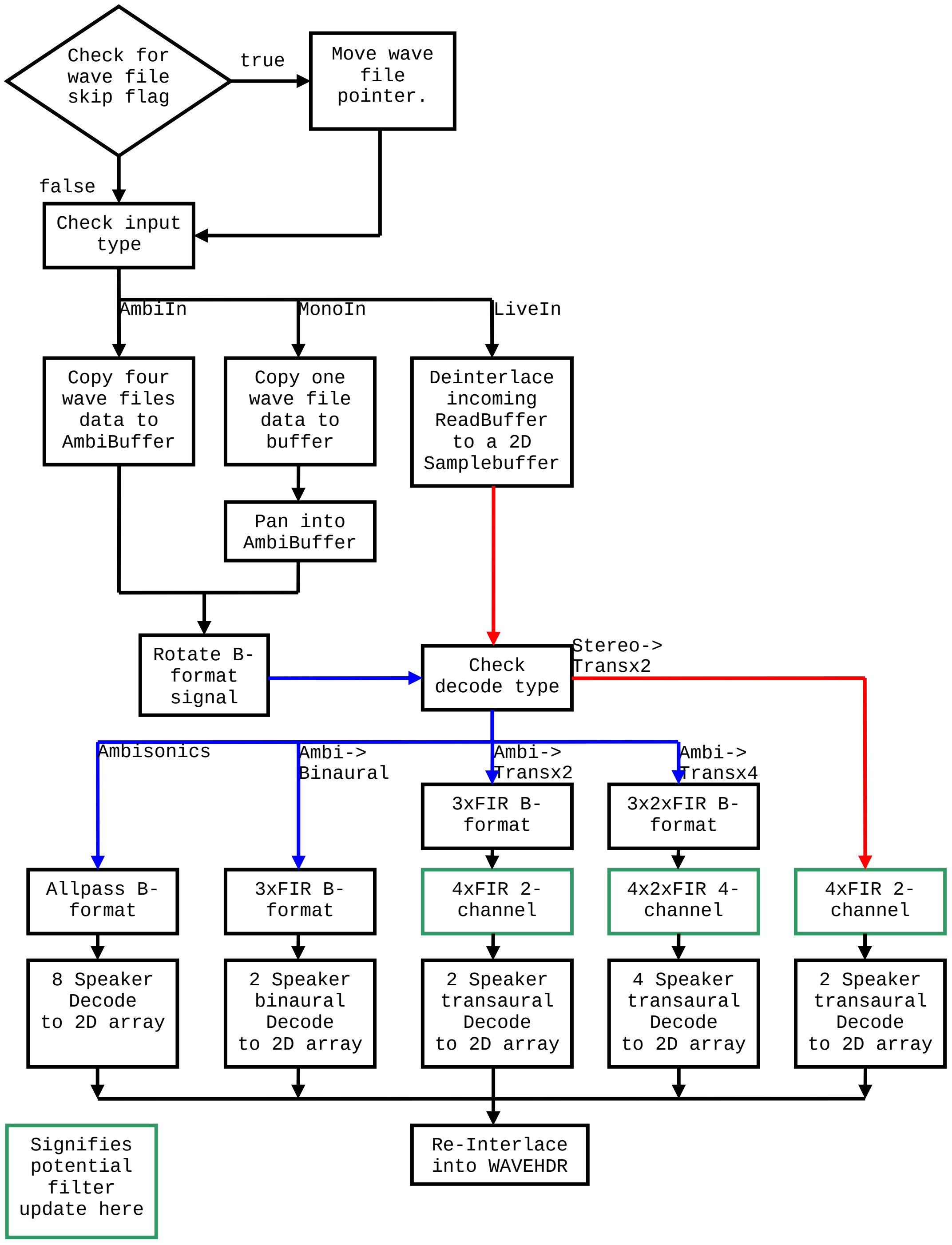

Check for wave file skip flag
true
Move wave file pointer.
false
Check input type
AmbiIn
MonoIn
LiveIn
Copy four wave files data to AmbiBuffer
Copy one wave file data to buffer
Deinterlace incoming ReadBuffer to a 2D Samplebuffer
Pan into AmbiBuffer
Rotate B-format signal
Check decode type
Stereo->Transx2
Ambisonics
Ambi->Binaural
Ambi->Transx2
Ambi->Transx4
3xFIR B-format
3x2xFIR B-format
Allpass B-format
3xFIR B-format
4xFIR 2-channel
4x2xFIR 4-channel
4xFIR 2-channel
8 Speaker Decode to 2D array
2 Speaker binaural Decode to 2D array
2 Speaker transaural Decode to 2D array
4 Speaker transaural Decode to 2D array
2 Speaker transaural Decode to 2D array
Re-Interlace into WAVEHDR
Signifies potential filter update here

Figure 6.15 Block diagram of the applications audio processing function.

The audio processing function is simplified because all of the various processing algorithms are carried out in separate objects/functions, making the coding a simpler task, as each function can be taken in isolation. So, for the final section of coding needed for this example application, the decoder type switch statement and code is shown in Table 6.19.

```
switch(Window->m_effect)
{
      case 0:     //8 Speaker Ambisonics
```

```
                WAP->ProcessAudio(ABuf->W,1.33,1.15);
                XAP->ProcessAudio(ABuf->X,1.33,1.15);
                YAP->ProcessAudio(ABuf->Y,1.33,1.15);
                B2Speakers(Decode,ABuf,Samples,usNoOfChannels,8,0);
                break;
        case 1:     //Ambisonics to Binaural
                B2Headphones(ABuf,Samples,usNoOfChannels);
                break;
        case 2:     //Ambisonics to Binaural to Transaural x 2
                if(UpdateFilter)
                {
                                ChooseFilter(SampleRate);
                                UpdateFilter = false;
                }
                B2Headphones(ABuf,Samples,usNoOfChannels);
                B2Trans(ABuf,Samples[0],Samples[1],
                                usNoOfChannels,h1fl,h2fl,h1fr,h2fr);
                break;
        case 3:     //Ambisonics to Binaural x 2 to Transaural x 4
                if(UpdateFilter)
                {
                                ChooseFilter(SampleRate);
                                UpdateFilter = false;
                }
                if(UpdateRearFilter)
                {
                                ChooseRearFilter(SampleRate);
                                UpdateRearFilter = false;
                }
                B2Headphones4(ABuf,BBuf,Samples,usNoOfChannels);
                B2Trans(ABuf,Samples[0],Samples[1],
                                usNoOfChannels,h1fl,h2fl,h1fr,h2fr);
                if(usNoOfChannels>=4)
                        B2Trans(ABuf,Samples[2],Samples[3],
                                usNoOfChannels,h1rl,h2rl,h1rr,h2rr);
                break;
        case 4:     //Live input to Transaural x 2
                if(UpdateFilter)
                {
                                ChooseFilter(SampleRate);
                                UpdateFilter = false;
                }
                B2Trans(ABuf,Samples[0],Samples[1],
                                usNoOfChannels,h1fl,h2fl,h1fr,h2fr);
                break;
        default:    //if none of the above
                B2Speakers(Decode,ABuf,Samples,usNoOfChannels,8,0);
                break;
}
```

Table 6.19 Decoding switch statement in the example application

To look at the code in its entirety, this example application is given in the Appendix.

6.5 Conclusions

Writing the application in this modular fashion makes the potentially complex audio processing function much easier to manage and change, if necessary, and has resulted in a large library of functions and classes that can be used to create a working multi-channel audio application very quickly.

Due to the use of the fast convolution algorithm, and the utilisation of the Intel Signal Processing Library (although Intel have now discontinued this, and the Intel Integrated Performance Primitives must be used instead (Intel, 2003b)), the implemented surround sound system will run on Intel Pentium II processors and faster, even when decoding to eight or more speakers.

Most of the Ambisonic algorithmic testing was carried out in Matlab and Simulink, but regarding sound quality, the software libraries described in this Chapter work well and without audio glitches. It must also be noted that using custom C software was the only way to test and evaluate the Transaural or Binaural decoders in real-time due to the lack of a real-time (that is, frame based) overlap add convolution function in Simulink, so this software was invaluable in the rapid evaluation and testing of the crosstalk cancellation filters described in Chapter 5.

Chapter 7 - Conclusions

7.1 Introduction

This thesis has identified the following problems with the current state of surround sound systems (as described in Section 3.4):

1. Although Gerzon and Barton (1992) suggested a number of optimisation equations for use with irregular speaker arrangements, the equations are difficult to solve, and so no further research seems to have been carried out in this area.
2. At least four speakers must be used to decode a horizontal 1st order signal, and six speakers must be used to decode a horizontal 2nd order system and although the conversion to binaural has been done by McKeag & McGrath (1996) initially, and later by Noisternig *et al.* (2003), none of this work takes into account the correct presentation of the lateralisation parameters which has been addressed in point 1, above.
3. Only a handful of software utilities for the encoding and decoding of Ambisonic material are available (McGriffy, 2002), and no psychoacoustically correct decoding software for irregular arrays exists.

These problems have been addressed in this research as follows:

1. A method of solving the equations given by Gerzon and Barton (1992) has been demonstrated that simplifies the design of Ambisonic decoders for irregular speaker arrangements using the velocity and energy vector criterion as described by Gerzon & Barton (1992) which also corrects the problem of low and high frequency decoder discrepancies as shown in section 5.3.
2. Also, a new method of HRTF analysis has been developed in order to differentiate between decoders designed using the method described in point 1, above. This data has then been utilised directly in the design of multi-channel decoders. This form of decoder is not strictly Ambisonic, as it does not conform to the Ambisonic definition as described by Gerzon & Barton (1998) and described in section 3.3.1, but will allow for the further optimisation of the B-Format decoding

process than is possible using the original velocity/energy vector theory (i.e. more frequency bands can be used).

3. The use of B-format and higher order Ambisonic encoded signals as a carrier format for Binaural and Transaural reproduction systems has been demonstrated. The optimisation of both Binaural and Transaural techniques through the use of inverse filtering has been formulated, with the transaural reproduction technique benefiting particularly from this technique. Also, a new Ambisonic to four speaker Transaural decode has been formulated and discussed, although sound quality issues have hindered this work, possibly due to the HRTF set used in this research, and so work in this area is still ongoing.
4. Software utilities have been implemented for both the design of decoders for irregular speaker arrays, and the replaying of the Ambisonic carrier signal over:
    a. Headphones
    b. Two or four speaker Transaural
    c. Multi-speaker, optimised, Ambisonic arrays.

The details of these achievements are discussed below.

7.2 Ambisonics Algorithm development

This project has concentrated on the decoding of a hierarchical based surround sound format based on the Ambisonic system.

The traditional method of analysing and optimising Ambisonic decoders is through the use of the energy and velocity vector theories. The algorithmic development in this report, in the most part, has been centred on the use of HRTF data in order to analyse and optimise the performance of the Ambisonic decoders directly. This form of analysis was shown, in Chapter 5, to give results that backed up the original energy and velocity vector theory.

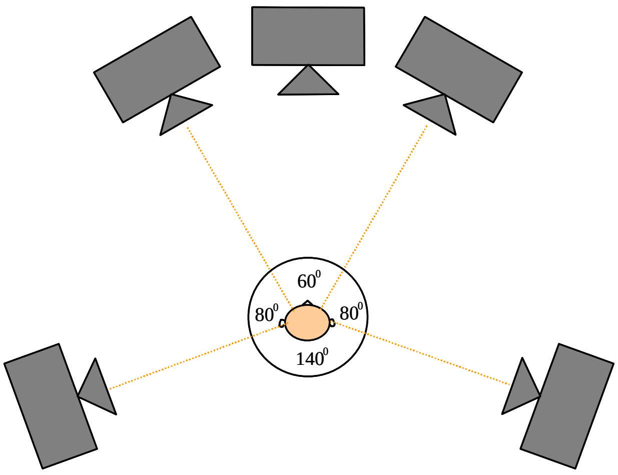


Figure 7.1 **Recommended loudspeaker layout, as specified by the ITU.**

That is, if an Ambisonic decoder was optimised using the energy and velocity vectors, then this result also gave a very good match when analysed using the HRTF method. A number of interesting observations were made from this experiment:

- Although a standard ITU five speaker arrangement was used (as shown in Figure 7.1) in the analysis and optimisation stages, the velocity vector analysis gave a perfect low frequency match for the decoder, as shown in Figure 7.2. This was surprising as there is such a large speaker 'hole' at the rear of the rig.
- However, the HRTF analysis showed some error in the rear of the sound fields reproduction, which seems to show a more realistic result, as demonstrated in Figure 7.3.

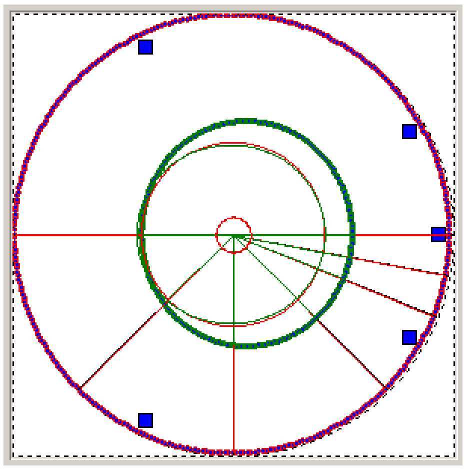


Figure 7.2 **Low frequency (in red) and high frequency (in green) analysis of an optimised Ambisonic decode for the ITU five speaker layout.**

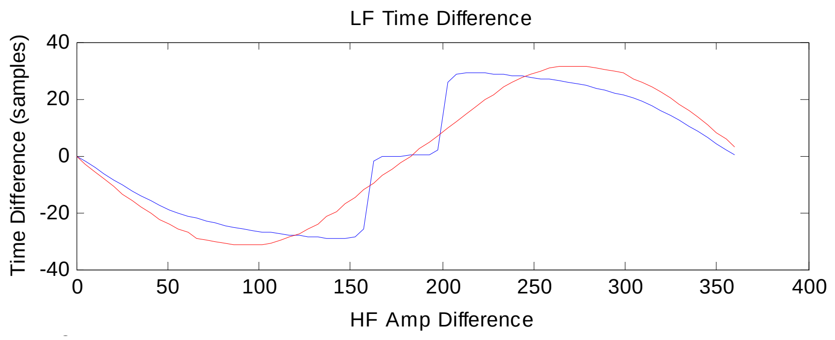
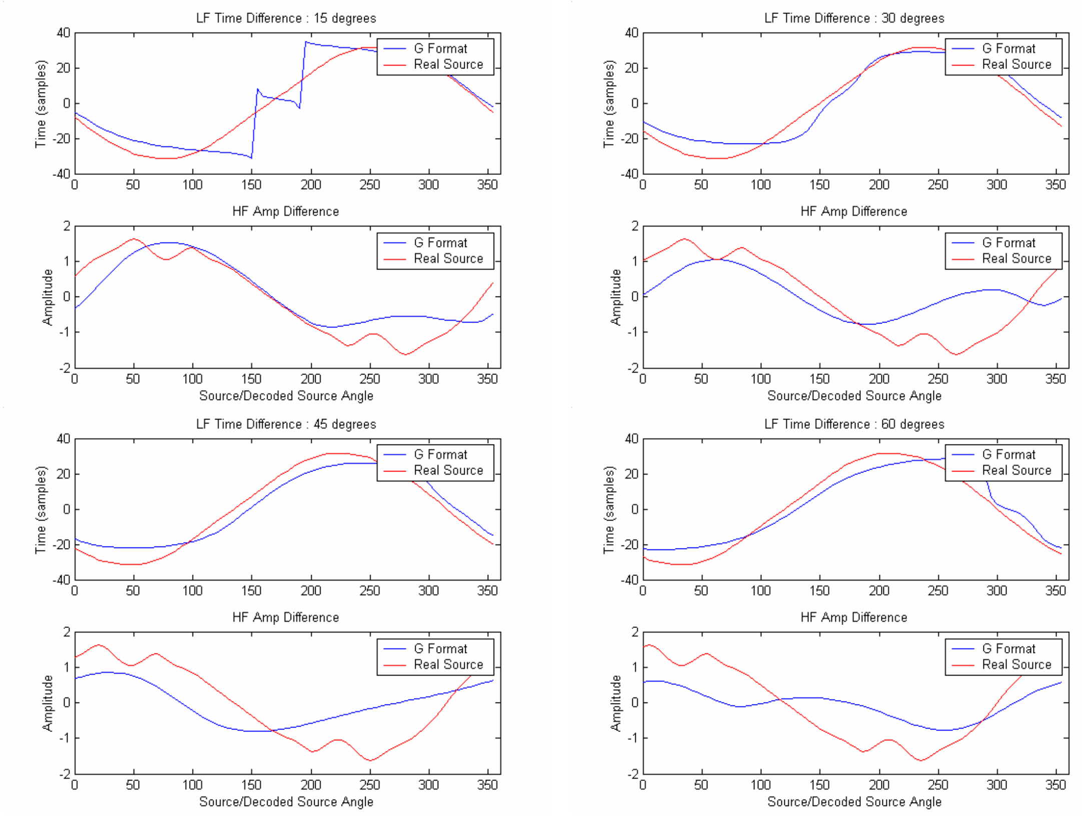


Figure 7.3 **A graph showing a real source's (in red) and a low frequency decoded source's (in blue) inter aural time differences.**

Also, a number of benefits were found due to the inherent increased flexibility of the HRTF analysis technique when compared to the analysis using the energy and velocity vectors. Using the HRTF technique, the effect of head movements could be analysed in a quantitative manner. This can prove invaluable when trying to differentiate between a number of potentially optimal sets of decoder coefficients, and significant differences can be observed. For example, see Figure 7.4 which shows a comparison between two sets of optimised decoder coefficients (using energy and velocity vector theory) and their analytical performance under head rotation. One prominent feature of Figure 7.4 can be seen if the low frequency time difference plots for a source at 0^0 are observed. The second coefficients response to head rotation shows that the time difference stays at roughly zero samples no matter what direction the listener is facing, indicating that the source is tracking with the listener. However, the first coefficients low frequency graphs shows that the time difference of a source at 0^0 changes in the same way as a real source would, that is, the source does not track with the listener and more correct cues are presented.

Coefficient Set 1



Coefficient Set 2

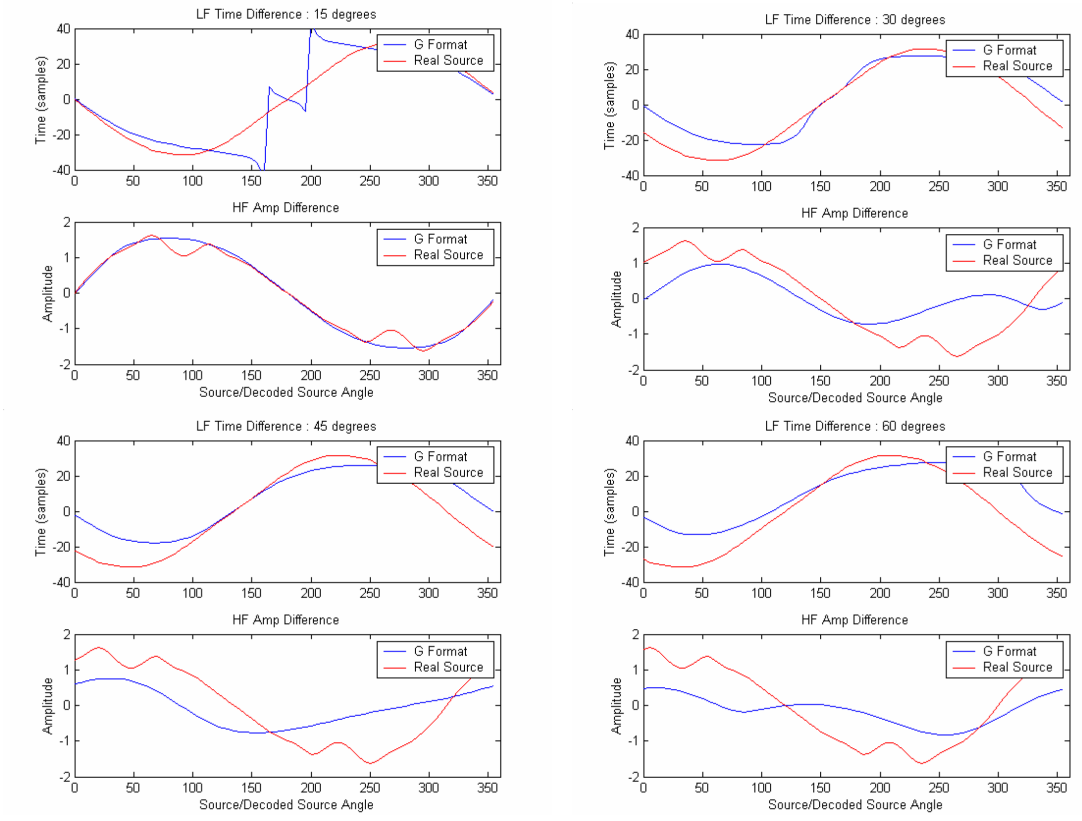


Figure 7.4 HRTF Simulation of head movement using two sets of decoder coefficients.

Such observed variations between different decoders' analytical performance can give more indications as to how well the decoder will perform than previous techniques allow.

Although the Vienna decoding optimisation technique (using the velocity and energy vectors) was proposed in 1992, very little (if any) Vienna decoders have been calculated and used, mainly due to both the mathematical complexity in deriving decoder coefficients using this method and the fact that Gerzon's paper gave results for a speaker layout very different from the ITU standard, which was proposed after this paper's publication.

To this end, software based on a Tabu search algorithm was developed that, once the five speaker positions were entered, would calculate optimised decoders automatically. This heuristic mechanism has proved a valuable tool, and once the program was written to optimise decoders using the Vienna equations, it could easily be adapted to use the HRTF method, both with and without head-turning considerations.

A limited set of formal listening tests have been carried out on a number of decoders optimised using the two techniques described above, as a precursor to further research in this area. Two tests were carried out:

1. Perceived localisation of a panned, dry, source.
2. Decoder preference when listening to an excerpt of a reverberant recording.

Although a very small test base was used, decoders optimised using both energy/velocity vectors and HRTF data directly, via the Tabu search algorithm, were shown to outperform the reference decoder in both tests. The best performing decoder in test 1 was an expected result, after observing the performance of the decoder using HRTF data. However, the decoder that was chosen unanimously as the preferred choice when auditioning pre-recorded material was not as easy to predict. Reasons for this may be:

1. The most accurate decoder may not be the one that actually sounds best, when replaying pre-recorded material, and will be material dependant.

2. It was noticed that the two best (analytically speaking) performing optimised decoders exhibited a slightly uncomfortable, in-head, sound when auditioned in the sweet-spot, which was not apparent with the preferred decoder. This effect disappeared when the listener moved slightly off-centre.

This result suggests that when designing decoders artistically, rather than for spatial accuracy, other parameters may need to be taken into account or be available to the user so intuitive control of the decoder can be carried out in order to alter the spatial attributes of the presentation (such as spaciousness and perceived depth, for example).

Overall the tests were encouraging and showed that the Ambisonic technique can reproduce phantom images both to the side and behind the listener. However, a much larger test base should be used to further test the new decoders, along with more source positions, due to the reasonably subtle differences between the decoders used in this test (especially for test 1).

It has also been shown how this software can be adapted to optimise higher order decoders for irregular arrays, as described by Craven (2003) and two decoders for such a system (using 4th order circular harmonics) are shown below. One suggested by Craven (2003) and another optimised using the Tabu search methodology described above.

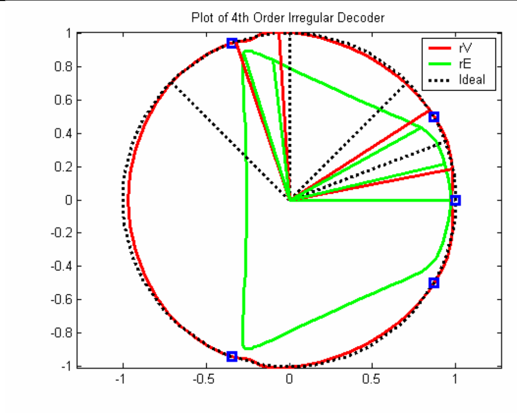
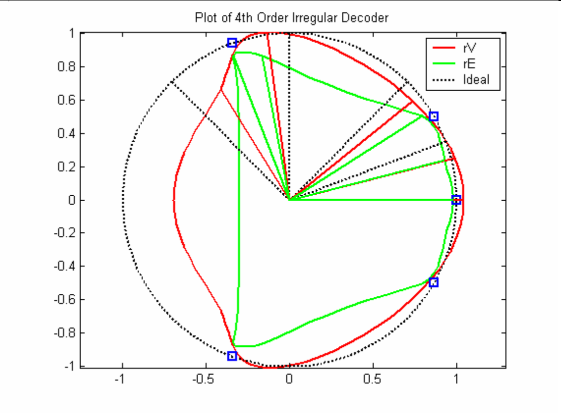
Decoder optimised using Tabu Search	Decoder proposed by Craven (2003)
	

Figure 7.5 **Energy and Velocity vector analysis of two 4th order, frequency independent decoders for an ITU five speaker array. The proposed Tabu search's optimal performance with respect to low frequency vector length and high/low frequency matching of source position can be seen clearly.**

7.2.1 Further Work

This project has raised a number of questions and results that require future work:

1. Altering the coefficients of decoders (i.e. their virtual microphone patterns) can drastically alter how reverberant a recording is perceived to be (as well as altering other spatial attributes). This is probably related to the amount of anti-phase components being reproduced from speakers, but needs further work to the relationship between more complex spatial attributes and decoder coefficients can be formulated..
2. The uncomfortable, 'in-head' perception reported by the listening test subjects when listening to pre-recorded material requires further work which could be coupled into a study of how optimising decoders affects its off-centre performance.
3. Altering the optimisation criterion to take into account off-centre positions could be investigated so determine whether the sweet area of the system can be increased.
4. A study of the higher order decoders, such as the one proposed by Craven (2003), or decoders optimised using the Tabu search method, as described in section 5.3.4, in order to evaluate what effect higher order components have, and whether an upper limit, with respect to harmonic order, can be judged.

7.3 Binaural and Transaural Algorithm Development

7.3.1 B-format to Binaural Conversion

The main optimisation method employed using the decoding technologies based on binaural techniques is that of inverse filtering. This is needed for the HRTF set used in this report due to the noticeable colouration of the sound perceived when these HRTFs are used. The inverse filtering technique works well in improving the quality of these filters, while maintaining their performance, as the differences between the ears remain the same and the pinna filtering is likely to be incorrect when compared to that of a listener's (in fact, the likelihood of the pinna filtering being the same is extremely slim, if not impossible). However, the B-format HRTFs created (see Figure 7.6) do give the impression of a more spatial headphone reproduction, when compared to listening in conventional stereo, even though these are the anechoic forms of the filters. This is especially true when listening to sounds recorded in reverberant fields as the ear/brain system will now receive more coherent cues than when mixing the B-format to it's stereo equivalent (which is based on mid and side microphone signals and relies on the crosstalk between the ears which is destroyed using headphones – see section 3.2.2 on Blumlein Stereo for more details). Two recordings have been obtained from the company Serendipity (2000) where recordings of the musicians were made in Lincoln Cathedral using both a SoundField microphone and a binaural, in-ear system, simultaneously. Although the binaural recording was not from the same position (it was carried out by Dallas Simpson, a binaural sound artist, who tends to move around often during recordings for artistic effect), a qualitative comparison of the spatial qualities of the two recordings could be made over headphones.

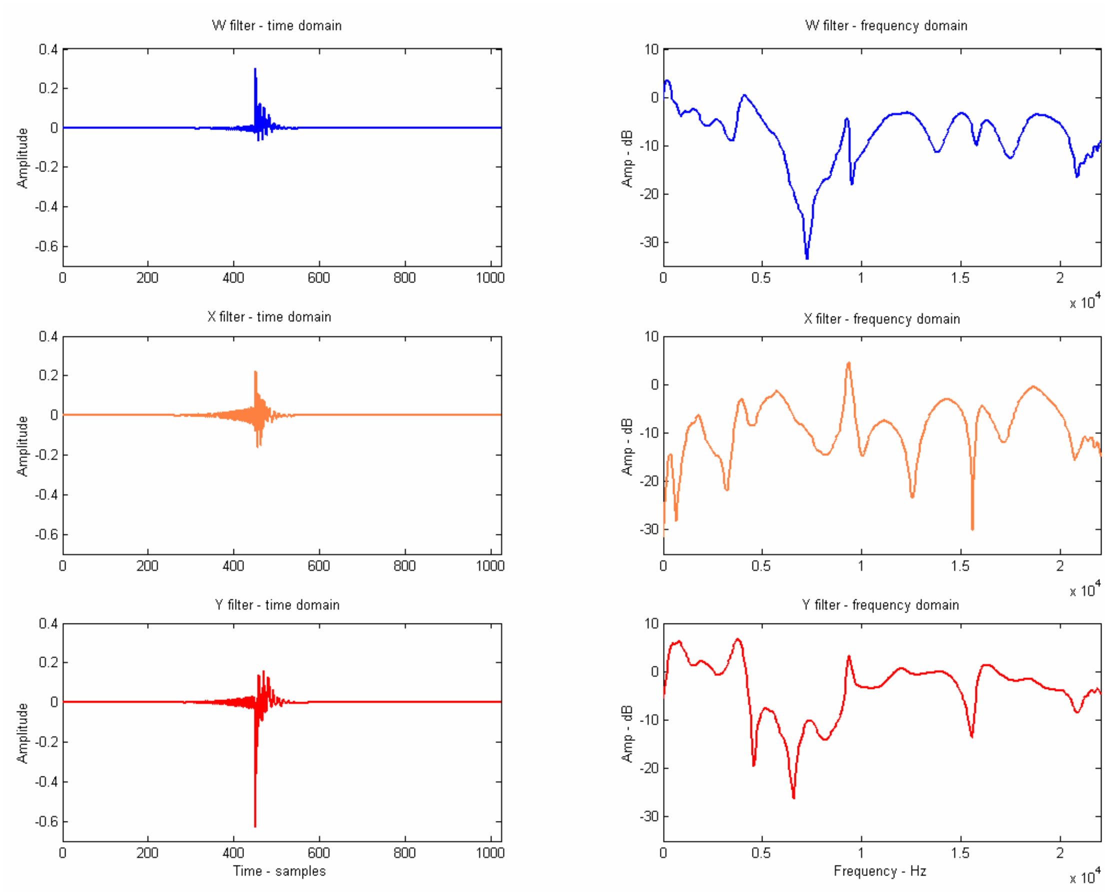


Figure 7.6 **B-format HRTF filters used for conversion from B-format to binaural decoder.**

This confirmed that the B-format to binaural system seems to perform favourably when compared to the plain binaural system, although good out of head effects are still difficult to achieve with both recordings. This is not due to algorithmic errors, but to the fact that the ear/brain system isn't receiving enough coherent cues, and it is interesting as the work by Lake (McKeag & McGrath, 1997) has shown that out of head images are possible using headphones alone. However, they do restrict themselves to recording the impulses of 'good' listening rooms for this purpose, with their large hall impulse responses seeming no more out-of-head than their smaller room impulses (Lake DSP, 1997).

7.3.2 Binaural to Two Speaker Transaural

Once the B-format to binaural transform has been executed, the resulting two channels can then be played over a transaural reproduction system, employing the filter design techniques outlined and discussed in Chapter 5. The inverse filtered crosstalk cancellation filters perform better when auditioning standard binaural material when compared to binauralised B-Format material, with colouration of the sound being noticeable when replying B-Format in this way, although the colouration is not noticeable when auditioning either the B-Format to binaural, or binaural to crosstalk cancelled material in isolation.

As mentioned in Chapter 5, pinna errors seem to worsen the system's accuracy and, to this end, the Ambiophonics system employs a pinna-less dummy head in the calculation of the inverse filters for the crosstalk cancellation, and in the recording of the event itself (Glasgal, 2001).

7.3.3 Binaural to Four Speaker Transaural

The binaural to four speaker transaural system has an interesting effect. The testing of this system has mainly been on the front and rear pair of a standard 5.1 setup as this speaker array is readily available for quick testing (that is, speakers at +/- 30^0 and +/- 110^0). The B-format to four speaker binaural filters are shown in Figure 7.7 where an overall level difference can be seen between the two sets of filters. This is due to the front decode containing the combined response of five speakers and the rear decode containing only the combined response of three, which is due to the virtual speakers at +/- 90^0 being assigned to the front hemisphere decoder (a regular eight speaker array was simulated).

When carrying out A/B comparisons between the two speaker and four speaker systems (note, that the sound colouration problems mentioned above are still present), a number of points are noticeable:

- The four speaker crosstalk cancelled decode produces images further away from the listener.
- The four speaker decode also has a more open, surrounding sound (as one would expect from adding the rear speakers).
- The localisation seems slightly clearer and more precise (although this seems to be a little dependent on the type of material used in testing).

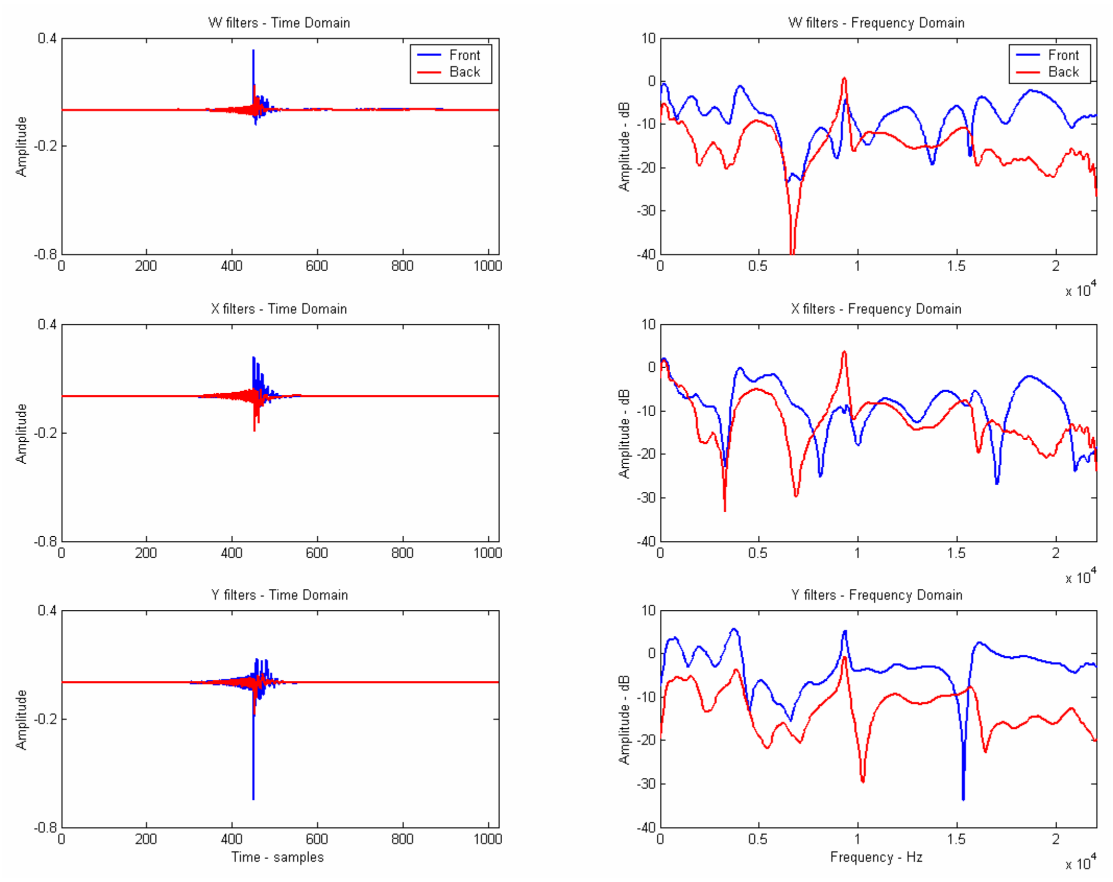


Figure 7.7 B-format HRTF filters used for conversion from B-format to binaural decoder.

Much of this is probably due to the increase in localisation cue consistency associated with splitting the front and rear portions of the decode and reproducing this from the correct portion of the listening room (that is, the rear speaker feeds come from behind and the front portion of the decode comes from in front), although the ‘moving back’ of the material is an interesting effect: it is not yet certain whether it is a ‘moving back’ of the sound stage or a more realistic sense of depth that is being perceived. It must also be noticed that this effect only occurs when the rear speakers are engaged. That is, it is not noticed when just changing the front pair of speakers’ filters from five to eight speaker virtual decodes, meaning that it is not due to the ‘folding back’ of the rear speakers into the frontal hemisphere in the two speaker, eight virtual speaker, decode. It should also be noted that because the Ambisonic system is designed so that the sum of the speaker outputs at the ear of the listener (in the centre of the array) produce the correct psychoacoustic cues (as far as is possible), this makes it particularly suited to the binaural/transaural playback system, as this should make the system less dependent on the quality of the actual speaker simulation. This is in contrast to the simulation of the five speakers of the 5.1 system over headphones (such as the Lake developed Dolby Headphones system (Lake DSP, 1997)).

One other promising feature of the four speaker crosstalk cancellation system is that if the speaker span described above is used (+/- 30^0 and +/- 110^0), although the most ‘correct’ listening experience is found in the middle of the rig, the system still produces imaging outside of this area. This is in contrast to the single +/- 3^0 speaker placement that, although possessing very good imaging in the sweet area, has virtually no imaging off this line. This would make this setup more desirable for home use where other listeners could still get a reasonable approximation to the sound field, but with the central listener experiencing an improved version. However, it must also be noted that, as mentioned in chapter 5, the virtual imaging of the filters created for +/- 30^0 is not as accurate as those created for a smaller span (such as +/- 3^0), although its frequency response does not lack (or boost depending on the level of inverse filtering used) lower frequencies as much.

7.3.4 Further Work

A number of optimisations have been suggested for the crosstalk cancellation system, where much less work has been carried out when compared to standard binaural audio reproduction systems, mostly striving for the minimisation of the use of the regularisation parameter as described by Kirkby *et al.* (1999) and Farina *et al.* (2001). This is because, although regularisation accounts for any ill-conditioning that the system may possess, it is at the expense of crosstalk cancellation accuracy. This can have the effect of the images pulling towards the speakers at these frequencies (Kirkby *et al*, 1999). In this report a number of inverse filtering steps were taken where single inversion was used to reduce regularisation, and double inversion used to remove the need for regularisation completely. However, this has the effect of altering the frequency response of the crosstalk cancelled system quite noticeably when the speakers are set up in an optimum configuration (that is, closely spaced). Nevertheless, this is still not the whole picture. The single inverted filters show (mathematically speaking) that no bass boost is perceived by the listener, although it is noticed in reality, and the double inverse filtering takes away too much bass response. A filter part way between these two extremes is needed, and this is the next step in the development of the crosstalk cancellation filter structures. Also, much work is still needed in how it is that the listener actually perceives the sound stage of a crosstalk cancelled system as a number of interesting ‘features’ have been noted during informal listening tests.

- When listening to straight binaural pieces (where the crosstalk cancellation system still works best), good distance perception is apparent, with sources able to appear closer and further away than the speakers actually are.
- Room reflections can have an interesting effect on the playback. If the two speakers are against the wall, then the perceived material is, for the most part (see above), located in a semi-circle around the front of the listener. However, if the speakers are moved inwards, then the material is generally still perceived towards the back of the room. In this way, it is as if the room is superimposed onto the recorded material.

These are two situations that need further investigation, as they may hold more clues as to our distance perception models, one attribute that can be difficult to synthesise in audio presentations.

Overall, it is the original Ambisonic system that sounds the most natural, although much of this could be attributed to the filters used in the HRTF processing. With filters recorded in a non-anechoic room and a better speaker/microphone combination it may be possible to achieve a more out-of-head experience, especially if accompanied with some form of head-tracking, where the rotation could be carried out using a standard B-format transformation, removing the need for complex dynamic filter changing in real-time (where careful interpolation is needed to eliminate audible artefacts when moving between the different HRTF filter structures) as recently demonstrated by Noisternig *et al* (2003).

Chapter 8 - References

Alexander, R.C. (1997) *Chapter Three – The Audio Patents.* Retrieved: May, 2003, from http://www.doramusic.com/chapterthree.htm, Focal Press.

Atal, B.S. (1966) *Apparent Sound Source Translator.* US Patent 3236949.

Bamford, J.S. (1995) *An Analysis of Ambisonic Sound Systems of First and Second Order*, Master of Science thesis, University of Waterloo, Ontario, Canada.

Begault, D.R. (2000) *3-D Sound for Virtual Reality and Multimedia*, Retrieved: March, 2003, from http://human-factors.arc.nasa.gov/ihh/spatial/papers/pdfs_db/Begault_2000_3d_Sound_Multimedia.pdf, NASA.

Berg, J., Rumsey, R. (2001) Verification and Correlation of Attributes Used For Describing the Spatial Quality of Reproduced Sound. *Proceedings of the 19th International AES Conference, Germany.* p. 233 – 251.

Berkhout, A.J. *et al.* (1992) Acoustic Control by Wave Field Synthesis. *Journal of the AES*, Vol. 93, Num. 5, p. 2765 – 2778.

Berry, S. & Lowndes V. (2001) *Deriving a Memetic Algorithm to Solve Heat Flow Problems.* University of Derby Technical Report.

Blauert, J. (1997) *Spatial Hearing – The Psychophysics of Human Sound Localization*, MIT Press, Cambridge.

Blumlein, A. (1931) *Improvements in and relating to Sound-transmission, Sound-recording and Sound-reproducing Systems*, British Patent Application 394325.

Borland Software Corporation (2003) *C++ Builder Studio Main Product Page*. Retrieved: August, 2003, from http://www.borland.com/cbuilder/index.html.

Borwick, J. (1981) Could 'Surround Sound' Bounce Back. *The Gramophone*, February, p 1125-1126.

Brown, C. P. & Duda, R. O. (1997) *An Efficient HRTF Model for 3-D Sound*, Retrieved: April, 2003, from http://interface.cipic.ucdavis.edu/PAPERS/Brown1997(Efficient3dHRTFModels).pdf.

CMedia (N.D.) *An Introduction to Xear 3D™Sound Technology*, Retrieved: July, 2004 from http://www.cmedia.com.tw/doc/Xear%203D.pdf

Craven, P.G., Gerzon, M.A. (1977) *Coincident Microphone Simulation Covering Three Dimensional Space and Yielding Various Directional Outputs*, U.S. Patent no 4042779.

Craven, P. (2003), Continuous Surround Panning for 5-speaker Reproduction, *AES 24th International Conference*, Banff, Canada.

Daniel, J. *et al.* (2003) Further Investigations of High Order Ambisonics and Wavefield Synthesis for Holophonic Sound Imaging. *114th AES Convention, Amsterdam.* Preprint 5788

De Lancie, P. (1998) *Meridian Lossless Packing:Enabling High-Resolution Surround on DVD-Audio*. Retrieved: July, 2004 from http://www.meridian-audio.com/p_mlp_mix.htm.

Dolby Labs (2002) *A history of Dolby Labs.* Retrieved: June, 2003, from http://www.dolby.com/company/is.ot.0009.History.08.html.

Dolby Labs (2004) *Dolby Digital – General.* Retreived: July, 2004 from http://www.dolby.com/digital/diggenl.html.

Duda (1993) Modeling Head Related Transfer Functions. *Preprint for the 27th Asilomar Conference on Signals, Systems & Computers*, Asilomar, October 31st – November 3rd.

Farina, A. *et al.* (2001) Ambiophonic Principles for the Recording and Reproduction of Surround Sound for Music. *Proceedings of the 19th AES International Conference of Surround Sound*, Schloss Elmau, Germany, p. 26-46.

Farino A., Ugolotti E. (1998) Software Implementation of B-Format Encoding and Decoding. *Preprints of the 104th International AES Convention*, Amsterdam, 15 – 20 May.

Farrah, K. (1979a) Soundfield Microphone – Design and development of microphone and control unit. *Wireless World*, October, p. 48-50.

Farrar, K. (1979b) Soundfield Microphone. Parts 1 & 2. *Wireless World*, October & November. p. 48 – 50 & p. 99 – 103

Kramer, L. (N.D.) *DTS: Brief History and Technical Overview*. Retrieved: July, 2004 from http://www.dtsonline.com/media/uploads/pdfs/history,whitepapers,downloads.pdf.

Furse, R. (n.d.) *3D Audio Links and Information*. Retrieved: May, 2003, from http://www.muse.demon.co.uk/3daudio.html.

Gardner B., Martin K. (1994) *HRTF Measurements of a KEMAR Dummy-Head Microphone*, Retrieved: May, 2003, from http://sound.media.mit.edu/KEMAR.html.

Gerzon, M. A. (1974a) *Sound Reproduction Systems.* Patent No. 1494751.

Gerzon, M. A. (1974b) *What's wrong with Quadraphonics*. Retrieved: July, 2004 from http://www.audiosignal.co.uk/What's%20wrong%20with%20quadraphonics.html

Gerzon, M.A. (1977a) *Sound Reproduction Systems.* UK Patent No. 1494751.

Gerzon, M. A. (1977b) Multi-system Ambisonic Decoder, parts 1 & 2. *Wireless World*, July & August. p. 43 – 47 & p. 63 – 73.

Gerzon, M.A. (1985) Ambisonics in Multichannel Broadcasting and Video. *Journal of the Audio Engineering Society*, Vol. 33, No. 11, p. 851-871.

Gerzon, M. A. & Barton, G. J. (1992) Ambisonic Decoders for HDTV. *Proceedings of the 92nd International AES Convention,* Vienna. 24 – 27 March. Preprint 3345.

Gerzon, M.A. (1992a) Optimum Reproduction Matrices for Multispeaker Stereo. *Journal of the AES*, Vol. 40, No. 7/8, p. 571 – 589.

Gerzon M. (1992b) Psychoacoustic Decoders for Multispeaker Stereo and Surround Sound. *Proceedings of the 93rd International AES Convention*, *San Francisco*. *October* Preprint 3406

Gerzon, M.A. (1992c) General Methatheory of Auditory Localisation. *92nd International AES Convention*, Vienna, 24 – 27 March Preprint 3306.

Gerzon, M.A. (1994) Application of Blumlein Shuffling to Stereo Microphone Techniques. *Journal of the AES*, vol. 42, no. 6, p. 435-453.

Gerzon, M.A, Barton, G.J. (1998) *Surround Sound Apparatus*. U.S. Patent No. 5,757,927

Glasgal, R. (2001) The Ambiophone - Derivation of a Recording Methodology Optimized for Ambiophonic Reproduction. *Proceedings of the 19th AES International Conference*, Germany, 21 – 24 June. p. 13-25.

Glasgal, R. (2003a) *The Blumlein Conspiracy.* Retrieved: August, 2003, from http://www.ambiophonics.org/blumlein_conspiracy.htm.

Glasgal, R. (2003b) *AmbioPhonics – Chapter 4, Pinna Power.* Retrieved: June, 2003, from http://www.ambiophonics.org/Ch_4_ambiophonics_2nd_edition.htm.

Glasgal, R. (2003c) *Ambiophonics - The Science of Domestic Concert Hall Design.* Retrieved: May, 2003, from http://www.ambiophonics.org.

Gulick, W.L. *et al*. (1989) *Hearing – Physiological Acoustics, Neural Coding, and Psychoacoustics*, Oxford University Press, New York.

Huopaniemi, J. *et al* (1999) Objective and Subjective Evaluation of Head-Related Transfer Function Filter Design. *Journal of the Audio Engineers Society*, Vol 47, No. 4, p218-239

Inanaga, K. *et al.* (1995) Headphone System with Out-of-Head Localisation Applying Dynamic HRTF. *98th International AES Convention*, Paris, 25 – 28 February. Preprint 4011.

Intel Corporation (2003a), *Intel Corporation.* Retrieved: June, 2003, from http://www.intel.com.

Intel Corporation (2003b) *Intel® Software Development Projects*. Retreived: August, 2003, from http://www.intel.com/software/products/ipp/ipp30/index.htm.

Ircam (2002) *Carrouso*. Retrieved: July, 2004, from http://www.ircam.fr/produits/technologies/CARROUSO-e.html

Kahana, Y. *et al* (1997). Objective and Subjective Assessment of Systems for the Production of Virtual Acoustic Images for Multiple Listeners. *103rd AES Convention, New York*, September. Preprint 4573

Kay, J. *et al.* (1998) *Film Sound History – 40's.* Retrieved: August, 2003, from http://www.mtsu.edu/~smpte/forties.html.

Kientzle, T. (1997) *A Programmer's Guide to Sound*, Addison Wesley. New York.

Kirkeby, O. *et al.* (1999) Analysis of Ill-Conditioning of Multi-Channel Deconvolution Problems. *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New York. 17 – 20 October

Kleiner, M. (1978) Problems in the Design and Use of 'Dummy-Heads'. *Acustica*, Vol. 41, p. 183-193.

Lake DSP (1997) *Lake DSP Acoustic Explorer CD/CD-ROM v2*, Lake DSP Pty. Ltd.

Leese, M. (n.d.) *Ambisonic Surround Sound.* Retrieved: August, 2003, from http://members.tripod.com/martin_leese/Ambisonic/

Leitner *et al* (2000) Multi-Channel Sound Reproduction system for Binaural signals – The Ambisonic Approach. *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00.,* Verona, Italy, December, p. 277 – 280.

Lopez, J.J., Gonzalez, A. (2001) PC Based Real-Time Multichannel Convolver for Ambiophonic Reproduction. *Proceedings of the 19th International Conference of Surround Sound*, Germany, 21 – 24 June. p. 47-53.

Mackerson, P. *et al.* (1999) *Binaural Room Scanning – A New Tool for Acoustic and Psychoacoustic Research*. Retrieved: May, 2003, from http://www.irt.de/wittek/hauptmikrofon/theile/BRS_DAGA_1999_Paper.PDF.

Malham, D. (1998) *Spatial Hearing Mechanisms and Sound Reproduction*. Retrieved: June,2003, from http://www.york.ac.uk/inst/mustech/3d_audio/ambis2.htm.

Malham, D. (2002) *Second and Third Order Ambisonics*. Retrieved: August, 2003, from http://www.york.ac.uk/inst/mustech/3d_audio/secondor.html.

Martin, G., *et al.* (2001) A Hybrid Model For Simulating Diffused First Reflections in Two-Dimensional Synthetic Acoustic Environments. *Proceedings of the 19th International AES Conference, Germany.* p. 339 – 355.

Mason, R., *et al* (2000) Verbal and non-verbal elicitation techniques in the subjective assessment of spatial sound reproduction. *Presented at 109th AES Convention, Los Angeles, 22-25 September*. Preprint 5225.

McGriffy, D (2002) *Visual Virtual Microphone*. Retrieved: August, 2003, from, http://mcgriffy.com/audio/ambisonic/vvmic/.

McKeag, A., McGrath, D. (1996) Sound Field Format to Binaural Decoder with Head-Tracking. *6th Austrailian Regional Convention of the AES, Melbourne, Austrailia.* 10 – 12 September. Preprint 4302.

McKeag, A., McGrath, D.S. (1997) Using Auralisation Techniques to Render 5.1 Surround To Binaural and Playback. *102nd AES Convention in Munich, Germany*, 22 – 25 March. preprint 4458

Microphone Techniques (n.d.). Retrieved: August, 2003, from http://www.mhsoft.nl/MicTips.asp, *Microsoft Corporation* (2003), Retrieved: June 2003, from http://www.microsoft.com/windows/.

MIT Media Lab (2000) *MPEG-4 Structured Audio (MP4 Structured Audio).* Retrieved: August, 2003, from http://sound.media.mit.edu/mpeg4/.

Moller, H. *et al.* (1996) Binaural Technique: Do We Need Individual Recordings? *Journal of the AES*, Vol. 44, No. 6, p. 451 – 468.

Moller, H. *et al.* (1999) Evaluation of Artificial Heads in Listening Tests. *J. Acoust. Soc. Am.* 47(3), p. 83-100.

Multi Media Projekt Verdi (2002) *Design of the Listening Test*. Retrieved: July, 2004 from http://www.stud.tu-ilmenau.de/~proverdi/indexen.html.

Nelson, P.A. *et al.* (1997) Sound Fields for the Production of Virtual Acoustic Images. *Journal of Sound and Vibration*, Vol. 204(2), p. 386-396.

Nielsen, S. (1991) Depth Perception – Finding a Design Goal for Sound Reproduction systems. *90th AES Convention, Paris.* Preprint 3069.

Noisternig, M. *et al.* (2003) A 3D Ambisonic Based Binaural Sound Reproduction System. *Proceedings of the 24th International Conference on Multichannel Audio*, *Banff, Canada.* Paper 1

Orduna, F. *et al.* (1995) Subjective Evaluation of a Virtual Source Emulation System. *Proceedings of Active 95, Newport Beach, USA.* P. 1271 – 1278.

Paterson-Stephens I., Bateman A. (2001) *The DSP Handbook, Algorithms, Applications and Design Techniques*, Prentice Hall. Harlow

Petzold, C. (1998) *Programming Windows – The definitive guide to the Win32 API*, Microsoft Press, New York.

Poletti, M. (2000) A Unified Theory of Horizontal Holographic Sound Systems. *Journal of the AES*, Vol. 48, No. 12, p. 1155 – 1182.

Pulkki, V. (1997) Virtual sound source positioning using vector base amplitude panning. *Journal of the Audio Engineering Society*, Vol. 45, No. 6 p. 456-466.

Rossing T. (1990) *The Science of Sound*, Addison Wesley. Reading

Rumsey, F., McCormick, T. (1994) *Sound & Recording – an introduction*, Focal Press. Oxford

Ryan, C. and Furlong, D. (1995) Effects of headphone placement on headphone equalisation for binaural reproduction. *98th International Convention of the Audio Engineering Society*, Paris, 25 – 28 February. preprint no. 4009.

Savioja, L (1999) *Air Absorption.* Retrieved: July, 2004, from http://www.tml.hut.fi/~las/publications/thesis/Air_Absorption.html.

Schillebeeckx, P. *et al.* (2001) Using Matlab/Simulink as an implementation tool for Multi-Channel Surround Sound. *Proceedings of the 19th International AES conference on Surround Sound*, Schloss Elmau, Germany, 21 – 25 June. p. 366-372.

Serendipity (2000) *SERENDIPITY- Audio, Music, Recording and Mastering Studio*. Retrieved: August, 2003, from http://www.seripity.demon.co.uk/.

Sibbald, A. (2000) *Virtual Audio for Headphones*. Retrieved: July 2004, from http://www.sensaura.com/whitepapers/pdfs/devpc007.pdf

Sontacchi, A., Holdrich, R. (2003) Optimization Criteria For Distance Coding in 3D Sound Fields. *24th International AES Conference on Multichannel Audio, Banff.* Paper 32.

SoundField Ltd. (n.d. a) *SP451 Surround Sound Processor.* Retrieved: August, 2003, from http://www.soundfield.com/sp451.htm.

SoundField Ltd. (n.d. b). Retrieved: August, 2003, from http://www.soundfield.com.

Spikofski, G., Fruhmann, M. (2001) Optimization of Binaural Room Scanning (BRS): Considering inter-individual HRTF-characteristics. In: *Proceedings of the AES 19th International Conference, Schloss Elmau, Germany* 21 – 25 June. p.124-134.

Steinberg, J., Snow, W. (1934) Auditory Perspective – Physical Factors. In: *Electrical Engineering, January,* p.12-17*.*

Surround Sound Mailing List Archive (2001), Retrieved: June, 2003, from http://www.tonmeister.de/foren/surround/ssf_archiv/SSF_Diskussion_2001_12_2.pdf, p. 5.

Sydec Audio Engineering (2003), Retrieved: June 2003, from http://www.sydec.be.

The MathWorks (2003), Retrieved: June 2003, from http://www.mathworks.com/.

Theile, G. (2001) Multi-channel Natural Music Recording Based on Psycho-acoustic Principles. Extended version of the paper presented at the *AES 19th International Conference*. Schloss Elmau, Germany, 21 – 25 June. Retrieved: May, 2003, from http://www.irt.de/IRT/FuE/as/multi-mr-ext.pdf.

University of Erlangen-Nuremberg (N.D), *Wave Field Synthesis and Analysis,* Retrieved: July, 2004 from http://www.lnt.de/LMS/research/projects/WFS/index.php?lang=eng

Verheijen, E.N.G. *et al.* (1995) Evaluation of Loudspeaker Arrays for Wave Field Synthesis in Audio Reproduction. *98th International AES Convention*, Paris, 25 – 28 February. preprint 3974.

Vermeulen, J. (n.d.) *The Art of Optimising – Part 1*. Retrieved: August, 2003, from http://www.cfxweb.net/modules.php?name=News&file=article&sid=630.

Wiggins, B. *et al.* (2001) The analysis of multi-channel sound reproduction algorithms using HRTF data. *19th International AES Surround Sound Convention*, Schloss Elmau, Germany, 21 – 24 June. p. 111-123.

Wiggins, B. *et al.* (2003) The Design and Optimisation of Surround Sound Decoders Using Heuristic Methods. *Proceedings of UKSim 2003, Conference of the UK Simulation Society p.106-114*.

Wightman, F.L. and Kistler, D.J. (1992). The Dominant Role of Low-Frequency Interaural Time Differences in Sound Localization. *Journal of the Acoustical Society of America* 91(3), p. 1648-1661

Zacharov, N. *et al.* (1999) Round Robin Subjective Evaluation of Virtual Home Theatre Sound Systems At The AES 16th International Conference. *Proceedings of the 16th AES International Conference, Finland.* p. 544 – 553.

Zwicker, E., Fastl, H. (1999) *Psychoacoustics – Facts and Models*, Springer. Berlin.

Chapter 9 - Appendix

In this appendix, example code is given for selected programs used in this investigation. A list of all code is not given due to the extensive amount of C and Matlab code used during this research, but significant programs are given so as to aid in the reproduction of the programs that are not present. The Matlab script code is given in the first part of this appendix, followed by two programs written in C++ for the Windows operating system.

9.1 Matlab Code

9.1.1 Matlab Code Used to Show Phase differences created in Blumlein's Stereo

```
%Blumlien Stereo Phase differences
%Showing amplitude differences at a
%speaker converted to phase differences
%at the ears of a listener

N = 1024;
fs = 1024;
n=0:N;
f = 2;

%Create Left and Right Speaker Feeds
%Along with phase shifted versions
Left = sin(f*2*pi*n/fs);
Leftd = sin(f*2*pi*n/fs - pi/2);
Right = 0.3 * sin(f*2*pi*n/fs);
Rightd = 0.3 * sin(f*2*pi*n/fs - pi/2);

%Sum Example Signals arriving at Ears
LeftEar = Left + Rightd;
RightEar = Right + Leftd;

%Plot Speaker Signals
figure(1)
clf;
subplot(2,1,1)
plot(Left);
hold on
plot(Right,'r');
legend('Left Speaker','Right Speaker');
ylabel('Amplitude');
xlabel('Samples');
axis([0 N -1.2 1.2 ]);

%Plot Signals Arriving at Ears
subplot(2,1,2)
plot(LeftEar);
hold on;
plot(RightEar,'r');
legend('Left Ear','Right Ear');
```

```
ylabel('Amplitude');
xlabel('Samples');
axis([0 N -1.2 1.2 ]);
```

9.1.2 Matlab Code Used to Demonstrate Simple Blumlein Spatial Equalisation

```
%Example of Blumleins Spatial Equalisation
%used to align auditory cues in Stereo

angle=0:2*pi/127:2*pi;

Sum     = sin(angle);
Dif     = cos(angle);

Left    = (Sum - Dif)/1.13;
Right   = (Sum + Dif)/1.13;

%Angle Offset used in spatial EQ
offset = pi/16;

%Derive Left and Right Speaker feeds for both
%Low and High frequencies
SumL = (sin(pi/4-offset)*Sum+cos(pi/4-offset)*Dif);
SumH = (sin(pi/4)*Sum+cos(pi/4)*Dif);

%Plot Mid and Side Signals
figure(1)
clf;
polar(angle,abs(Sum));
hold on
polar(angle,abs(Dif),'r');
legend('Mid','Side');
FSize = 16;
Co = 0.4;
text(Co,0,'+','FontSize',FSize);
text(-Co,0,'-','FontSize',FSize+4);
text(0,-Co,'+','FontSize',FSize);
text(0,Co,'-','FontSize',FSize+4);

%Plot M+S and M-S
figure(2)
clf;
polar(angle,abs(Right));
hold on
polar(angle,abs(Left),'r');
legend('Sum of MS','Difference of MS');
FSize = 16;
Co = 0.5;
text(0,Co,'+','FontSize',FSize);
text(0,-Co,'-','FontSize',FSize+4);
text(Co,0,'+','FontSize',FSize);
text(-Co,0,'-','FontSize',FSize+4);

%Plot Low and High Frequency Versions
%of the Left and Right Speaker Feeds
figure(3)
clf;
polar(angle,abs(SumL));
hold on;
polar(angle,abs(SumH),'r');
```

```
legend('Low Frequency Pickup','High Frequency Pickup');
```

9.1.3 Matlab Code Used To Plot Spherical Harmonics

```
%Plot 0th and 1st Order Spherical Harmonics
%Reolution
N=32;
%Setup Angle Arrays
Azim = 0:2*pi/(N-1):2*pi;
Elev = -pi/2:pi/(N-1):pi/2;

%Loop Used to create Matrices representing X,Y,Z and
%Colour Values for W,X,Y and Z B-format signals
a=1;
b=1;
for i=2:N
    for j=2:N
        r=1/sqrt(2);
        [WX(a  ,b  ),WY(a  ,b  ),WZ(a  ,b  )]= ...
            sph2cart(Azim(i-1),Elev(j-1),1/sqrt(2));
        [WX(a+1,b  ),WY(a+1,b  ),WZ(a+1,b  )]= ...
            sph2cart(Azim(i-1),Elev(j  ),1/sqrt(2));
        [WX(a+2,b  ),WY(a+2,b  ),WZ(a+2,b  )]= ...
            sph2cart(Azim(i  ),Elev(j  ),1/sqrt(2));
        [WX(a+3,b  ),WY(a+3,b  ),WZ(a+3,b  )]= ...
            sph2cart(Azim(i  ),Elev(j-1),1/sqrt(2));
        [WX(a+4,b  ),WY(a+4,b  ),WZ(a+4,b  )]= ...
            sph2cart(Azim(i-1),Elev(j-1),1/sqrt(2));
        if(r>=0)
            WC(:,b)=[1;1;1;1;0];
        else
            WC(:,b)=[0;0;0;0;0];
        end

        r=cos(Azim(i-1))*cos(Elev(j-1));
        [XX(a  ,b  ),XY(a  ,b  ),XZ(a  ,b  )]= ...
            sph2cart(Azim(i-1),Elev(j-1),abs(r));
        r=cos(Azim(i-1))*cos(Elev(j));
        [XX(a+1,b  ),XY(a+1,b  ),XZ(a+1,b  )]= ...
            sph2cart(Azim(i-1),Elev(j  ),abs(r));
        r=cos(Azim(i  ))*cos(Elev(j));
        [XX(a+2,b  ),XY(a+2,b  ),XZ(a+2,b  )]= ...
            sph2cart(Azim(i  ),Elev(j  ),abs(r));
        r=cos(Azim(i  ))*cos(Elev(j-1));
        [XX(a+3,b  ),XY(a+3,b  ),XZ(a+3,b  )]= ...
            sph2cart(Azim(i  ),Elev(j-1),abs(r));
        r=cos(Azim(i-1))*cos(Elev(j-1));
        [XX(a+4,b  ),XY(a+4,b  ),XZ(a+4,b  )]= ...
            sph2cart(Azim(i-1),Elev(j-1),abs(r));
        if(r>=0)
            XC(:,b)=[1;1;1;1;0];
        else
            XC(:,b)=[0;0;0;0;0];
        end

        r=sin(Azim(i-1))*cos(Elev(j-1));
        [YX(a  ,b  ),YY(a  ,b  ),YZ(a  ,b  )]= ...
            sph2cart(Azim(i-1),Elev(j-1),abs(r));
        r=sin(Azim(i-1))*cos(Elev(j));
        [YX(a+1,b  ),YY(a+1,b  ),YZ(a+1,b  )]= ...
            sph2cart(Azim(i-1),Elev(j  ),abs(r));
        r=sin(Azim(i  ))*cos(Elev(j));
        [YX(a+2,b  ),YY(a+2,b  ),YZ(a+2,b  )]= ...
            sph2cart(Azim(i  ),Elev(j  ),abs(r));
        r=sin(Azim(i  ))*cos(Elev(j-1));
        [YX(a+3,b  ),YY(a+3,b  ),YZ(a+3,b  )]= ...
            sph2cart(Azim(i  ),Elev(j-1),abs(r));
        r=sin(Azim(i-1))*cos(Elev(j-1));
        [YX(a+4,b  ),YY(a+4,b  ),YZ(a+4,b  )]= ...
            sph2cart(Azim(i-1),Elev(j-1),abs(r));
        if(r>=0)
            YC(:,b)=[1;1;1;1;0];
        else
            YC(:,b)=[0;0;0;0;0];
        end

        r=sin(Elev(j-1));
        [ZX(a  ,b  ),ZY(a  ,b  ),ZZ(a  ,b  )]= ...
            sph2cart(Azim(i-1),Elev(j-1),abs(r));
        r=sin(Elev(j));
        [ZX(a+1,b  ),ZY(a+1,b  ),ZZ(a+1,b  )]= ...
            sph2cart(Azim(i-1),Elev(j  ),abs(r));
        r=sin(Elev(j));
        [ZX(a+2,b  ),ZY(a+2,b  ),ZZ(a+2,b  )]= ...
            sph2cart(Azim(i  ),Elev(j  ),abs(r));
        r=sin(Elev(j-1));
        [ZX(a+3,b  ),ZY(a+3,b  ),ZZ(a+3,b  )]= ...
            sph2cart(Azim(i  ),Elev(j-1),abs(r));
        r=sin(Elev(j-1));
        [ZX(a+4,b  ),ZY(a+4,b  ),ZZ(a+4,b  )]= ...
            sph2cart(Azim(i-1),Elev(j-1),abs(r));
        if(r>=0)
            ZC(:,b)=[1;1;1;1;0];
        else
            ZC(:,b)=[0;0;0;0;0];
        end

        b=b+1;
    end
end
%Plot W
figure(1)
fill3(WX,WY,WZ,WC);
light;
lighting phong;
shading interp;
axis equal
axis off;
view(-40,30);
axis([-1 1 -1 1 -1 1]);
%Plot X
figure(2)
fill3(XX,XY,XZ,XC);
light;
lighting phong;
shading interp;
axis equal
axis off;
view(-40,30);
axis([-1 1 -1 1 -1 1]);
%Plot Y
figure(3)
fill3(YX,YY,YZ,YC);
light;
```

```
lighting phong;
shading interp;
axis equal
axis off;
view(-40,30);
axis([-1 1 -1 1 -1 1]);
%Plot Z
figure(4)
fill3(ZX,ZY,ZZ,ZC);
light;
lighting phong;
shading interp;
axis equal
axis off;
view(-40,30);
axis([-1 1 -1 1 -1 1]);
```

9.1.4 Code used to plot A-format capsule responses (in 2D) using oversampling.

```
%scaling
sc=1.5;
%oversampling
fsmult = 64;
%number of capsules
noofcaps = 4;
%sampling frequency
fs = 48000 * fsmult;
h=figure(1)
h1=figure(3)
set(h,'DoubleBuffer','on');
set(h1,'DoubleBuffer','on');
i=0;
%capsule spacing
spacing = 0.012;
%resolution
N=360*32;
n=0:2*pi/(N-1):2*pi;
n=n';

AOffset = 2*pi/(2*noofcaps):2*pi/(noofcaps):2*pi;
POffsetx = spacing * cos(AOffset);
POffsety = spacing * sin(-AOffset);

xplot = zeros(N,noofcaps);
yplot = zeros(N,noofcaps);
for a=1:noofcaps
   CPolar = 0.5*(2+cos(n+AOffset(a)));
   [xplot(:,a),yplot(:,a)] = pol2cart(n,CPolar);
   xplot(:,a) =xplot(:,a) + POffsetx(a);
   yplot(:,a) =yplot(:,a) + POffsety(a);
end

%For loop uncomment out next line and comment out
%the SignalAngle = 5...
for SignalAngle = 0:2*pi/32:2*pi;
%SignalAngle = deg2rad(0);
   i=i+1;

   figure(1)
```

```
clf
hold on;

plot(xplot,yplot,'LineWidth',1.5);
signalx = cos(SignalAngle) * 2;
signaly = sin(SignalAngle) * 2;
plot([signalx,0],[signaly,0]);
axis equal;
title('Polar Diagram of A-Format and signal direction');

GainIndex = round(SignalAngle*(N-1)/(2*pi))+1;
pos = 1;
for a=1:noofcaps
   if a > noofcaps/4 & a <= 3 * noofcaps / 4
      pos = -1;
   else
      pos = 1;
   end
   plot(xplot(GainIndex,a),yplot(GainIndex,a),'p','LineWidth',3);
   Gain(a) = sqrt((xplot(GainIndex,a)-POffsetx(a))^2 ...
       + (yplot(GainIndex,a)-POffsety(a))^2);
     Gain8(a) = (sqrt((xplot(GainIndex,a)-POffsetx(a))^2 ...
       + (yplot(GainIndex,a)-POffsety(a))^2)) * pos;
end
axis([-sc,sc,-sc,sc]);

Delay = spacing - (spacing * Gain);
SDelay = (Delay*fs/340) + (spacing*fs/340) + 1;
FilterBank = zeros(round(2*spacing*fs/340) + 1,1);
FilterBank8 = zeros(round(2*spacing*fs/340) + 1,1);

for a=1:noofcaps
   FilterBank(round(SDelay(a))) = ...
       FilterBank(round(SDelay(a))) + Gain(a)/2;
   FilterBank8(round(SDelay(a))) = ...
       FilterBank8(round(SDelay(a))) + Gain8(a)*sqrt(2);
   CD(a) = Delay(a);
   CG(a) = Gain(a);
end

figure(3)
clf;
subplot(2,1,1)
stem(FilterBank);
ylim([-4 4]);
hold on;
stem(FilterBank8,'r');
title('Omni and Figure of 8 impulses (8 imp taken from X rep)');
subplot(2,1,2)
invFB = inversefilt(FilterBank);
f = 20*log10(abs(fft(FilterBank/noofcaps,512*fsmult)));
g = 20*log10(abs(fft(FilterBank8/noofcaps,512*fsmult)));
h = 1./f;

x = 120;
plot(0:24000/255:24000,f(1:512*fsmult/(2*fsmult)))
text(x*24000/255,f(x),'\leftarrow Omni Rep', ...
    'HorizontalAlignment','left');
hold on;
plot(0:24000/255:24000,g(1:512*fsmult/(2*fsmult)),'r')
text(x*24000/255,g(x),'Figure of 8 Rep \rightarrow', ...
    'HorizontalAlignment','right');
   title('Omni and Figure of 8 responses');


   ylim([-20 6]);
   xlim([0 24000]);
   xlabel('Frequency (Hz)');
   ylabel('Amplitude (dB)');
   pause(0.1);

%remember to uncomment me too!!
end

figure(2)
clf;
Wx = (xplot(:,1) + xplot(:,2) + xplot(:,3) + xplot(:,4))/2;
Wy = (yplot(:,1) + yplot(:,2) + yplot(:,3) + yplot(:,4))/2;
Xx = (xplot(:,1) + xplot(:,2) - xplot(:,3) - xplot(:,4))*sqrt(2);
Xy = (yplot(:,1) + yplot(:,2) - yplot(:,3) - yplot(:,4))*sqrt(2);
Yx = (xplot(:,1) - xplot(:,2) - xplot(:,3) + xplot(:,4))*sqrt(2);
Yy = (yplot(:,1) - yplot(:,2) - yplot(:,3) + yplot(:,4))*sqrt(2);

plot(Wx,Wy);
hold on
plot(Xx,Xy,'m');
plot(-Xx,-Xy,'m');
plot(Yx,Yy,'r');
plot(-Yx,-Yy,'r');
axis equal;
title('Reconstructed polar diagram of B Format');
x = 0.5;
text(x,0,'+X');
text(-x,0,'-X');
text(0,x,'+Y');
text(0,-x,'-Y');
```

9.1.5 Code Used to Create Free Field Crosstalk Cancellation Filters

```
%Create matlab free field dipole filters
%Speakers = +/- 30 deg
%Distance = 1m
%Mic spacing radius = 7 cm    (head radius)

%Filter Size
N = 1024;
%Mic Spacing Radius
MSpacing = 0.07;
%Speaker spacing +/- n degrees
SSpacing = 30;
%Sampling Frequency
fs = 96000;
%Speed of Sound in Air
c = 342;
%Middle of Head x & y co-ords (speaker is at origin, symmetry
%assumed)
x = sin(deg2rad(SSpacing));
y = cos(deg2rad(SSpacing));

%Left and Right Mic Coords
xr = x - MSpacing;
yr = y;

xl = x + MSpacing;
```

```
yl = y;

%Calculate Distances from origin (speaker)
rdist = sqrt(xr*xr + yr*yr);
ldist = sqrt(xl*xl + yl*yl);
%Calculate Amplitude difference at mics using inverse square law
ADif = 1-(ldist-rdist);
%Convert distance to time using speed of sound
rtime = rdist/c;
ltime = ldist/c;
timedif = ltime - rtime;
%Convert time to number of samples
sampdif = round(timedif * fs);

%Create filters
h1=zeros(1,N);
count=1;
for a=1:N
    if a==1
        h1(a) = 1;
        count=count+2;
    elseif round(a/(sampdif*2))==a/(sampdif*2)
        h1(a+1) = ADif^count;
        count=count+2;
    end
end
ht = zeros(1,sampdif+1);
ht(sampdif+1) = -ADif;
h2=conv(h1,ht);

%Plot Time Domain Representation
figure(1)
clf;
a=stem(h1);
hold on
b=stem(h2,'r');
set(a,'LineWidth',2);
set(b,'LineWidth',2);
title(['x-talk filters at +/- ',num2str(SSpacing),' degrees']);
legend('h1',' ','h2',' ');
ylabel('Amplitude');
xlabel('Sample Number (at 96kHz, c = 342ms-1)');
axis([0 1024 -1.05 1.05]);
%Plot Frequency Domain Respresentation
figure(2)
clf;
freq=0:fs/(N-1):fs;
plot(freq,20*log10(abs(fft(h1))),'LineWidth',2);
hold on
plot(freq,20*log10(abs(fft(h2,1024))),'r:','LineWidth',2);
xlim([0 fs/4]);
title(['Frequency Response at +/- ',num2str(SSpacing),' degrees']);
xlabel('Frequency (Hz)');
ylabel('Amplitude (dB)');
legend('h1','h2');
```

9.1.6 Code Used to Create Crosstalk Cancellation Filters Using HRTF Data and Inverse Filtering Techniques

```
pinna = 1;
d = 'd:\matlab\hrtf\ofull\elev0\';
```

```
ref = wavread([d, 'L0e175a.wav']);
refR = ref(:,pinna);
ref = wavread([d, 'L0e185a.wav']);
refL = ref(:,pinna);
hrtf = wavread([d, 'L0e175a.wav']);
hrtfR = hrtf(:,pinna);
hrtf = wavread([d, 'L0e185a.wav']);
hrtfL = hrtf(:,pinna);

len=4096;
temp=zeros(1,len);
offset=2048;
temp(offset:offset-1+length(hrtfL))=refL;
iL=inversefilt(temp);

win=hanning(len);
iL=iL.*win';
figure(5)
clf;
plot(iL);
hold on
plot(win);

L2 = conv(hrtfL,iL);
R2 = conv(hrtfR,iL);
win=hanning(length(L2));
L2=L2.*win';
R2=R2.*win';

figure(1)
clf;
plot(L2);
hold on
plot(R2,'r');

figure(2)
clf;
freqz(L2);
figure(3)
clf;
freqz(R2);

[h1,h2] = freqdip([L2'],[R2'],len,0,0);

h1inv = inversefilt(h1,0.0);
h1i = conv(h1,h1inv);
h2i = conv(h2,h1inv);
h1i = h1i((len-1024):(len+1023));
h2i = h2i((len-1024):(len+1023));
win = hanning(length(h1i));
h1i = h1i .* win;
h2i = h2i .* win;
figure(6)
plot([h1i,h2i]);
h1i48 = resample(h1i,48000,44100);
h2i48 = resample(h2i,48000,44100);
h148 = resample(h1,48000,44100);
h248 = resample(h2,48000,44100);

%Carry out test dipole simulation
%c = wavread('h0e030a.wav');
%c1 = c(:,2);
```

```
%c2 = c(:,1);
c1 = hrtfL;
c2 = hrtfR;
source=zeros(8191,2);
source(1,1)=1;

dipolesig=[conv(source(:,1),h1i)+conv(source(:,2),h2i),conv(source(:,
2),h1i)+conv(source(:,1),h2i)];
leftspeakerl=conv(dipolesig(:,1),c1);
leftspeakerr=conv(dipolesig(:,1),c2);
rightspeakerl=conv(dipolesig(:,2),c2);
rightspeakerr=conv(dipolesig(:,2),c1);

stereoout=[leftspeakerl+rightspeakerl,leftspeakerr+rightspeakerr];
figure(7)
clf;
freqz(stereoout(:,1));
hold on
freqz(stereoout(:,2));
```

9.1.7 Matlab Code Used in FreqDip Function for the Generation of Crosstalk Cancellation Filters

```
function [h1,h2]=freqdip(tc1,tc2,FiltLength,inband,outband)
%[h1,h2]=freqdip(tc1,tc2,FiltLength,inband,outband)
%   Frequency Domain XTalk Cancellation Filters

Lf = 500;
Hf = 20000;
if(nargin<3)
    FiltLength=2048;
    inband=0.0002;
    outband=1;
elseif(nargin<5)
    inband=0.0002;
    outband=1;
end
LowerFreq=round(FiltLength*Lf/22050);
UpperFreq=round(FiltLength*Hf/22050);
reg=ones(FiltLength,1);
reg(1:LowerFreq) = outband;
reg(LowerFreq:UpperFreq) = inband;
reg(UpperFreq:FiltLength)= outband;
regx=0:22051/FiltLength:22050;
figure(1)
clf
plot(regx,reg);

c1=tc1;
c2=tc2;

fc1=fft(c1,FiltLength);
fc2=fft(c2,FiltLength);
fnc2=fft(-c2,FiltLength);

Filt=(fc1.*fc1)-(fc2.*fc2);
FiltDenom=1./Filt;

fh1=fc1.*FiltDenom;
fh2=fnc2.*FiltDenom;
```

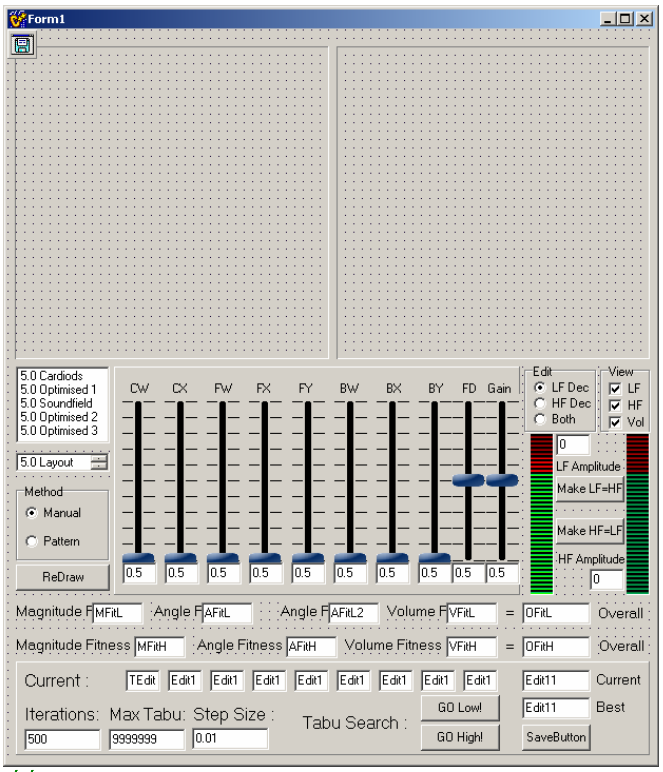
```
w = hanning(FiltLength);

h1=real(ifft(fh1,FiltLength)) .* w;
h2=real(ifft(fh2,FiltLength)) .* w;

figure(2)
clf;
plot(h1)
hold on
plot(h2,'r');
figure(3)
clf
freqz(h1,1,length(h1),44100)
hold on
freqz(h2,1,length(h2),44100)

%Carry out test dipole simulation
source=zeros(1024,2);
source(1,1)=1;

dipolesig=[conv(source(:,1),h1)+conv(source(:,2),h2),conv(source(:,2)
,h1)+conv(source(:,1),h2)];
leftspeakerl=conv(dipolesig(:,1),c1);
leftspeakerr=conv(dipolesig(:,1),c2);
rightspeakerl=conv(dipolesig(:,2),c2);
rightspeakerr=conv(dipolesig(:,2),c1);

stereoout=[leftspeakerl+rightspeakerl,leftspeakerr+rightspeakerr];
figure(4)
plot(stereoout);
```

9.1.8 Matlab Code Used To Generate Inverse Filters

```
function res = inversefilt(signal,mix)
%RES = INVERSEFILT(SIGNAL)

if(nargin==1)
    mix = 1;
end
fftsize=2^(ceil(log2(length(signal))));
fsignal=fft(signal,fftsize);

mag = abs(fsignal);
ang = angle(fsignal);

newmag = 1./mag;
newang = -ang;

newfsignal = newmag.*exp(i*newang);

newsignal = real(ifft(newfsignal,fftsize));

if(nargin==1)
    res = newsignal(1:length(signal));
else
    out = newsignal(1:length(signal));
    a = grpdelay(out,1,fftsize);
    b = round(sum(a)/fftsize);
    sig = zeros(size(out));
    sig(b) = 1;
    fo = fft(out);
    fm = fft(sig);

    fomag = abs(fo);
    fmmag = abs(fm);

    foang = angle(fo);
    fmang = angle(fm);

    newmag = (mix * fomag) + ((1-mix) * fmmag);
    newang = fmang;
    newfft = newmag.*exp(i*newang);

    fres = ifft(newfft,fftsize);
    res = real(fres);
    res = res(1:length(signal));
end
```

9.2 Windows C++ Code

9.2.1 Code Used for Heuristic Ambisonic Decoder Optimisations



```
//---------------------------------------------------------------------
//----------------------------MAIN.CPP-------------------------------
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Main.h"

#include <math.h>
#include <fstream.h>
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "VolSlider"
#pragma link "RotorSlider"
#pragma link "LevelMeter"
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
        LamL=LamH=1;
        OGainL=OGainH=1;
        SliderLength=32768;
        Bitmap = new Graphics::TBitmap;
        Bitmap2 = new Graphics::TBitmap;
        Bitmap->Height = Bevel1->Height-4;
        Bitmap->Width = Bevel1->Width-4;
        Bitmap2->Height = Bevel2->Height-4;
        Bitmap2->Width = Bevel2->Width-4;
        MaxX = Bitmap->Width/2;
        MaxY = Bitmap->Height/2;
        NoOfSpeakers = 5;
        SpeakPos[0] = 0;
        SpeakPos[1] = Deg2Rad(30);
        SpeakPos[2] = Deg2Rad(115);
        SpeakPos[3] = Deg2Rad(-115);
        SpeakPos[4] = Deg2Rad(-30);
        ListBox1->ItemIndex=0;
        ListBox1Click(this);
        WGain[0] = WGainH[0] =
                        (double)VolSlider1->Position/SliderLength;
        WGain[1] = WGainH[1] =
                        (double)VolSlider3->Position/SliderLength;
        WGain[2] = WGainH[2] =
                        (double)VolSlider6->Position/SliderLength;
        XGain[0] = XGainH[0] =
                        (double)VolSlider2->Position/SliderLength;
        XGain[1] = XGainH[1] =
                        (double)VolSlider4->Position/SliderLength;
        XGain[2] = XGainH[2] =
                        -(double)VolSlider7->Position/SliderLength;
        YGain[1] = YGainH[1] =
                        (double)VolSlider5->Position/SliderLength;
        YGain[2] = YGainH[2] =
                        (double)VolSlider8->Position/SliderLength;
        RadioGroup1->ItemIndex=1;
        VolSlider1Change(this);
        RadioGroup1->ItemIndex=0;
        VolSlider1Change(this);
}
//---------------------------------------------------------------------
double TForm1::Deg2Rad(double Deg)
{
        return (Deg*M_PI/180);
}
//---------------------------------------------------------------------
void TForm1::GPaint()
{
        long a,b,c,d;
        int SpRad = 5;
        Bitmap->Canvas->Pen->Style = psDot;
        Bitmap->Canvas->Pen->Color = clBlack;
        Bitmap->Canvas->Brush->Style = bsSolid;
        Bitmap->Canvas->Brush->Color = clWhite;
        Bitmap->Canvas->Rectangle(0,0,Bitmap->Width,Bitmap->Height);
        Bitmap->Canvas->Ellipse(0,0,Bitmap->Width,Bitmap->Height);
        Bitmap->Canvas->Pen->Style = psSolid;
        Bitmap->Canvas->Brush->Style = bsSolid;
        Bitmap->Canvas->Brush->Color = clBlue;
        for(int i=0;i<NoOfSpeakers;i++)
        {
                double x,y;
                int r = MaxY - 10;
                x = r * cos(SpeakPos[i]) + MaxX;
                y = r * sin(SpeakPos[i]) + MaxY;
                Bitmap->Canvas->Rectangle(
                              x-SpRad,y-SpRad,x+SpRad,y+SpRad);
        }
        double r8 = 0.35355339059327376220042218105242;
        double r2 = 0.70710678118654752440084436210485;
        double MFitnessL=0,AFitnessL=0,OFitnessL=0,VFitnessL=0,Ang;
        double MFitnessH=0,AFitnessH=0,OFitnessH=0,VFitnessH=0;
        for(int i=0;i<360;i++)
        {
                double Rad = Deg2Rad(i);
                WSig = 1/sqrt(2);
                XSig = cos(Rad);
                YSig = sin(Rad);
                WSigL = (0.5*(LamL+ILamL)*WSig) +
                        (r8*(LamL-ILamL)*XSig);
                XSigL = (0.5*(LamL+ILamL)*XSig) +
                        (r2*(LamL-ILamL)*WSig);
                YSigL = YSig;
                WSigH = (0.5*(LamH+ILamH)*WSig) +
                        (r8*(LamH-ILamH)*XSig);
                XSigH = (0.5*(LamH+ILamH)*XSig) +
                        (r2*(LamH-ILamH)*WSig);
                YSigH = YSig;

                SpGain[0] = (WGain[0]*WSigL + XGain[0]*XSigL);
                SpGain[1] = (WGain[1]*WSigL + XGain[1]*XSigL +
                              YGain[1]*YSigL);
                SpGain[2] = (WGain[2]*WSigL + XGain[2]*XSigL +
                              YGain[2]*YSigL);
                SpGain[3] = (WGain[2]*WSigL + XGain[2]*XSigL -
                        YGain[2]*YSigL);
                SpGain[4] = (WGain[1]*WSigL + XGain[1]*XSigL –
                        YGain[1]*YSigL);
                SpGainH[0] = (WGainH[0]*WSigH + XGainH[0]*XSigH);
                SpGainH[1] = (WGainH[1]*WSigH + XGainH[1]*XSigH +
                        YGainH[1]*YSigH);
                SpGainH[2] = (WGainH[2]*WSigH + XGainH[2]*XSigH +
                        YGainH[2]*YSigH);
                SpGainH[3] = (WGainH[2]*WSigH + XGainH[2]*XSigH –
                        YGainH[2]*YSigH);
                SpGainH[4] = (WGainH[1]*WSigH + XGainH[1]*XSigH –
                        YGainH[1]*YSigH);

                P=P2=E=VecLowX=VecLowY=VecHighX=VecHighY=0;
                for(int j=0;j<NoOfSpeakers;j++)
                {
                        P+=SpGain[j];
                        P2+=SpGainH[j]*SpGainH[j];
                        E+=pow(SpGainH[j],2);
                }
                VolLx[i]=(P*cos(Rad)*MaxX/5)+MaxX;
                VolLy[i]=(P*sin(Rad)*MaxY/5)+MaxY;
                VolHx[i]=(P2*cos(Rad)*MaxX/5)+MaxX;
                VolHy[i]=(P2*sin(Rad)*MaxY/5)+MaxY;
                if(i==0)
                {
                        LFVol = P/NoOfSpeakers;
                        HFVol = P2/NoOfSpeakers;
                }
                for(int j=0;j<NoOfSpeakers;j++)
                {
                        VecLowX+=SpGain[j]*cos(SpeakPos[j]);
                        VecLowY+=SpGain[j]*sin(SpeakPos[j]);
                        VecHighX+=pow(SpGainH[j],2)*cos(SpeakPos[j]);
                        VecHighY+=pow(SpGainH[j],2)*sin(SpeakPos[j]);
                }
                if(P && E)
                {
                        VecLowX/=P;
                        VecLowY/=P;
                        VecHighX/=E;
                        VecHighY/=E;
                }

                VFitnessL+=(1-((LFVol*NoOfSpeakers)/P))*
      (1-((LFVol*NoOfSpeakers)/P));//*((LFVol*NoOfSpeakers)-P);
                if(P2) VFitnessH+=(1-((HFVol*NoOfSpeakers)/P2))*
      (1-((HFVol*NoOfSpeakers)/P2));//*((HFVol*NoOfSpeakers)-P2);
                MFitnessL+=pow(1-
                sqrt((VecLowX*VecLowX)+(VecLowY*VecLowY)),2);
                MFitnessH+=pow(1-
                sqrt((VecHighX*VecHighX)+(VecHighY*VecHighY)),2);
                Ang=Rad-atan2(VecLowY,VecLowX);
                if(Ang>M_PI) Ang-=(2*M_PI);
                if(Ang<-M_PI) Ang+=(2*M_PI);
                AFitnessL+=(Ang)*(Ang);
                if(VecHighY || VecHighX)
                  Ang=Rad-atan2(VecHighY,VecHighX);
                if(Ang>M_PI) Ang-=(2*M_PI);
                if(Ang<-M_PI) Ang+=(2*M_PI);
                AFitnessH+=Ang*Ang;

                VecLowX*=MaxX;
                VecLowY*=MaxY;
                VecHighX*=MaxX;
                VecHighY*=MaxY;
                VecLowX+=MaxX;
                VecLowY+=MaxY;
                VecHighX+=MaxX;
                VecHighY+=MaxY;
                if(CheckBox1->Checked)
                {
                        Bitmap->Canvas->Pen->Color = clRed;
                        Bitmap->Canvas->Ellipse(VecLowX-2,
                              VecLowY-2,VecLowX+2,VecLowY+2);
                }
                if(CheckBox2->Checked)
                {
                        Bitmap->Canvas->Pen->Color = clGreen;
                        Bitmap->Canvas->Ellipse(VecHighX-2,
                              VecHighY-2,VecHighX+2,VecHighY+2);
                }
                if(i==0||i==11||i==22||i==45||i==90||i==135||i==180)
                {
                        Bitmap->Canvas->Pen->Color = clBlack;
                        Bitmap->Canvas->MoveTo(MaxX,MaxY);
                        Bitmap->Canvas->LineTo((XSig+1)*MaxX,
                                                (YSig+1)*MaxY);
                        if(CheckBox1->Checked)
                        {
                                Bitmap->Canvas->Pen->Color = clRed;
                                Bitmap->Canvas->MoveTo(MaxX,MaxY);
                                Bitmap->Canvas->LineTo(VecLowX,
                                                      VecLowY);
                        }
                        if(CheckBox2->Checked)
                        {
                                Bitmap->Canvas->Pen->Color = clGreen;
                                Bitmap->Canvas->MoveTo(MaxX,MaxY);
                                Bitmap->Canvas->LineTo(VecHighX,
                                                      VecHighY);
                        }
                }
        }
        if(CheckBox3->Checked)
        {
                int Div=5;
                Bitmap->Canvas->Pen->Color=clRed;
                Bitmap->Canvas->MoveTo((int)VolLx[359],
                                          (int)VolLy[359]);
                for(int a=0;a<360;a++)
                {
                        Bitmap->Canvas->LineTo((int)VolLx[a],
                                                (int)VolLy[a]);
                }
                Bitmap->Canvas->MoveTo(
                  (int)((VolLx[359]-MaxX)/Div)+MaxX,
                  (int)((VolLy[359]-MaxY)/Div)+MaxY);
                for(int a=0;a<360;a++)
                {
                        Bitmap->Canvas->LineTo(
                              (int)((VolLx[a]-MaxX)/Div)+MaxX,
                              (int)((VolLy[a]-MaxY)/Div)+MaxY);
                }
                Bitmap->Canvas->Pen->Color=clGreen;
                Bitmap->Canvas->MoveTo((int)VolHx[359],
                                          (int)VolHy[359]);
                for(int a=0;a<360;a++)
                {
                        Bitmap->Canvas->LineTo((int)VolHx[a],
                                                (int)VolHy[a]);
                }
        }
        VFitnessL=sqrt(VFitnessL/360.0f);
        VFitnessH=sqrt(VFitnessH/360.0f);
        AFitnessL=sqrt(AFitnessL/360.0f);
        AFitnessH=sqrt(AFitnessH/360.0f);
        MFitnessL=sqrt(MFitnessL/360.0f);
        MFitnessH=sqrt(MFitnessH/360.0f);
        OFitnessL=VFitnessL + AFitnessL + MFitnessL;
        OFitnessH=VFitnessH + AFitnessH + MFitnessH;
        a = Bevel1->Left + 2;
        b = Bevel1->Top + 2;
        c = Bevel1->Width + a -2;
        d = Bevel1->Height + b -2;
        BitBlt(Form1->Canvas->Handle,a,b,c,d,
                        Bitmap->Canvas->Handle,0,0,SRCCOPY);
        MFitL->Text=FloatToStrF(MFitnessL,ffFixed,5,5);
        MFitH->Text=FloatToStrF(MFitnessH,ffFixed,5,5);
        AFitL->Text=FloatToStrF(AFitnessL,ffFixed,5,5);
        AFitL2->Text=FloatToStrF(AFitnessL,ffFixed,5,5);
        AFitH->Text=FloatToStrF(AFitnessH,ffFixed,5,5);
        VFitL->Text=FloatToStrF(VFitnessL,ffFixed,5,5);
        VFitH->Text=FloatToStrF(VFitnessH,ffFixed,5,5);
        OFitL->Text=FloatToStrF(OFitnessL,ffFixed,5,5);
        OFitH->Text=FloatToStrF(OFitnessH,ffFixed,5,5);
        LFEdit->Text=FloatToStrF(LFVol,ffFixed,3,3);
        HFEdit->Text=FloatToStrF(HFVol,ffFixed,3,3);
        LevelMeter1->MeterReading=(int)(LFVol*75);
        LevelMeter2->MeterReading=(int)(HFVol*75);
}
//---------------------------------------------------------------------------
```

```
void TForm1::RPaint()
{
        long a,b,c,d;
        int skip = 9;
        Bitmap2->Canvas->Pen->Style = psDot;
        Bitmap2->Canvas->Pen->Color = clBlack;
        Bitmap2->Canvas->Brush->Style = bsSolid;
        Bitmap2->Canvas->Brush->Color = clWhite;
        Bitmap2->Canvas->Rectangle(0,0,
                        Bitmap2->Width,Bitmap2->Height);
        for(int i=0;i<360;i+=skip)
        {
                if(RadioGroup1->ItemIndex==0)
                {
                        Rep1[i] = 0.5 * (0.7071 * WGain[0] +
                                    cos(Deg2Rad(i))*XGain[0]);
                        Rep2[i] = 0.5 * (0.7071 * WGain[1] +
            cos(Deg2Rad(i))*XGain[1] + sin(Deg2Rad(i))*YGain[1]);
                        Rep3[i] = 0.5 * (0.7071 * WGain[2] +
            cos(Deg2Rad(i))*XGain[2] + sin(Deg2Rad(i))*YGain[2]);
                        Rep4[i] = 0.5 * (0.7071 * WGain[2] +
            cos(Deg2Rad(i))*XGain[2] - sin(Deg2Rad(i))*YGain[2]);
                        Rep5[i] = 0.5 * (0.7071 * WGain[1] +
            cos(Deg2Rad(i))*XGain[1] - sin(Deg2Rad(i))*YGain[1]);
                        Rep1[i]<0?Rep1[i]=-Rep1[i]:Rep1[i]=Rep1[i];
                        Rep2[i]<0?Rep2[i]=-Rep2[i]:Rep2[i]=Rep2[i];
                        Rep3[i]<0?Rep3[i]=-Rep3[i]:Rep3[i]=Rep3[i];
                        Rep4[i]<0?Rep4[i]=-Rep4[i]:Rep4[i]=Rep4[i];
                        Rep5[i]<0?Rep5[i]=-Rep5[i]:Rep5[i]=Rep5[i];
                }
                else
                {
                        Rep1[i] = 0.5 * (0.7071 * WGainH[0] +
            cos(Deg2Rad(i))*XGainH[0]);
                        Rep2[i] = 0.5 * (0.7071 * WGainH[1] +
            cos(Deg2Rad(i))*XGainH[1] + sin(Deg2Rad(i))*YGainH[1]);
                        Rep3[i] = 0.5 * (0.7071 * WGainH[2] +
            cos(Deg2Rad(i))*XGainH[2] + sin(Deg2Rad(i))*YGainH[2]);
                        Rep4[i] = 0.5 * (0.7071 * WGainH[2] +
            cos(Deg2Rad(i))*XGainH[2] - sin(Deg2Rad(i))*YGainH[2]);
                        Rep5[i] = 0.5 * (0.7071 * WGainH[1] +
            cos(Deg2Rad(i))*XGainH[1] - sin(Deg2Rad(i))*YGainH[1]);
                        Rep1[i]<0?Rep1[i]=-Rep1[i]:Rep1[i]=Rep1[i];
                        Rep2[i]<0?Rep2[i]=-Rep2[i]:Rep2[i]=Rep2[i];
                        Rep3[i]<0?Rep3[i]=-Rep3[i]:Rep3[i]=Rep3[i];
                        Rep4[i]<0?Rep4[i]=-Rep4[i]:Rep4[i]=Rep4[i];
                        Rep5[i]<0?Rep5[i]=-Rep5[i]:Rep5[i]=Rep5[i];
                }
        }
        Bitmap2->Canvas->Pen->Width = 2;
        Bitmap2->Canvas->Pen->Style=psSolid;
        Bitmap2->Canvas->Pen->Color=clBlack;
        PlotPolar(Bitmap2,Rep1,skip);
        Bitmap2->Canvas->Pen->Color=clRed;
        PlotPolar(Bitmap2,Rep2,skip);
        Bitmap2->Canvas->Pen->Color=clBlue;
        PlotPolar(Bitmap2,Rep3,skip);
        Bitmap2->Canvas->Pen->Color=clPurple;
        PlotPolar(Bitmap2,Rep4,skip);
        Bitmap2->Canvas->Pen->Color=clTeal;
        PlotPolar(Bitmap2,Rep5,skip);
        a = Bevel2->Left + 2;
        b = Bevel2->Top + 2;
        c = Bevel2->Width + a -2;
        d = Bevel2->Height + b -2;
        BitBlt(Form1->Canvas->Handle,a,b,c,d,
                  Bitmap2->Canvas->Handle,0,0,SRCCOPY);

}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        GPaint();
        RPaint();
}
//---------------------------------------------------------------------
void __fastcall TForm1::FormPaint(TObject *Sender)
{
        GPaint();
        RPaint();
}
//---------------------------------------------------------------------
void __fastcall TForm1::VolSlider1Change(TObject *Sender)
{
                if(RadioGroup1->ItemIndex==0)
                {
                        OGainL   =
                  (double)VolSlider10->Position*2/SliderLength;
                        WGain[0] =
                  (double)OGainL*VolSlider1->Position/SliderLength;
                        WGain[1] =
                  (double)OGainL*VolSlider3->Position/SliderLength;
                        WGain[2] =
                  (double)OGainL*VolSlider6->Position/SliderLength;
                        XGain[0] =
                  (double)OGainL*VolSlider2->Position/SliderLength;
                        XGain[1] =
                  (double)OGainL*VolSlider4->Position/SliderLength;
                        XGain[2] =
                  -(double)OGainL*VolSlider7->Position/SliderLength;
                        YGain[1] =
                  (double)OGainL*VolSlider5->Position/SliderLength;
                        YGain[2] =
                  (double)OGainL*VolSlider8->Position/SliderLength;
                        LamL     =
                  (double)VolSlider9->Position*2/SliderLength;
                        if(LamL)
                                ILamL=1/LamL;

                }
                else if(RadioGroup1->ItemIndex==1)
                {
                        WGainH[0] =
                  (double)OGainH*VolSlider1->Position/SliderLength;
                        WGainH[1] =
                  (double)OGainH*VolSlider3->Position/SliderLength;
                        WGainH[2] =
                  (double)OGainH*VolSlider6->Position/SliderLength;
                        XGainH[0] =
                  (double)OGainH*VolSlider2->Position/SliderLength;
                        XGainH[1] =
                  (double)OGainH*VolSlider4->Position/SliderLength;
                        XGainH[2] =
                  -(double)OGainH*VolSlider7->Position/SliderLength;
```

```
                        YGainH[1] =
                  (double)OGainH*VolSlider5->Position/SliderLength;
                        YGainH[2] =
                  (double)OGainH*VolSlider8->Position/SliderLength;
                        LamH     =
                  (double)VolSlider9->Position*2/SliderLength;
                        if(LamH)
                                ILamH=1/LamH;
                        OGainH   =
                  (double)VolSlider10->Position*2/SliderLength;

                }
                else if(RadioGroup1->ItemIndex==2)
                {
                        OGainH = OGainL   =
                  (double)VolSlider10->Position*2/SliderLength;
                        WGainH[0] = WGain[0] =
                  (double)OGainL*VolSlider1->Position/SliderLength;
                        WGainH[1] = WGain[1] =
                  (double)OGainL*VolSlider3->Position/SliderLength;
                        WGainH[2] = WGain[2] =
                  (double)OGainL*VolSlider6->Position/SliderLength;
                        XGainH[0] = XGain[0] =
                  (double)OGainL*VolSlider2->Position/SliderLength;
                        XGainH[1] = XGain[1] =
                  (double)OGainL*VolSlider4->Position/SliderLength;
                        XGainH[2] = XGain[2] = -
                  (double)OGainL*VolSlider7->Position/SliderLength;
                        YGainH[1] = YGain[1] =
                  (double)OGainL*VolSlider5->Position/SliderLength;
                        YGainH[2] = YGain[2] =
                  (double)OGainL*VolSlider8->Position/SliderLength;
                        LamH = LamL     =
                  (double)VolSlider9->Position*2/SliderLength;
                        if(LamL)
                                ILamL=1/LamL;
                        if(LamH)
                                ILamH=1/LamH;
                }
        UpdateEdits();
        GPaint();
        RPaint();
}
//---------------------------------------------------------------------
void TForm1::UpdateEdits()
{
                if(RadioGroup1->ItemIndex==0)
                {
                        Edit1->Text=FloatToStrF(WGain[0],
                                    ffFixed,3,3);
                        Edit3->Text=FloatToStrF(WGain[1],
                                    ffFixed,3,3);
                        Edit6->Text=FloatToStrF(WGain[2],
                                    ffFixed,3,3);
                        Edit2->Text=FloatToStrF(XGain[0],
                                    ffFixed,3,3);
                        Edit4->Text=FloatToStrF(XGain[1],
                                    ffFixed,3,3);
                        Edit7->Text=FloatToStrF(XGain[2],
                                    ffFixed,3,3);
                        Edit5->Text=FloatToStrF(YGain[1],
                                    ffFixed,3,3);
                        Edit8->Text=FloatToStrF(YGain[2],
                                        ffFixed,3,3);
                        Edit9->Text=FloatToStrF(LamL,ffFixed,3,3);
                        Edit10->Text=FloatToStrF(OGainL,ffFixed,3,3);
                }
                else if(RadioGroup1->ItemIndex==1)
                {
                        Edit1->Text=FloatToStrF(WGainH[0],
                                        ffFixed,3,3);
                        Edit3->Text=FloatToStrF(WGainH[1],
                                        ffFixed,3,3);
                        Edit6->Text=FloatToStrF(WGainH[2],
                                        ffFixed,3,3);
                        Edit2->Text=FloatToStrF(XGainH[0],
                                        ffFixed,3,3);
                        Edit4->Text=FloatToStrF(XGainH[1],
                                        ffFixed,3,3);
                        Edit7->Text=FloatToStrF(XGainH[2],
                                        ffFixed,3,3);
                        Edit5->Text=FloatToStrF(YGainH[1],
                                        ffFixed,3,3);
                        Edit8->Text=FloatToStrF(YGainH[2],
                                        ffFixed,3,3);
                        Edit9->Text=FloatToStrF(LamH,ffFixed,3,3);
                        Edit10->Text=FloatToStrF(OGainH,ffFixed,3,3);
                }
}
//-------------------------------------------------------------------
void TForm1::UpdateNewEdits()
{
                if(RadioGroup1->ItemIndex==0)
                {
                        GEdit1->Text=FloatToStrF(
                  (float)GainSlider1->Position/100,ffFixed,3,3);
                        GEdit2->Text=FloatToStrF(
                  (float)GainSlider2->Position/100,ffFixed,3,3);
                        GEdit3->Text=FloatToStrF(
                  (float)GainSlider3->Position/100,ffFixed,3,3);
                        DEdit1->Text=FloatToStrF(
                  (float)DSlider1->Position/100,ffFixed,3,3);
                        DEdit2->Text=FloatToStrF(
                  (float)DSlider2->Position/100,ffFixed,3,3);
                        DEdit3->Text=FloatToStrF(
                  (float)DSlider3->Position/100,ffFixed,3,3);
                        AEdit1->Text=IntToStr(
                  (int)ASlider1->DotPosition);
                        AEdit2->Text=IntToStr(
                  (int)ASlider2->DotPosition);
                        AEdit3->Text=IntToStr(
                  (int)ASlider3->DotPosition);
                }
                else if(RadioGroup1->ItemIndex==1)
                {
                        GEdit1->Text=FloatToStrF(
                  (float)GainSlider1->Position/100,ffFixed,3,3);
                        GEdit2->Text=FloatToStrF(
                  (float)GainSlider2->Position/100,ffFixed,3,3);
                        GEdit3->Text=FloatToStrF(
                  (float)GainSlider3->Position/100,ffFixed,3,3);
                        DEdit1->Text=FloatToStrF(
                  (float)DSlider1->Position/100,ffFixed,3,3);
                        DEdit2->Text=FloatToStrF(
                  (float)DSlider2->Position/100,ffFixed,3,3);
                        DEdit3->Text=FloatToStrF(
                  (float)DSlider3->Position/100,ffFixed,3,3);
                        AEdit1->Text=FloatToStrF(
                  (float)ASlider1->DotPosition/100,ffFixed,3,3);
                        AEdit2->Text=FloatToStrF(
                  (float)ASlider2->DotPosition/100,ffFixed,3,3);
                        AEdit3->Text=FloatToStrF(
                  (float)ASlider3->DotPosition/100,ffFixed,3,3);
            }
}
//--------------------------------------------------------------------
void __fastcall TForm1::ListBox1Click(TObject *Sender)
{
        if(ListBox1->ItemIndex==0)
        {
                VolSlider1->Position = 0.34190f*SliderLength;
                VolSlider3->Position = 0.26813f*SliderLength;
                VolSlider6->Position = 0.56092f*SliderLength;
                VolSlider2->Position = 0.23322f*SliderLength;
                VolSlider4->Position = 0.38191f*SliderLength;
                VolSlider7->Position = 0.49852f*SliderLength;
                VolSlider5->Position = 0.50527f*SliderLength;
                VolSlider8->Position = 0.45666f*SliderLength;
                VolSlider9->Position = 1*SliderLength/2;
                VolSlider10->Position = 1*SliderLength/2;
                VolSlider1Change(this);
                WGainH[0]=0.38324f;
                WGainH[1]=0.44022f;
                WGainH[2]=0.78238f;
                XGainH[0]=0.37228f;
                XGainH[1]=0.23386f;
                XGainH[2]=-0.55322f;
                YGainH[1]=0.54094f;
                YGainH[2]=0.42374f;
                LamH=1;
                ILamH=1/LamH;
                OGainH=1;


        }
        else if(ListBox1->ItemIndex==1)
        {
                RadioGroup1->ItemIndex=0;
                VolSlider1->Position = 0.58*SliderLength;
                VolSlider3->Position = 0.16*SliderLength;
                VolSlider6->Position = 1*SliderLength;
                VolSlider2->Position = 0.47*SliderLength;
                VolSlider4->Position = 0.53*SliderLength;
                VolSlider7->Position = 0.77*SliderLength;
                VolSlider5->Position = 0.55*SliderLength;
                VolSlider8->Position = 0.83*SliderLength;
                VolSlider9->Position = 1*SliderLength/2;
                VolSlider10->Position = 1*SliderLength/2;
                VolSlider1Change(this);
                WGainH[0]=0.260;
                WGainH[1]=0.320;
                WGainH[2]=1.000;
                XGainH[0]=0.200;
                XGainH[1]=0.280;
                XGainH[2]=-0.64;
                YGainH[1]=0.480;
                YGainH[2]=0.340;
                LamH=1;
                ILamH=1/LamH;
                OGainH=1;
        }
        else if(ListBox1->ItemIndex==2)
        {
                RadioGroup1->ItemIndex=0;
                VolSlider1->Position = sqrt(2.0f)*SliderLength;
                VolSlider3->Position = sqrt(2.0f)*SliderLength;
                VolSlider6->Position = sqrt(2.0f)*SliderLength;
                VolSlider2->Position = cos(SpeakPos[0])*SliderLength;
                VolSlider4->Position = cos(Deg2Rad(45))*SliderLength;
                VolSlider7->Position = -cos(Deg2Rad(135))
                                          *SliderLength;
                VolSlider5->Position = sin(Deg2Rad(45))*SliderLength;
                VolSlider8->Position = sin(Deg2Rad(135))
                                          *SliderLength;
                VolSlider9->Position = 1*SliderLength/2;
                VolSlider10->Position = 1*SliderLength/2;
                VolSlider1Change(this);
                WGainH[0]=WGain[0];
                WGainH[1]=WGain[1];
                WGainH[2]=WGain[2];
                XGainH[0]=XGain[0];
                XGainH[1]=XGain[1];
                XGainH[2]=XGain[2];
                YGainH[1]=YGain[1];
                YGainH[2]=YGain[2];
                LamH=1;
                ILamH=1/LamH;
                OGainH=1;
        }
        else if(ListBox1->ItemIndex==3)
        {
                RadioGroup1->ItemIndex=0;
                VolSlider1->Position = 0.023*SliderLength;
                VolSlider3->Position = 0.4232*SliderLength;
                VolSlider6->Position = 0.9027*SliderLength;
                VolSlider2->Position = 0.2518*SliderLength;
                VolSlider4->Position = 0.6014*SliderLength;
                VolSlider7->Position = 0.7245*SliderLength;
                VolSlider5->Position = 0.2518*SliderLength;
                VolSlider8->Position = 0.9062*SliderLength;
                VolSlider9->Position = 1*SliderLength/2;
                VolSlider10->Position = 1*SliderLength/2;
                VolSlider1Change(this);
                WGainH[0]=0;
                WGainH[1]=0.6086;
                WGainH[2]=1.0290;
                XGainH[0]=0;
                XGainH[1]=0.4998;
                XGainH[2]=-0.2058;
                YGainH[1]=0.3861;
                YGainH[2]=0.2489;
                LamH=0.9270;
                ILamH=1/LamH;
                OGainH=1;
        }
        else if(ListBox1->ItemIndex==4)
        {
                RadioGroup1->ItemIndex=0;
                VolSlider1->Position = 0.26*SliderLength;
                VolSlider3->Position = 0.34*SliderLength;
                VolSlider6->Position = 1*SliderLength;
                VolSlider2->Position = 0.247*SliderLength;
                VolSlider4->Position = 0.66*SliderLength;
                VolSlider7->Position = 0.78*SliderLength;
                VolSlider5->Position = 1*SliderLength;
                VolSlider8->Position = 0.587*SliderLength;
                VolSlider9->Position = 1*SliderLength/2;
                VolSlider10->Position = 1*SliderLength/2;
                VolSlider1Change(this);
                WGainH[0]=0.312;
                WGainH[1]=0.503;
                WGainH[2]=0.868;
                XGainH[0]=0.176;
                XGainH[1]=0.563;
                XGainH[2]=-0.41;
                YGainH[1]=0.517;
                YGainH[2]=0.510;
                LamH=1.030;
                ILamH=1/LamH;
                OGainH=1;
        }
        GPaint();
        RPaint();

}
//---------------------------------------------------------------------
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
        VolSlider1Change(this);
}
//---------------------------------------------------------------------
void TForm1::PlotPolar(Graphics::TBitmap *Bmap,double *Radius,
                        int skip)
{
        int t1,t2;
        t1=(int)(Radius[360-skip]*cos(Deg2Rad(360-skip))*MaxX)+MaxX;
        t2=(int)(Radius[360-skip]*sin(Deg2Rad(360-skip))*MaxY)+MaxY;
        Bmap->Canvas->MoveTo(t1,t2);
        for(int i=0;i<360;i+=skip)
        {
                t1=(int)(Radius[i]*cos(Deg2Rad(i))*MaxX)+MaxX;
                t2=(int)(Radius[i]*sin(Deg2Rad(i))*MaxY)+MaxY;
                Bmap->Canvas->LineTo(t1,t2);
        }
}
//---------------------------------------------------------------------
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{
        if(RadioGroup1->ItemIndex==0)
        {
                VolSlider1->Position = (int)(WGain[0]*SliderLength);
                VolSlider3->Position = (int)(WGain[1]*SliderLength);
                VolSlider6->Position = (int)(WGain[2]*SliderLength);
                VolSlider2->Position = (int)(XGain[0]*SliderLength);
                VolSlider4->Position = (int)(XGain[1]*SliderLength);
                VolSlider7->Position = (int)(-XGain[2]*SliderLength);
                VolSlider5->Position = (int)(YGain[1]*SliderLength);
                VolSlider8->Position = (int)(YGain[2]*SliderLength);
                VolSlider9->Position = (int)(LamL*SliderLength/2);
                VolSlider10->Position = (int)(OGainL*SliderLength/2);
        }
        else if(RadioGroup1->ItemIndex==1)
        {
                VolSlider1->Position = (int)(WGainH[0]*SliderLength);
                VolSlider3->Position = (int)(WGainH[1]*SliderLength);
                VolSlider6->Position = (int)(WGainH[2]*SliderLength);
                VolSlider2->Position = (int)(XGainH[0]*SliderLength);
                VolSlider4->Position = (int)(XGainH[1]*SliderLength);
                VolSlider7->Position =
                                    (int)(-XGainH[2]*SliderLength);
                VolSlider5->Position = (int)(YGainH[1]*SliderLength);
                VolSlider8->Position = (int)(YGainH[2]*SliderLength);
                VolSlider9->Position = (int)(LamH*SliderLength/2);
                VolSlider10->Position = (int)(OGainH*SliderLength/2);
        }
        UpdateEdits();
        RPaint();
}
//---------------------------------------------------------------------
void __fastcall TForm1::GainSlider1Change(TObject *Sender)
{
        if(RadioGroup1->ItemIndex==0)
        {
                WGain[0] = (double)((double)GainSlider1->Position/100
                  *(2-(double)DSlider1->Position/100));
                WGain[1] = (double)((double)GainSlider2->Position/100
                  *(2-(double)DSlider2->Position/100));
                WGain[2] = (double)((double)GainSlider3->Position/100
                  *(2-(double)DSlider3->Position/100));
                XGain[0] = (double)((double)GainSlider1->Position/100
                  *((double)DSlider1->Position/100
                  * cos(Deg2Rad((double)ASlider1->DotPosition))));
                XGain[1] = (double)((double)GainSlider2->Position/100
                  *((double)DSlider2->Position/100
                  * cos(Deg2Rad((double)ASlider2->DotPosition))));
                XGain[2] = (double)((double)GainSlider3->Position/100
                  *((double)DSlider3->Position/100
                  * cos(Deg2Rad((double)ASlider3->DotPosition))));
                YGain[1] = (double)((double)GainSlider2->Position/100
                  *((double)DSlider2->Position/100
                  * sin(Deg2Rad((double)ASlider2->DotPosition))));
                YGain[2] = (double)((double)GainSlider3->Position/100
                  *((double)DSlider3->Position/100
                  * sin(Deg2Rad((double)ASlider3->DotPosition))));
        }
        else if(RadioGroup1->ItemIndex==1)
        {
                WGainH[0] = (double)(GainSlider1->Position/100
                  *(2-DSlider1->Position/100));
                WGainH[1] = (double)(GainSlider2->Position/100
                  *(2-DSlider2->Position/100));
                WGainH[2] = (double)(GainSlider3->Position/100
                  *(2-DSlider3->Position/100));
                XGainH[0] = (double)(GainSlider1->Position/100
                  *(DSlider1->Position/100
                  * cos(Deg2Rad((double)ASlider1->DotPosition))));
                XGainH[1] = (double)(GainSlider2->Position/100
                  *(DSlider2->Position/100
                  * cos(Deg2Rad((double)ASlider1->DotPosition))));
                XGainH[2] = (double)(GainSlider3->Position/100
                  *(DSlider3->Position/100
                  * cos(Deg2Rad((double)ASlider1->DotPosition))));
                YGainH[1] = (double)(GainSlider2->Position/100
                  *(DSlider2->Position/100
                  * sin(Deg2Rad((double)ASlider1->DotPosition))));
                YGainH[2] = (double)(GainSlider3->Position/100
                  *(DSlider3->Position/100
                  * sin(Deg2Rad((double)ASlider1->DotPosition))));
        }
        UpdateNewEdits();
        GPaint();
        RPaint();

}
//---------------------------------------------------------------------
void __fastcall TForm1::RadioGroup2Click(TObject *Sender)
{
        if(RadioGroup2->ItemIndex==0)
        {
                Panel1->Show();
                Panel2->Hide();
        }
        else if(RadioGroup2->ItemIndex==1)
        {
                Panel2->Show();
                Panel1->Hide();
        }
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        RadioGroup1->ItemIndex=0;
        double GainDif=HFVol/LFVol;
        VolSlider1->Position*=GainDif;
        VolSlider2->Position*=GainDif;
        VolSlider3->Position*=GainDif;
        VolSlider4->Position*=GainDif;
        VolSlider5->Position*=GainDif;
        VolSlider6->Position*=GainDif;
        VolSlider7->Position*=GainDif;
        VolSlider8->Position*=GainDif;
        VolSlider1Change(this);
        RPaint();
        GPaint();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
        RadioGroup1->ItemIndex=1;
        double GainDif=LFVol/HFVol;
        VolSlider1->Position*=GainDif;
        VolSlider2->Position*=GainDif;
        VolSlider3->Position*=GainDif;
        VolSlider4->Position*=GainDif;
        VolSlider5->Position*=GainDif;
        VolSlider6->Position*=GainDif;
        VolSlider7->Position*=GainDif;
        VolSlider8->Position*=GainDif;
        VolSlider1Change(this);
        RPaint();
        GPaint();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
```

```
{
        Button4->Enabled=false;
        RadioGroup1->ItemIndex=0;
        Iterations = StrToInt(Edit12->Text);
        int ItCount = Iterations;
        MaxTabu = StrToInt(Edit13->Text);
        StepSize = StrToFloat(Edit14->Text);
        TempArray[0]=WGain[0];
        TempArray[2]=WGain[1];
        TempArray[5]=WGain[2];
        TempArray[1]=XGain[0];
        TempArray[3]=XGain[1];
        TempArray[6]=-XGain[2];
        TempArray[4]=YGain[1];
        TempArray[7]=YGain[2];
        TempArray[8]=LamL;
        TSearch = new Tabu(TempArray,SpeakPos,5);
        TSearch->StepSize = StepSize;
        TSearch->MMax = MaxTabu;
        for(int a=0;a<Iterations;a++)
        {
                TSearch->StartTabu();
                WGain[0]=TSearch->CBest[0];
                XGain[0]=TSearch->CBest[1];
                WGain[1]=TSearch->CBest[2];
                XGain[1]=TSearch->CBest[3];
                YGain[1]=TSearch->CBest[4];
                WGain[2]=TSearch->CBest[5];
                XGain[2]=-TSearch->CBest[6];
                YGain[2]=TSearch->CBest[7];
                LamL=TSearch->CBest[8];
                TEdit1->Text=FloatToStrF(
                  TSearch->CBest[0],ffFixed,3,3);
                TEdit2->Text=FloatToStrF(
                  TSearch->CBest[1],ffFixed,3,3);
                TEdit3->Text=FloatToStrF(
                  TSearch->CBest[2],ffFixed,3,3);
                TEdit4->Text=FloatToStrF(
                  TSearch->CBest[3],ffFixed,3,3);
                TEdit5->Text=FloatToStrF(
                  TSearch->CBest[4],ffFixed,3,3);
                TEdit6->Text=FloatToStrF(
                  TSearch->CBest[5],ffFixed,3,3);
                TEdit7->Text=FloatToStrF(
                  -TSearch->CBest[6],ffFixed,3,3);
                TEdit8->Text=FloatToStrF(
                  TSearch->CBest[7],ffFixed,3,3);
                TEdit9->Text=FloatToStrF(
                  TSearch->CBest[8],ffFixed,3,3);
                TEditRes->Text=FloatToStrF(
                  TSearch->ResBestLocal,ffFixed,5,5);
                Edit11->Text=FloatToStrF(
                  TSearch->ResBestOverall,ffFixed,5,5);
                RadioGroup1Click(this);
                VolSlider1Change(this);
                Edit12->Text = IntToStr(--ItCount);
                Application->ProcessMessages();
        }
        WGain[0]=TSearch->OBest[0];
        XGain[0]=TSearch->OBest[1];
        WGain[1]=TSearch->OBest[2];
        XGain[1]=TSearch->OBest[3];
        YGain[1]=TSearch->OBest[4];
        WGain[2]=TSearch->OBest[5];
        XGain[2]=-TSearch->OBest[6];
        YGain[2]=TSearch->OBest[7];
        RadioGroup1Click(this);
        VolSlider1Change(this);
        Application->ProcessMessages();
        delete TSearch;
        Button4->Enabled=true;
        Edit12->Text = IntToStr(Iterations);
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
{
        Button5->Enabled=false;
        RadioGroup1->ItemIndex=1;
        Iterations = StrToInt(Edit12->Text);
        int ItCount = Iterations;
        MaxTabu = StrToInt(Edit13->Text);
        StepSize = StrToFloat(Edit14->Text);
        TempArray[0]=WGainH[0];
        TempArray[2]=WGainH[1];
        TempArray[5]=WGainH[2];
        TempArray[1]=XGainH[0];
        TempArray[3]=XGainH[1];
        TempArray[6]=-XGainH[2];
        TempArray[4]=YGainH[1];
        TempArray[7]=YGainH[2];
        TempArray[8]=LamH;
        TSearchH = new HighTabu(TempArray,SpeakPos,5);
        TSearchH->StepSize = StepSize;
        TSearchH->MMax = MaxTabu;
        for(int a=0;a<Iterations;a++)
        {
                TSearchH->StartTabu();
                WGainH[0]=TSearchH->CBest[0];
                XGainH[0]=TSearchH->CBest[1];
                WGainH[1]=TSearchH->CBest[2];
                XGainH[1]=TSearchH->CBest[3];
                YGainH[1]=TSearchH->CBest[4];
                WGainH[2]=TSearchH->CBest[5];
                XGainH[2]=-TSearchH->CBest[6];
                YGainH[2]=TSearchH->CBest[7];
                LamH=TSearchH->CBest[8];
                TEdit1->Text=FloatToStrF(
                  TSearchH->CBest[0],ffFixed,3,3);
                TEdit2->Text=FloatToStrF(
                  TSearchH->CBest[1],ffFixed,3,3);
                TEdit3->Text=FloatToStrF(
                  TSearchH->CBest[2],ffFixed,3,3);
                TEdit4->Text=FloatToStrF(
                  TSearchH->CBest[3],ffFixed,3,3);
                TEdit5->Text=FloatToStrF(
                  TSearchH->CBest[4],ffFixed,3,3);
                TEdit6->Text=FloatToStrF(
                  TSearchH->CBest[5],ffFixed,3,3);
                TEdit7->Text=FloatToStrF(
                  -TSearchH->CBest[6],ffFixed,3,3);
                TEdit8->Text=FloatToStrF(
                  TSearchH->CBest[7],ffFixed,3,3);
                TEdit9->Text=FloatToStrF(
                  TSearchH->CBest[8],ffFixed,3,3);
                TEditRes->Text=FloatToStrF(
                  TSearchH->ResBestLocal,ffFixed,5,5);
                Edit11->Text=FloatToStrF(
                  TSearchH->ResBestOverall,ffFixed,5,5);
                RadioGroup1Click(this);
                VolSlider1Change(this);
                Edit12->Text = IntToStr(--ItCount);
                Application->ProcessMessages();

        }
        WGainH[0]=TSearchH->OBest[0];
        XGainH[0]=TSearchH->OBest[1];
        WGainH[1]=TSearchH->OBest[2];
        XGainH[1]=TSearchH->OBest[3];
        YGainH[1]=TSearchH->OBest[4];
        WGainH[2]=TSearchH->OBest[5];
        XGainH[2]=-TSearchH->OBest[6];
        YGainH[2]=TSearchH->OBest[7];
        RadioGroup1Click(this);
        VolSlider1Change(this);
        Application->ProcessMessages();
        delete TSearchH;
        Button5->Enabled=true;
        Edit12->Text = IntToStr(Iterations);
}
//---------------------------------------------------------------------
#define Write(a)
fwrite((FloatToStrF(a,ffFixed,5,5)).c_str(),1,5,File)
#define WriteTxt(a) fwrite(a,1,sizeof(a)-1,File)
#define NewLine fwrite("\n",1,1,File)
void __fastcall TForm1::SaveButtonClick(TObject *Sender)
{
        FILE *File;
        if(SaveDialog1->Execute())
        {
                File = fopen(SaveDialog1->FileName.c_str(),"w");
                WriteTxt("WLow-C\t");Write(WGain[0]);NewLine;
                WriteTxt("XLow-C\t");Write(XGain[0]);NewLine;
                WriteTxt("WLow-F\t");Write(WGain[1]);NewLine;
                WriteTxt("XLow-F\t");Write(XGain[1]);NewLine;
                WriteTxt("YLow-F\t");Write(YGain[1]);NewLine;
                WriteTxt("WLow-R\t");Write(WGain[2]);NewLine;
                WriteTxt("XLow-R\t");Write(XGain[2]);NewLine;
                WriteTxt("YLow-R\t");Write(YGain[2]);NewLine;
                NewLine;
                WriteTxt("WHigh-C\t");Write(WGainH[0]);NewLine;
                WriteTxt("XHigh-C\t");Write(XGainH[0]);NewLine;
                WriteTxt("WHigh-F\t");Write(WGainH[1]);NewLine;
                WriteTxt("XHigh-F\t");Write(XGainH[1]);NewLine;
                WriteTxt("YHigh-F\t");Write(YGainH[1]);NewLine;
                WriteTxt("WHigh-R\t");Write(WGainH[2]);NewLine;
                WriteTxt("XHigh-R\t");Write(XGainH[2]);NewLine;
                WriteTxt("YHigh-R\t");Write(YGainH[2]);NewLine;
                fclose(File);
        }
}
//---------------------------------------------------------------------
```

```
//---------------------------------------------------------------------
//-------------------------MAIN.H-----------------------------------
//---------------------------------------------------------------------
#ifndef MainH
#define MainH
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
#include "VolSlider.h"
#include "RotorSlider.h"
#include "LevelMeter.h"
#include "Tabu.h"
#include "HighTabu.h"
#include <Dialogs.hpp>
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:      // IDE-managed Components
        TBevel *Bevel1;
        TButton *Button1;
        TListBox *ListBox1;
        TBevel *Bevel2;
        TRadioGroup *RadioGroup1;
        TGroupBox *GroupBox1;
        TCheckBox *CheckBox2;
        TCheckBox *CheckBox1;
        TListBox *ListBox2;
        TPanel *Panel1;
        TVolSlider *VolSlider1;
        TVolSlider *VolSlider2;
        TVolSlider *VolSlider3;
        TVolSlider *VolSlider4;
        TVolSlider *VolSlider5;
        TVolSlider *VolSlider6;
        TVolSlider *VolSlider7;
        TVolSlider *VolSlider8;
        TEdit *Edit1;
        TEdit *Edit2;
        TEdit *Edit3;
        TEdit *Edit4;
        TEdit *Edit5;
        TEdit *Edit6;
        TEdit *Edit7;
        TEdit *Edit8;
        TLabel *CW;
        TLabel *CX;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        TLabel *Label5;
        TLabel *Label6;
        TLabel *Label7;
        TRadioGroup *RadioGroup2;
        TPanel *Panel2;
        TVolSlider *GainSlider1;
        TRotorSlider *ASlider1;
        TVolSlider *DSlider1;
        TEdit *GEdit1;
        TEdit *AEdit1;
        TEdit *DEdit1;
        TLabel *Label1;
        TLabel *Label8;
        TVolSlider *GainSlider2;
        TEdit *GEdit2;
        TEdit *AEdit2;
        TRotorSlider *ASlider2;
        TVolSlider *DSlider2;
        TEdit *DEdit2;
        TLabel *Label9;
        TVolSlider *GainSlider3;
        TEdit *GEdit3;
        TEdit *AEdit3;
        TRotorSlider *ASlider3;
        TVolSlider *DSlider3;
        TEdit *DEdit3;
        TLevelMeter *LevelMeter1;
        TLevelMeter *LevelMeter2;
        TEdit *LFEdit;
        TEdit *HFEdit;
        TLabel *Label10;
        TLabel *Label11;
        TButton *Button2;
        TButton *Button3;
        TCheckBox *CheckBox3;
        TVolSlider *VolSlider9;
        TVolSlider *VolSlider10;
        TLabel *Label12;
        TLabel *Label13;
        TEdit *Edit9;
        TEdit *Edit10;
        TLabel *Label14;
        TLabel *Label15;
        TLabel *Label16;
        TEdit *MFitL;
        TEdit *AFitL;
        TEdit *VFitL;
        TLabel *Label17;
        TLabel *Label18;
        TLabel *Label19;
        TEdit *MFitH;
        TEdit *AFitH;
        TEdit *VFitH;
        TLabel *Label20;
        TLabel *Label21;
        TEdit *OFitL;
        TEdit *OFitH;
        TLabel *Label22;
        TLabel *Label23;
        TPanel *Panel3;
        TLabel *Label24;
        TEdit *TEdit1;
        TEdit *TEdit2;
        TEdit *TEdit3;
        TEdit *TEdit4;
        TEdit *TEdit5;
        TEdit *TEdit6;
        TEdit *TEdit7;
        TEdit *TEdit8;
        TEdit *TEdit9;
        TLabel *Label25;
        TEdit *TEditRes;
        TButton *Button4;
        TEdit *Edit11;
        TLabel *Label26;
        TLabel *Label27;
        TButton *Button5;
        TEdit *Edit12;
        TLabel *Label28;
        TLabel *Label29;
        TEdit *Edit13;
        TLabel *Label30;
        TEdit *Edit14;
        TButton *SaveButton;
        TSaveDialog *SaveDialog1;
        TEdit *AFitL2;
        TLabel *Label31;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall FormPaint(TObject *Sender);
        void __fastcall VolSlider1Change(TObject *Sender);
        void __fastcall ListBox1Click(TObject *Sender);
        void __fastcall CheckBox1Click(TObject *Sender);
        void __fastcall RadioGroup1Click(TObject *Sender);
        void __fastcall GainSlider1Change(TObject *Sender);
        void __fastcall RadioGroup2Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
        void __fastcall Button5Click(TObject *Sender);
        void __fastcall SaveButtonClick(TObject *Sender);
private:    // User declarations
        bool InUse;
        long MaxX, MaxY;
        Graphics::TBitmap *Bitmap,*Bitmap2;
        int NoOfSpeakers,SliderLength,Iterations;
        double SpeakPos[8],SpGain[8],SpGainH[8],WSig,XSig,YSig,
                  WGain[3],XGain[3],YGain[3],WGainH[3],XGainH[3],
                  YGainH[3],WSigH,WSigL,XSigH,XSigL,YSigH,YSigL;
        double P,P2,E,VecLowX,VecLowY,VecHighX,VecHighY,
                  Rep1[360],Rep2[360],Rep3[360],Rep4[360],Rep5[360],
                  LFVol,HFVol,VolLx[360],VolHx[360],VolLy[360],
                  VolHy[360],LamL,ILamL,LamH,ILamH,OGainL,OGainH;
        double Deg2Rad(double Deg);
        void PlotPolar(Graphics::TBitmap *Bitmap,double *Radius,
                        int skip);
        void UpdateEdits();
        void UpdateNewEdits();
        double TempArray[9],StepSize,MaxTabu;
public:           // User declarations
        __fastcall TForm1(TComponent* Owner);
        void GPaint();
        void RPaint();
        Tabu   *TSearch;
        HighTabu *TSearchH;

};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

```
//---------------------------------------------------------------------
//---------------------------TABU.H-----------------------------------
//---------------------------------------------------------------------
#ifndef TabuH
#define TabuH
//---------------------------------------------------------------------
#include <math.h>
class Tabu
{
private:
        double Current[32],SPosition[32],SGain[32],Vx[512],Vy[512],
                  V2x[512],V2y[512];
        double ResCurrent;
        double MFit,VFit,AFit,AFit2,P,VolScale,E;
        double NAngles,AStep;
        double W,X,Y,WSig,XSig,YSig;
        int NSpeakers,ResControl,CDir[32],ResCDir;
public:
        double CBest[32],OBest[32],ResBestLocal,ResBestOverall;
        double StepSize;
        int MUp[32],MDown[32],MMax;
        Tabu(double *Array, double *SPos, int NPoints);
        ~Tabu();
        void StartTabu();
        double CalcArrays();
};
//---------------------------------------------------------------------
Tabu::Tabu(double *Array, double *SPos, int NPoints)
{
        NAngles=90;
        StepSize=0.01;
        AStep=M_PI*2/NAngles;
        NSpeakers=NPoints;
        MMax=99999999;

        for(int a=0;a<(NPoints*2)-1;a++)
        {
                //Copy initial Startup array
                Current[a]=CBest[a]=OBest[a]=Array[a];
                SPosition[a]=SPos[a];
                MUp[a]=MDown[a]=0;
        }
        W=1/(sqrt(2.0f));
        ResBestOverall=CalcArrays();
}
//---------------------------------------------------------------------
Tabu::~Tabu()
{
}
//---------------------------------------------------------------------
void Tabu::StartTabu()
{
        double CMax;
        ResBestLocal=999999;
        for(int control=0;control<(NSpeakers*2)-2;control++)
        {
                if(control==(NSpeakers*2)-2)
                        CMax=2.0f;
                else
                        CMax=1.0f;

                for(int test=1;test<3;test++)
                {
                        if(!MUp[control] && test==1)
                        {
                                if(Current[control]>=CMax)
                                {
                                        Current[control]=CMax;
                                        MUp[control]+=5;
                                        CDir[control]=0;
                                }
                                else
                                {
                                        Current[control]+=StepSize;
                                        CDir[control]=1;
                                }
                        }
                        else if(test==1)
                        {
                                CDir[control]=0;
                        }

                        if(!MDown[control] && test==2)
                        {
                                if(Current[control]<=0)
                                {
                                        Current[control]=0;
                                        MDown[control]+=5;
                                        CDir[control]=0;
                                }
                                else
                                {
                                        Current[control]-=StepSize;
                                        CDir[control]=-1;
                                }
                        }
                        else if(test==2)
                        {
                                CDir[control]=0;
                        }

                        if(MUp[control]&&MDown[control])
                        {
                                CDir[control]=0;
                        }

                        if(CDir[control])
                        {
                                ResCurrent=CalcArrays();
                        }
                        else
                        {
                                ResCurrent=999999;
                        }

                        if(ResCurrent<ResBestLocal)
                        {
                                ResCDir=CDir[control];
                                ResControl=control;
                                for(int a=0;a<(NSpeakers*2)-1;a++)
                                        CBest[a]=Current[a];
                                ResBestLocal=ResCurrent;
                        }
                        Current[control]-=StepSize
```

```
                            *((double)CDir[control]);
                }
                if(MDown[control]>MMax) MDown[control]=MMax;
                if(MUp[control]>MMax) MUp[control]=MMax;

                if(MDown[control]) MDown[control]--;
                if(MUp[control]) MUp[control]--;
        }
        if(ResCDir==1) MDown[ResControl]+=5;
        if(ResCDir==-1) MUp[ResControl]+=5;
        for(int a=0;a<(NSpeakers*2)-1;a++)
        {
                Current[a]=CBest[a];
        }
        if(ResBestLocal<ResBestOverall)
        {
                ResBestOverall=ResBestLocal;
                for(int a=0;a<(NSpeakers*2)-1;a++)
                        OBest[a]=CBest[a];
        }
}
//---------------------------------------------------------------------
double Tabu::CalcArrays()
{
        if(!NSpeakers) Application->MessageBox("Stop1",NULL,NULL);
        double Ll=Current[8];
        double w1=Current[0],x1=Current[1],y1=0;
        double w2=Current[2],x2=Current[3],y2=Current[4];
        double w3=Current[5],x3=Current[6],y3=Current[7];
        double iLl=1/Ll,P;
        int i=0;
        MFit=VFit=AFit=E=0;
        for(double Ang=0;Ang<2*M_PI;Ang+=AStep)
        {
                X=cos(Ang);
                Y=sin(Ang);

                WSig=(0.5*(Ll+iLl)*W) + ((1/sqrt(8))*(Ll-iLl)*X);
                XSig=(0.5*(Ll+iLl)*X) + ((1/sqrt(2))*(Ll-iLl)*W);
                YSig=Y;

                SGain[0]=(w1*WSig) + (x1*XSig) + (y1*YSig);
                SGain[1]=(w2*WSig) + (x2*XSig) + (y2*YSig);
                SGain[2]=(w3*WSig) - (x3*XSig) + (y3*YSig);
                SGain[3]=(w3*WSig) - (x3*XSig) - (y3*YSig);
                SGain[4]=(w2*WSig) + (x2*XSig) - (y2*YSig);

                P=0;Vx[i]=0;Vy[i]=0;E=0;V2x[i]=0;V2y[i]=0;
                if(!NSpeakers)
                  Application->MessageBox("Stop2",NULL,NULL);
                for(int a=0;a<NSpeakers;a++)
                {
                        P+=SGain[a];
                        E+=SGain[a]*SGain[a];
                }
                if(i==0) VolScale=P;
                for(int a=0;a<NSpeakers;a++)
                {
                        Vx[i]+=SGain[a]*cos(SPosition[a]);
                        Vy[i]+=SGain[a]*sin(SPosition[a]);
                        V2x[i]+=SGain[a]*SGain[a]*cos(SPosition[a]);
                        V2y[i]+=SGain[a]*SGain[a]*sin(SPosition[a]);
                }
                if(P)
                {
                        Vx[i]/=P;
                        Vy[i]/=P;
                        V2x[i]/=E;
                        V2y[i]/=E;
                }
                VFit+=(1-(VolScale/P))*(1-(VolScale/P));
                MFit+=pow(1-sqrt((Vx[i]*Vx[i])+(Vy[i]*Vy[i])),2);
                double tAng=Ang-atan2(Vy[i],Vx[i]);
                if(tAng>M_PI) tAng-=(2*M_PI);
                if(tAng<-M_PI) tAng+=(2*M_PI);
                double tAng2=Ang-atan2(V2y[i],V2x[i]);
                if(tAng2>M_PI) tAng2-=(2*M_PI);
                if(tAng2<-M_PI) tAng2+=(2*M_PI);
                AFit2+=tAng2*tAng2;
                i++;
        }
        VFit=sqrt(VFit/(double)NAngles);
        MFit=sqrt(MFit/(double)NAngles);
        AFit=sqrt(AFit/(double)NAngles);
        AFit2=sqrt(AFit2/(double)NAngles);
        return(AFit+(AFit2)+(MFit*4.0f/5.0f)+(VFit));
}
#endif
```

```
//---------------------------------------------------------------------
//-------------------------HIGHTABU.H-------------------------------
//---------------------------------------------------------------------
#ifndef HighTabuH
#define HighTabuH

#include <math.h>

class HighTabu
{
private:
        double Current[32],SPosition[32],SGain[32],Vx[512],Vy[512];
        double ResCurrent;
        double MFit,VFit,AFit,AFit2,P,VolScale,E;
        double NAngles,AStep;
        double W,X,Y,WSig,XSig,YSig;
        int NSpeakers,ResControl,CDir[32],ResCDir;
public:
        double CBest[32],OBest[32],ResBestLocal,ResBestOverall;
        double StepSize;
        int MUp[32],MDown[32],MMax;
        HighTabu(double *Array, double *SPos, int NPoints);
        ~HighTabu();
        void StartTabu();
        double CalcArrays();
};
//---------------------------------------------------------------------
HighTabu::HighTabu(double *Array, double *SPos, int NPoints)
{
        NAngles=90;
        StepSize=0.01;
        AStep=M_PI*2/NAngles;
        NSpeakers=NPoints;
        MMax=99999999;

        for(int a=0;a<(NPoints*2)-1;a++)
        {
                //Copy initial Startup array
                Current[a]=CBest[a]=OBest[a]=Array[a];
                SPosition[a]=SPos[a];
                MUp[a]=MDown[a]=0;
        }
        W=1/(sqrt(2.0f));
        ResBestOverall=CalcArrays();
}
//---------------------------------------------------------------------
HighTabu::~HighTabu()
{
}
//---------------------------------------------------------------------
void HighTabu::StartTabu()
{
        double CMax;
        ResBestLocal=999999;
        for(int control=0;control<(NSpeakers*2)-1;control++)
        {
                if(control==(NSpeakers*2)-2)
                        CMax=2.0f;
                else
                        CMax=1.0f;

                for(int test=1;test<3;test++)
                {
                        if(!MUp[control] && test==1)
                        {
                                if(Current[control]>=CMax)
                                {
                                        Current[control]=CMax;
                                        MUp[control]+=5;
                                        CDir[control]=0;
                                }
                                else
                                {
                                        Current[control]+=StepSize;
                                        CDir[control]=1;
                                }
                        }
                        else if(test==1)
                        {
                                CDir[control]=0;
                        }

                        if(!MDown[control] && test==2)
                        {
                                if(Current[control]<=0)
                                {
                                        Current[control]=0;
                                        MDown[control]+=5;
                                        CDir[control]=0;
                                }
                                else
                                {
                                        Current[control]-=StepSize;
                                        CDir[control]=-1;
                                }
                        }
                        else if(test==2)
                        {
                                CDir[control]=0;
                        }

                        if(MUp[control]&&MDown[control])
                        {
                                CDir[control]=0;
                        }

                        if(CDir[control])
                        {
                                ResCurrent=CalcArrays();
                        }
                        else
                        {
                                ResCurrent=999999;
                        }

                        if(ResCurrent<ResBestLocal)
                        {
                                ResCDir=CDir[control];
                                ResControl=control;
                                for(int a=0;a<(NSpeakers*2)-1;a++)
                                        CBest[a]=Current[a];
                                ResBestLocal=ResCurrent;
                        }
                        Current[control]-=StepSize*
```

```
                            ((double)CDir[control]);
                }
                if(MDown[control]>MMax) MDown[control]=MMax;
                if(MUp[control]>MMax) MUp[control]=MMax;

                if(MDown[control]) MDown[control]--;
                if(MUp[control]) MUp[control]--;
        }
        if(ResCDir==1) MDown[ResControl]+=5;
        if(ResCDir==-1) MUp[ResControl]+=5;
        for(int a=0;a<(NSpeakers*2)-1;a++)
        {
                Current[a]=CBest[a];
        }
        if(ResBestLocal<ResBestOverall)
        {
                ResBestOverall=ResBestLocal;
                for(int a=0;a<(NSpeakers*2)-1;a++)
                        OBest[a]=CBest[a];
        }
}
//---------------------------------------------------------------------
double HighTabu::CalcArrays()
{
        if(!NSpeakers) Application->MessageBox("Stop1",NULL,NULL);
        double Ll=Current[8];
        double w1=Current[0],x1=Current[1],y1=0;
        double w2=Current[2],x2=Current[3],y2=Current[4];
        double w3=Current[5],x3=Current[6],y3=Current[7];
        double iLl=1/Ll,P;
        int i=0;
        MFit=VFit=AFit=0;
        for(double Ang=0;Ang<2*M_PI;Ang+=AStep)
        {
                X=cos(Ang);
                Y=sin(Ang);

                WSig=(0.5*(Ll+iLl)*W) + ((1/sqrt(8))*(Ll-iLl)*X);
                XSig=(0.5*(Ll+iLl)*X) + ((1/sqrt(2))*(Ll-iLl)*W);
                YSig=Y;

                SGain[0]=(w1*WSig) + (x1*XSig) + (y1*YSig);
                SGain[1]=(w2*WSig) + (x2*XSig) + (y2*YSig);
                SGain[2]=(w3*WSig) - (x3*XSig) + (y3*YSig);
                SGain[3]=(w3*WSig) - (x3*XSig) - (y3*YSig);
                SGain[4]=(w2*WSig) + (x2*XSig) - (y2*YSig);

                P=0;Vx[i]=0;Vy[i]=0,E=0;
                for(int a=0;a<NSpeakers;a++)
                {
                        P+=SGain[a]*SGain[a];
                        E+=SGain[a]*SGain[a];
                }
                if(i==0) VolScale=P;
                for(int a=0;a<NSpeakers;a++)
                {
                        Vx[i]+=SGain[a]*SGain[a]*cos(SPosition[a]);
                        Vy[i]+=SGain[a]*SGain[a]*sin(SPosition[a]);
                }
                if(E)
                {
                        Vx[i]/=E;
                        Vy[i]/=E;
                }
                VFit+=(1-(VolScale/P))*(1-(VolScale/P));
                MFit+=pow(1-sqrt((Vx[i]*Vx[i])+(Vy[i]*Vy[i])),2);
                double tAng=Ang-atan2(Vy[i],Vx[i]);
                if(tAng>M_PI) tAng-=(2*M_PI);
                if(tAng<-M_PI) tAng+=(2*M_PI);
                AFit+=tAng*tAng;
                i++;
        }
        VFit=sqrt(VFit/(double)NAngles);
        MFit=sqrt(MFit/(double)NAngles);
        AFit=sqrt(AFit/(double)NAngles);
        return(AFit+MFit/3+VFit/2);
}
#endif
```

9.2.2 Windows C++ Code used in the Real-Time Audio System Software

```
//--------------------------------------------------------------------
//---------------------------MAIN.CPP---------------------------------
//--------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Main.h"
#include "WigSound2.h"
//--------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TAmbiToAll *AmbiToAll;
WigSound2 *WAudio;
//--------------------------------------------------------------------
__fastcall TAmbiToAll::TAmbiToAll(TComponent* Owner)
        : TForm(Owner)
{
}
//--------------------------------------------------------------------
void __fastcall TAmbiToAll::FormCreate(TObject *Sender)
{
        WAudio = new WigSound2(this); //Gives this pointer to the
form class
        Button2->Enabled=false;
        Button3->Enabled=false;
        Button4->Enabled=false;
        ScrollBar2Change(ScrollBar2);
        ScrollBar3Change(ScrollBar3);
}
//--------------------------------------------------------------------
void __fastcall TAmbiToAll::Button1Click(TObject *Sender)
{
        unsigned short Buff=2049;
        int nchan = (NumChannels->ItemIndex+1)*2;
        m_volume = -ScrollBar2->Position/100.0f;
        if(SampleRate->ItemIndex==1)
        {
                WAudio->InitMem(nchan,Buff,48000);
                WAudio->SkipAudio(ScrollBar1->Position);
                WAudio->Initialise(nchan,48000,Buff,4,4);
        }
        else
        {
                WAudio->InitMem(nchan,Buff,44100);
                WAudio->SkipAudio(ScrollBar1->Position);
                WAudio->Initialise(nchan,44100,Buff,4,4);
        }
        WAudio->OpenDevice(1);
        Button1->Enabled=false;
        Button3->Enabled=true;
        Button4->Enabled=false;
}
//--------------------------------------------------------------------
void __fastcall TAmbiToAll::Button3Click(TObject *Sender)
{
        WAudio->Pause();
        Button2->Enabled=true;
        Button3->Enabled=false;
        Button4->Enabled=true;
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::Button2Click(TObject *Sender)
{
        WAudio->SkipAudio(ScrollBar1->Position);
        WAudio->UnPause();
        Button2->Enabled=false;
        Button3->Enabled=true;
        Button4->Enabled=false;
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::Button4Click(TObject *Sender)
{
        unsigned short Buff=2049;
        Button1->Enabled=true;
        Button2->Enabled=false;
        Button3->Enabled=false;
        Button4->Enabled=false;
        WAudio->CloseDevice(1);
        WAudio->UnInitMem(2,Buff);
        ScrollBar1->Position = 0;
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::FormDestroy(TObject *Sender)
{
        if(Button3->Enabled)
        {
                Button3Click(Button3);
                Sleep(400);
        }
        if(Button4->Enabled)
        {
                Button4Click(Button4);
                Sleep(400);
        }
        delete WAudio;
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::WButClick(TObject *Sender)
{
        TEdit *ptr = (TEdit *)Sender;
        char *cptr = ptr->Name.c_str();
        bool result;
        if(cptr[0]!='c')
                result = OpenDialog1->Execute();
        else
                result = true;
        if(result)
        {
                switch(cptr[0])
                {
                        case 'W':
                                WFName = OpenDialog1->FileName;
                                WEdit->Text = WFName;
                        break;
                        case 'X':
                                XFName = OpenDialog1->FileName;
                                XEdit->Text = XFName;
                        break;
                        case 'Y':
                                YFName = OpenDialog1->FileName;
                                YEdit->Text = YFName;
                        break;
                        case 'Z':
                                ZFName = OpenDialog1->FileName;
                                ZEdit->Text = ZFName;
                        case 'c':
                                switch(cptr[1])
                                {
                                        case 'W':
                                        WFName = NULL;
                                        WEdit->Text = WFName;
                                        break;
                                        case 'X':
                                        XFName = NULL;
                                        XEdit->Text = XFName;
                                        break;
                                        case 'Y':
                                        YFName = NULL;
                                        YEdit->Text = YFName;
                                        break;
                                        case 'Z':
                                        ZFName = NULL;
                                        ZEdit->Text = ZFName;
                                        break;
                                }
                        break;
                }
        }
}
//---------------------------------------------------------------------
void TAmbiToAll::UpdateWaveTime(unsigned long WRead)
{
        WaveRead = WRead;
        ScrollBar1->Position =
(int)((float)(WaveRead)*200.0f/(float)(WaveSize));
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::RotorSlider1Change(TObject *Sender)
{
        Label1->Caption = IntToStr((int)(360 –
                  RotorSlider1->DotPosition + 0.5f));
        RotAngle = -RotorSlider1->DotPosition*M_PI/180.0f;
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::AmbiEffectClick(TObject *Sender)
{
        m_effect = AmbiEffect->ItemIndex;
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::RotorSlider2Change(TObject *Sender)
{
        Label2->Caption = IntToStr((int)(360 - RotorSlider2-
>DotPosition+0.5f));
        monopan = -RotorSlider2->DotPosition*M_PI/180.0f;
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::TransFilterClick(TObject *Sender)
{
        WAudio->UpdateFilter = true;
}
```

```
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::ScrollBar2Change(TObject *Sender)
{
        float db;
        m_volume = -ScrollBar2->Position/100.0f;
        if(m_volume)
        {
                db = 20 * log10(m_volume);
                Label5->Caption = FloatToStrF(db,ffFixed,3,1) + "dB";
        }
        else
                Label5->Caption = "-Inf";
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::RearFilterClick(TObject *Sender)
{
        WAudio->UpdateRearFilter = true;
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::ScrollBar3Change(TObject *Sender)
{
        m_width = -ScrollBar3->Position/100.0f;
        Label6->Caption = FloatToStrF(m_width,ffFixed,4,2);
}
//---------------------------------------------------------------------
void __fastcall TAmbiToAll::RotorSlider3Change(TObject *Sender)
{
        Label9->Caption = IntToStr(
                    (int)(RotorSlider3->DotPosition - 90.0f + 0.5f));
        TiltAngle = (RotorSlider3->DotPosition - 90.0f)*M_PI/180.0f;
}
//---------------------------------------------------------------------
```

```
//---------------------------------------------------------------------
//-----------------------------MAIN.H-------------------------------
//---------------------------------------------------------------------
#ifndef MainH
#define MainH
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "RotorSlider.h"
#include "LevelMeter2.h"
#include "Oscilloscope.h"
#include "GLGraph.h"
#include <ComCtrls.hpp>
#include <ExtCtrls.hpp>
#include <Dialogs.hpp>
//---------------------------------------------------------------------
class TAmbiToAll : public TForm
{
__published:      // IDE-managed Components
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TEdit *WEdit;
        TEdit *XEdit;
        TEdit *YEdit;
        TEdit *ZEdit;
        TButton *WBut;
        TButton *XBut;
        TButton *YBut;
        TButton *ZBut;
        TOpenDialog *OpenDialog1;
        TScrollBar *ScrollBar1;
        TButton *cW;
        TButton *cX;
        TButton *cY;
        TButton *cZ;
        TRotorSlider *RotorSlider1;
        TLabel *Label1;
        TOscilloscope *Oscilloscope1;
        TOscilloscope *Oscilloscope2;
        TRadioGroup *AmbiEffect;
        TRadioGroup *AmbiInput;
        TRotorSlider *RotorSlider2;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        TRadioGroup *NumChannels;
        TRadioGroup *SampleRate;
        TRadioGroup *TransFilter;
        TRadioGroup *RearFilter;
        TScrollBar *ScrollBar2;
        TLabel *Label5;
        TScrollBar *ScrollBar3;
        TLabel *Label6;
        TLabel *Label7;
        TLabel *Label8;
        TRotorSlider *RotorSlider3;
        TLabel *Label9;
        TLabel *Label10;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall FormDestroy(TObject *Sender);
        void __fastcall WButClick(TObject *Sender);
        void __fastcall RotorSlider1Change(TObject *Sender);
        void __fastcall AmbiEffectClick(TObject *Sender);
        void __fastcall RotorSlider2Change(TObject *Sender);
        void __fastcall TransFilterClick(TObject *Sender);
        void __fastcall ScrollBar2Change(TObject *Sender);
        void __fastcall RearFilterClick(TObject *Sender);
        void __fastcall ScrollBar3Change(TObject *Sender);
        void __fastcall RotorSlider3Change(TObject *Sender);
private:    // User declarations
        bool TWriting;
public:     // User declarations
        unsigned long WaveRead;
        unsigned long WaveSize;
        void UpdateWaveTime(unsigned long WRead);
        __fastcall TAmbiToAll(TComponent* Owner);
        AnsiString WFName, XFName, YFName, ZFName;
        short m_effect;
        float m_volume,m_width;
        float RotAngle,monopan,TiltAngle;
};
//---------------------------------------------------------------------
extern PACKAGE TAmbiToAll *AmbiToAll;
//---------------------------------------------------------------------
#endif
```

```
//---------------------------------------------------------------------
//-------------------------WIGSOUND.H------------------------------
//---------------------------------------------------------------------
#ifndef WigSoundH
#define WigSoundH

#include <mmsystem.h>

class WigSound
{
private:
        WAVEHDR *WaveHeadersOut,*WaveHeadersIn,*SampleBuffer;
        HWAVEOUT hWaveOut;
        HWAVEIN hWaveIn;
        MMRESULT Error;
        unsigned int NoOfBuffers,NoOfQueueBuffers;
        unsigned short NoOfChannels,BufferLengthPerChannel;

        friend void CALLBACK WaveOutCallback(HWAVEOUT hwo, UINT uMsg,
      WORD dwInstance,DWORD dwParam1, DWORD dwParam2);
        friend void CALLBACK WaveInCallback(HWAVEIN hwi, UINT uMsg,
      DWORD dwInstance,DWORD dwParam1, DWORD dwParam2);

        void ClearBufferFromFIFO();
        void ProcessErrorIn(MMRESULT Error);
        void ProcessErrorOut(MMRESULT Error);
protected:
        WAVEFORMATEX WaveFormat;
public:
        WigSound();
        void Initialise(unsigned short usNoOfChannels,
unsigned long usSampleRate,unsigned short usBufferLengthPerChannel,
unsigned int uiNoOfBuffers,unsigned int uiNoOfQueueBuffers);
        virtual void ProcessAudio(WAVEHDR *pWaveHeader,
            unsigned short usNoOfChannels,
            unsigned short usBufferLengthPerChannel);
        virtual void MonitorAudio(WAVEHDR *pWaveHeader,
            unsigned short usNoOfChannels,
            unsigned short usBufferLengthPerChannel);
        void ProcessAudioIn(WAVEHDR *pWaveHeader,
            unsigned short usNoOfChannels,
            unsigned short usBufferLengthPerChannel);
        void OpenDevice(UINT Device);
        void CloseDevice(UINT Device);
        void Pause();
        void UnPause();
        void WaveInFunc(WAVEHDR *pWaveHeader);
        void WaveOutFunc(WAVEHDR *pWaveHeader);
        bool Closing,Paused;
        WAVEHDR *ReadBuffer,*WriteBuffer;
};
//---------------------------------------------------------------------
WigSound::WigSound()
{
}
//---------------------------------------------------------------------
void WigSound::Initialise(
        unsigned short usNoOfChannels,unsigned long usSampleRate,
        unsigned short usBufferLengthPerChannel,
        unsigned int uiNoOfBuffers,unsigned int uiNoOfQueueBuffers)
{
        WaveFormat.wFormatTag           = WAVE_FORMAT_PCM;
        WaveFormat.nChannels            = usNoOfChannels;
        WaveFormat.nSamplesPerSec       = usSampleRate;
        WaveFormat.wBitsPerSample       = 16;
        WaveFormat.nBlockAlign          =
            (unsigned short)(usNoOfChannels*16/8);
        WaveFormat.nAvgBytesPerSec      =
            (unsigned long)(usSampleRate*WaveFormat.nBlockAlign);
        WaveFormat.cbSize               = 0;

        NoOfBuffers                     = uiNoOfBuffers;
        NoOfQueueBuffers                = uiNoOfQueueBuffers;
        NoOfChannels                    = usNoOfChannels;
        BufferLengthPerChannel          = usBufferLengthPerChannel;
        SampleBuffer                    =
                                    new WAVEHDR[NoOfQueueBuffers];
        WriteBuffer                     = SampleBuffer;
        ReadBuffer                      = SampleBuffer;
        WaveHeadersOut                  = new WAVEHDR[NoOfBuffers];
        WaveHeadersIn                   = new WAVEHDR[NoOfBuffers];
        Closing                         = false;
        Paused                          = true;

        for(UINT i=0;i<NoOfBuffers;i++)
        {
                WaveHeadersOut[i].dwBufferLength =
                  usBufferLengthPerChannel*16*usNoOfChannels/8;
                WaveHeadersOut[i].lpData =
                  new char[WaveHeadersOut[i].dwBufferLength];

memset(WaveHeadersOut[i].lpData,0,WaveHeadersOut[i].dwBufferLength);
                WaveHeadersOut[i].dwFlags=0;
                WaveHeadersOut[i].dwLoops=0;

                WaveHeadersIn[i].dwBufferLength =
                  usBufferLengthPerChannel*16*usNoOfChannels/8;
                WaveHeadersIn[i].lpData =
                  new char[WaveHeadersIn[i].dwBufferLength];

memset(WaveHeadersIn[i].lpData,0,WaveHeadersIn[i].dwBufferLength);
                WaveHeadersIn[i].dwFlags=0;
                WaveHeadersIn[i].dwLoops=0;
        }

        for(UINT i=0;i<NoOfQueueBuffers;i++)
        {
                SampleBuffer[i].dwBufferLength =
                  usBufferLengthPerChannel*16*usNoOfChannels/8;
                SampleBuffer[i].lpData =
                  new char[SampleBuffer[i].dwBufferLength];

memset(SampleBuffer[i].lpData,0,SampleBuffer[i].dwBufferLength);
                SampleBuffer[i].dwFlags = 0;
                SampleBuffer[i].dwLoops = 0;
        }
}
//-------------------------------------------------------------------
void WigSound::OpenDevice(UINT Device)
{
        Device?Device--:Device=WAVE_MAPPER;

        Error = waveOutOpen(&hWaveOut,Device,&WaveFormat,
                  (DWORD)WaveOutCallback,
	          (DWORD)this,CALLBACK_FUNCTION);
	if(Error)	ProcessErrorOut(Error);

	Error = waveOutPause(hWaveOut);
	if(Error)	ProcessErrorOut(Error);

	for(UINT i=0;i<NoOfBuffers;i++)
	{
		Error = waveOutPrepareHeader(hWaveOut,
		  &WaveHeadersOut[i],sizeof(WaveHeadersOut[i]));
		if(Error)	ProcessErrorOut(Error);

		Error = waveOutWrite(hWaveOut,
		  &WaveHeadersOut[i],sizeof(WaveHeadersOut[i]));
		if(Error)	ProcessErrorOut(Error);
	}

	Error = waveInOpen(&hWaveIn,Device,&WaveFormat,
	          (DWORD)WaveInCallback,
	          (DWORD)this,CALLBACK_FUNCTION);
	if(Error)	ProcessErrorIn(Error);

	for(UINT i=0;i<NoOfBuffers;i++)
	{
		Error = waveInPrepareHeader(hWaveIn,
		  &WaveHeadersIn[i],
		  sizeof(WaveHeadersIn[i]));
		if(Error)	ProcessErrorIn(Error);

		Error = waveInAddBuffer(hWaveIn,&WaveHeadersIn[i],
		  sizeof(WaveHeadersIn[i]));
		if(Error)	ProcessErrorIn(Error);
	}

	Error = waveOutRestart(hWaveOut);
	if(Error)	ProcessErrorOut(Error);

	Error = waveInStart(hWaveIn);
	if(Error)	ProcessErrorIn(Error);
	Paused=false;
}
//---------------------------------------------------------------------
void WigSound::CloseDevice(UINT Device)
{
	Closing=true;
	Error = waveInReset(hWaveIn);
	if(Error)	ProcessErrorIn(Error);
	Error = waveOutReset(hWaveOut);
	if(Error)	ProcessErrorOut(Error);
	Sleep(300);

	for(UINT i=0;i<NoOfBuffers;i++)
	{
		Error = waveOutUnprepareHeader(hWaveOut,
		  &WaveHeadersOut[i],sizeof(WaveHeadersOut[i]));
		if(Error)	ProcessErrorOut(Error);
		if(WaveHeadersOut[i].lpData)
		  delete [] WaveHeadersOut[i].lpData;

		Error = waveInUnprepareHeader(hWaveIn,
		  &WaveHeadersIn[i],
		  sizeof(WaveHeadersIn[i]));
```

```
                if(Error)       ProcessErrorIn(Error);
                if(WaveHeadersIn[i].lpData)
                  delete [] WaveHeadersIn[i].lpData;
        }
        for(UINT i=0;i<NoOfQueueBuffers;i++)
        {
                if(SampleBuffer[i].lpData)
                  delete [] SampleBuffer[i].lpData;
        }

        if(WaveHeadersOut) delete [] WaveHeadersOut;
        if(WaveHeadersIn) delete [] WaveHeadersIn;
        if(SampleBuffer) delete [] SampleBuffer;

        Error = waveInClose(hWaveIn);
        if(Error)       ProcessErrorIn(Error);
        Error = waveOutClose(hWaveOut);
        if(Error)       ProcessErrorOut(Error);
}
//---------------------------------------------------------------------
void WigSound::Pause()
{
        Paused=true;
}
//---------------------------------------------------------------------
void WigSound::UnPause()
{
        Paused=false;
}
//---------------------------------------------------------------------
void WigSound::ProcessErrorIn(MMRESULT Error)
{
        char Text[256];
        waveInGetErrorText(Error,Text,sizeof(Text));
        MessageBox(NULL,Text,"Error",MB_OK);
}
//---------------------------------------------------------------------
void WigSound::ProcessErrorOut(MMRESULT Error)
{
        char Text[256];
        waveOutGetErrorText(Error,Text,sizeof(Text));
        MessageBox(NULL,Text,"Error",MB_OK);

}
//---------------------------------------------------------------------
void WigSound::WaveInFunc(WAVEHDR *pWaveHeader)
{
        ProcessAudioIn(pWaveHeader,NoOfChannels,
            BufferLengthPerChannel);
        Error = waveInAddBuffer(hWaveIn,pWaveHeader,
            sizeof(*pWaveHeader));

}
//---------------------------------------------------------------------
void WigSound::WaveOutFunc(WAVEHDR *pWaveHeader)
{
        ProcessAudio(pWaveHeader,NoOfChannels,
            BufferLengthPerChannel);
        ClearBufferFromFIFO();
        Error = waveOutWrite(hWaveOut,pWaveHeader,
            sizeof(*pWaveHeader));
}
```

```
//---------------------------------------------------------------------
void CALLBACK WaveOutCallback(HWAVEOUT hwo, UINT uMsg,
            DWORD dwInstance,DWORD dwParam1, DWORD dwParam2)
{
        WigSound *me = (WigSound *)dwInstance;
        switch(uMsg)
        {
                case WOM_DONE:
                {
                        if(!me->Closing)
                              me->WaveOutFunc((WAVEHDR *)dwParam1);
                        break;
                }
                default:
                        break;
        }
}
//---------------------------------------------------------------------
void CALLBACK WaveInCallback(HWAVEIN hwi, UINT uMsg, DWORD
dwInstance,
                DWORD dwParam1, DWORD dwParam2)
{
        WigSound *me = (WigSound *)dwInstance;
        switch(uMsg)
        {
                case WIM_DATA:
                {
                        if(!me->Closing)
                              me->WaveInFunc((WAVEHDR *)dwParam1);
                        break;
                }
                default:
                        break;
        }
}
//---------------------------------------------------------------------
void WigSound::ProcessAudio(WAVEHDR *pWaveHeader,
            unsigned short usNoOfChannels,
            unsigned short usBufferLengthPerChannel)
{

}
//---------------------------------------------------------------------
void WigSound::MonitorAudio(WAVEHDR *pWaveHeader, unsigned short
usNoOfChannels,
                unsigned short usBufferLengthPerChannel)
{

}
//---------------------------------------------------------------------
void WigSound::ProcessAudioIn(WAVEHDR *pWaveHeader, unsigned short
usNoOfChannels,
                unsigned short usBufferLengthPerChannel)
{
        memcpy(WriteBuffer->lpData,pWaveHeader->lpData,
                  pWaveHeader->dwBufferLength);
        WriteBuffer++;
        if(WriteBuffer>&SampleBuffer[NoOfQueueBuffers-1])
                WriteBuffer=&SampleBuffer[NoOfQueueBuffers-1];

        MonitorAudio(pWaveHeader,usNoOfChannels,
            usBufferLengthPerChannel);
}
//-------------------------------------------------------------------
void WigSound::ClearBufferFromFIFO()
{
        for(UINT i=0;i<NoOfQueueBuffers-1;i++)
        {
                memcpy(SampleBuffer[i].lpData,
                        SampleBuffer[i+1].lpData,
                        SampleBuffer[i].dwBufferLength);
        }
        if(WriteBuffer>SampleBuffer)    WriteBuffer--;
}
//-------------------------------------------------------------------
#endif
```

```
//-------------------------------------------------------------------
//-------------------------WIGSOUND2.H-------------------------------
//-------------------------------------------------------------------
#ifndef WigSoundH2
#define WigSoundH2

#include <fstream.h>
#include "WigSound.h"
#include "WigAmbi.h"
#include "WaveFile.h"
#include "FastConv.h"
#include "AllPass.h"
#include "Main.h"

#define BLEN 4096
#define FFTORDER 12
#define FFTSIZE 4096

class WigSound2 : public WigSound
{
private:
        float **Samples,**Decode,*SElev,*SAzim,*mono;
        bool bSkip;
        long SkipOffset;
        AmbiBuffer *ABuf,*BBuf;
        int NoOfSpeakers,SampleRate;
        AnsiString DIR;

        //For 2 ears
        FastFilter *WF,*XF,*YF,*ZF;
        FastFilter *WF2D,*XF2D,*YF2D;
        //For 4 ears
        FastFilter *WFf,*WFr,*XFf,*XFr,*YFf,*YFr;
        //For Front...
        FastFilter *h1fl,*h2fl,*h1fr,*h2fr;
        // and Back X-Talk Cancellation Filters
        FastFilter *h1rl,*h2rl,*h1rr,*h2rr;
        //AllPass Filters for cheap Ambisonics decoder
        AllPass *WAP,*XAP,*YAP;

        void LoadFilters(int SRate);
        void UnloadFilters();

        void ChooseFilter(int SRate);
        void ChooseRearFilter(int SRate);
        void B2Headphones(AmbiBuffer *Signal, float **Samples,
                  int NoOfChannels);
        void B2Headphones2D(AmbiBuffer *Signal, float **Samples,
                  int NoOfChannels);
        void B2Headphones4(AmbiBuffer *Signal, AmbiBuffer *Signal2,
                  float **Samples,int NoOfChannels);
        void B2Trans(AmbiBuffer *Signal,float *Left,float *Right,
                  int NoOfChannels,FastFilter *h1, FastFilter *h2,
                  FastFilter *h1r, FastFilter *h2r);
public:
        WigSound2(TAmbiToAll *Sender);
        ~WigSound2();
        void InitMem(unsigned short usNoOfChannels,
                unsigned short usBufferLengthPerChannel,
                int SRate);
        void UnInitMem( unsigned short usNoOfChannels,
                  unsigned short usBufferLengthPerChannel);
        void ProcessAudio(WAVEHDR *pWaveHeader,
                  unsigned short usNoOfChannels,
                  unsigned short usBufferLengthPerChannel);
        void MonitorAudio(WAVEHDR *pWaveHeader,
                  unsigned short usNoOfChannels,
                  unsigned short usBufferLengthPerChannel);
        void SkipAudio(int Offset);
        WigFile WFile,XFile,YFile,ZFile;
        TAmbiToAll *Window;
        bool UpdateFilter,UpdateRearFilter;
};
//---------------------------------------------------------------------
WigSound2::WigSound2(TAmbiToAll *Sender)
{
        Window = Sender;
        NoOfSpeakers=8;

        SkipOffset = 0;
        bSkip = false;
        UpdateFilter = false;

        DIR = GetCurrentDir();
        DIR+="\\";
}
WigSound2::~WigSound2()
{
}
void WigSound2::LoadFilters(int SRate)
{
        AnsiString wname,xname,yname,zname;
        ZF=NULL;
        if(SRate==48000)
        {
                wname = DIR + "Wh481024.dat";
                xname = DIR + "Xh481024.dat";
                yname = DIR + "Yh481024.dat";
                zname = DIR + "Zh481024.dat";
                WF = new FastFilter(FFTORDER,&wname,1024);
                XF = new FastFilter(FFTORDER,&xname,1024);
                YF = new FastFilter(FFTORDER,&yname,1024,1);
                ZF = new FastFilter(FFTORDER,&zname,1024);
                wname = DIR + "Wh4810242D.dat";
                xname = DIR + "Xh4810242D.dat";
                yname = DIR + "Yh4810242D.dat";
                WF2D = new FastFilter(FFTORDER,&wname,1024);
                XF2D = new FastFilter(FFTORDER,&xname,1024);
                YF2D = new FastFilter(FFTORDER,&yname,1024,1);
                wname = DIR + "WhFront1024.dat";
                xname = DIR + "XhFront1024.dat";
                yname = DIR + "YhFront1024.dat";
                WFf = new FastFilter(FFTORDER,&wname,1024);
                XFf = new FastFilter(FFTORDER,&xname,1024);
                YFf = new FastFilter(FFTORDER,&yname,1024,1);
                wname = DIR + "WhRear1024.dat";
                xname = DIR + "XhRear1024.dat";
                yname = DIR + "YhRear1024.dat";
                WFr = new FastFilter(FFTORDER,&wname,1024);
                XFr = new FastFilter(FFTORDER,&xname,1024);
                YFr = new FastFilter(FFTORDER,&yname,1024,1);
                wname = DIR + "h1348.dat";
                xname = DIR + "h2348.dat";
                h1fl = new FastFilter(FFTORDER,&wname,2048);
                h2fl = new FastFilter(FFTORDER,&xname,2048);
                h1fr = new FastFilter(FFTORDER,&wname,2048);
                h2fr = new FastFilter(FFTORDER,&xname,2048);
        }
        else
        {
                wname = DIR + "Wh1024.dat";
                xname = DIR + "Xh1024.dat";
                yname = DIR + "Yh1024.dat";
                zname = DIR + "Zh1024.dat";
                WF = new FastFilter(FFTORDER,&wname,1024);
                XF = new FastFilter(FFTORDER,&xname,1024);
                YF = new FastFilter(FFTORDER,&yname,1024,1);
                ZF = new FastFilter(FFTORDER,&zname,1024);
                wname = DIR + "Wh1024.dat";
                xname = DIR + "Xh1024.dat";
                yname = DIR + "Yh1024.dat";
                WF2D = new FastFilter(FFTORDER,&wname,1024);
                XF2D = new FastFilter(FFTORDER,&xname,1024);
                YF2D = new FastFilter(FFTORDER,&yname,1024,1);
                wname = DIR + "WhFront1024.dat";
                xname = DIR + "XhFront1024.dat";
                yname = DIR + "YhFront1024.dat";
                WFf = new FastFilter(FFTORDER,&wname,1024);
                XFf = new FastFilter(FFTORDER,&xname,1024);
                YFf = new FastFilter(FFTORDER,&yname,1024,1);
                wname = DIR + "WhRear1024.dat";
                xname = DIR + "XhRear1024.dat";
                yname = DIR + "YhRear1024.dat";
                WFr = new FastFilter(FFTORDER,&wname,1024);
                XFr = new FastFilter(FFTORDER,&xname,1024);
                YFr = new FastFilter(FFTORDER,&yname,1024,1);
                wname = DIR + "h13.dat";
                xname = DIR + "h23.dat";
                h1fl = new FastFilter(FFTORDER,&wname,2048);
                h2fl = new FastFilter(FFTORDER,&xname,2048);
                h1fr = new FastFilter(FFTORDER,&wname,2048);
                h2fr = new FastFilter(FFTORDER,&xname,2048);
        }
}
void WigSound2::UnloadFilters()
{
        delete WF;
        delete XF;
        delete YF;
        delete ZF;
        delete WF2D;
        delete XF2D;
        delete YF2D;
        delete WFf;
        delete XFf;
        delete YFf;
        delete WFr;
        delete XFr;
        delete YFr;
        delete h1fl;
        delete h2fl;
        delete h1fr;
        delete h2fr;
}
void WigSound2::InitMem(   unsigned short usNoOfChannels,
                unsigned short usBufferLengthPerChannel,
                int SRate)
{
        SampleRate = SRate;
        Samples = AllocSampleBuffer(usNoOfChannels,
                  usBufferLengthPerChannel);
        ABuf = AmbiAllocate(usBufferLengthPerChannel,0,1);
        //BBuf used for 4-ear algorithms
        BBuf = AmbiAllocate(usBufferLengthPerChannel,0,1);

        SElev = new float[NoOfSpeakers];
        SAzim = new float[NoOfSpeakers];
        mono = new float[usBufferLengthPerChannel];
        for(int i=0;i<NoOfSpeakers;i++)
        {
                SElev[i]=0;
                SAzim[i]=(M_PI/(float)NoOfSpeakers)+
                  i*2*M_PI/(float)NoOfSpeakers;
        }
        Decode=AllocDecodeArray(NoOfSpeakers,0);
        DecoderCalc(SAzim,SElev,NoOfSpeakers,0,sqrt(2),Decode);

        WFile.WaveFile(Window->WFName.c_str());
        XFile.WaveFile(Window->XFName.c_str());
        YFile.WaveFile(Window->YFName.c_str());
        ZFile.WaveFile(Window->ZFName.c_str());
        Window->WaveSize = WFile.GetWaveSize();

        WAP = new AllPass(usBufferLengthPerChannel);
        XAP = new AllPass(usBufferLengthPerChannel);
        YAP = new AllPass(usBufferLengthPerChannel);
        WAP->SetCutOff(500.0f,(float)SRate);
        XAP->SetCutOff(500.0f,(float)SRate);
        YAP->SetCutOff(500.0f,(float)SRate);

        Application->GetNamePath();
        LoadFilters(SRate);
        Window->Oscilloscope1->Prepare();
        Window->Oscilloscope2->Prepare();
        UpdateFilter = UpdateRearFilter = true;
}
void WigSound2::UnInitMem( unsigned short usNoOfChannels,
                unsigned short usBufferLengthPerChannel)
{
        Window->Oscilloscope1->Unprepare();
        Window->Oscilloscope2->Unprepare();
        UnloadFilters();
        delete WAP;
        delete XAP;
        delete YAP;
        WFile.CloseWaveFile();
        XFile.CloseWaveFile();
        YFile.CloseWaveFile();
        ZFile.CloseWaveFile();
        FreeSampleBuffer(Samples,usNoOfChannels);
        delete[] mono;
        delete[] SAzim;
        delete[] SElev;
        FreeDecodeArray(Decode,0);
        AmbiFree(ABuf);
        AmbiFree(BBuf);
}
```

```
void WigSound2::MonitorAudio(WAVEHDR *pWaveHeader, unsigned short
usNoOfChannels,
                unsigned short usBufferLengthPerChannel)
{
        //Input Callback
        //Not Much Here as using Wave Files as input.
}
void WigSound2::ProcessAudio(WAVEHDR *pWaveHeader, unsigned short
usNoOfChannels,
                unsigned short usBufferLengthPerChannel)
{
        short *inPtr = (short *)ReadBuffer->lpData;
        short *outPtr = (short *)pWaveHeader->lpData;
        float yn;
        //Output Callback
        if(!Paused)
        {
                if(bSkip)
                {
                        bSkip = false;
                        //Scale Offset from 0->200 to 0->WaveSize
                        SkipOffset =
                              (long)(((double)SkipOffset/200.0)*
                              (double)WFile.GetWaveSize());
                        //Guarantee an even number (as offset is in
bytes)
                        //and wave file data is in shorts
                        SkipOffset = SkipOffset/2;
                        SkipOffset = SkipOffset*2;
                        //Offset all files
                        WFile.SkipIntoFile(SkipOffset);
                        XFile.SkipIntoFile(SkipOffset);
                        YFile.SkipIntoFile(SkipOffset);
                        ZFile.SkipIntoFile(SkipOffset);
                }
                switch(Window->AmbiInput->ItemIndex)
                {
                case 0:
                        //Wave File
                        WFile.GetWaveSamples(ABuf->W,ABuf->Length);
                        XFile.GetWaveSamples(ABuf->X,ABuf->Length);
                        YFile.GetWaveSamples(ABuf->Y,ABuf->Length);
                        ZFile.GetWaveSamples(ABuf->Z,ABuf->Length);
                        Window->UpdateWaveTime(WFile.GetWaveRead());
                        break;
                case 1:
                        //Mono in to be panned
                        WFile.GetWaveSamples(mono,ABuf->Length);
                        Window->UpdateWaveTime(WFile.GetWaveRead());
                        Mono2B(mono,ABuf,Window->monopan,0.0f);
                        break;
                case 2:
                        //Live in
                        DeInterlace(ReadBuffer,
                              Samples,usNoOfChannels);
                        break;

                }

                BTilt(ABuf,Window->TiltAngle);
                BRotate(ABuf,Window->RotAngle);
                const float vol = Window->m_volume;
```

```
switch(Window->m_effect)
{
        case 0:
                WAP->ProcessAudio(ABuf->W,1.33,1.15);
                XAP->ProcessAudio(ABuf->X,1.33,1.15);
                YAP->ProcessAudio(ABuf->Y,1.33,1.15);
                B2Speakers(Decode,ABuf,Samples,
                    usNoOfChannels,8,0);
                break;
        case 1:
                B2Headphones(ABuf,Samples,
                    usNoOfChannels);
                break;
        case 2:
                B2Headphones2D(ABuf,Samples,
                    usNoOfChannels);
                break;

        case 3:
                if(UpdateFilter)
                {
                        ChooseFilter(SampleRate);
                        UpdateFilter = false;
                }
                B2Headphones(ABuf,Samples,
                    usNoOfChannels);
                B2Trans(ABuf,Samples[0],Samples[1],
                usNoOfChannels,h1fl,h2fl,h1fr,h2fr);
                break;
        case 4:
                if(UpdateFilter)
                {
                        ChooseFilter(SampleRate);
                        UpdateFilter = false;
                }
                if(UpdateRearFilter)
                {
                        ChooseRearFilter(SampleRate);
                        UpdateRearFilter = false;
                }
                B2Headphones4(ABuf,BBuf,
                    Samples,usNoOfChannels);
                B2Trans(ABuf,Samples[0],Samples[1],
                usNoOfChannels,h1fl,h2fl,h1fr,h2fr);
                if(usNoOfChannels>=4)
                        B2Trans(ABuf,Samples[2],
                          Samples[3],
                          usNoOfChannels,h1rl,h2rl,
                          h1rr,h2rr);
                break;
        case 5:
                if(UpdateFilter)
                {
                        ChooseFilter(SampleRate);
                        UpdateFilter = false;
                }
                B2Trans(ABuf,Samples[0],Samples[1],

                usNoOfChannels,h1fl,h2fl,h1fr,h2fr);
                break;
        default:
                B2Speakers(Decode,ABuf,Samples,
                                usNoOfChannels,8,0);
                        break;
                }

                //Do Volume
                for(int i=0;i<usBufferLengthPerChannel;i++)
                {
                        for(int j=0;j<usNoOfChannels;j++)
                        {
                                Samples[j][i]*= vol;
                        }
                }
                Window->Oscilloscope1->SampleArray = Samples[0];
                Window->Oscilloscope2->SampleArray = Samples[1];
                Window->Oscilloscope1->UpdateGraph();
                Window->Oscilloscope2->UpdateGraph();
                ReInterlace(pWaveHeader,Samples,usNoOfChannels);
        }
        else
        {
                memset(pWaveHeader->lpData,0,
                        pWaveHeader->dwBufferLength);
        }
}
void WigSound2::SkipAudio(int Offset)
{
        SkipOffset = (unsigned long)Offset;
        bSkip = true;
}
void WigSound2::B2Headphones(AmbiBuffer *Signal, float **Samples,int
NoOfChannels)
{
        const int Len = Signal->Length;
        const float Wid = Window->m_width;
        if(Window->m_effect==1 || Window->m_effect==2)
        {
                WF->OverAddFir(Signal->W,Wid);
                XF->OverAddFir(Signal->X,Wid);
                YF->OverAddFir(Signal->Y,Wid);
                if(ZF)
                        ZF->OverAddFir(Signal->Z,Wid);
        }
        else
        {
                WF->OverAddFir(Signal->W);
                XF->OverAddFir(Signal->X);
                YF->OverAddFir(Signal->Y);
                if(ZF)
                        ZF->OverAddFir(Signal->Z);
        }

        for(int i=0;i<Len;i++)
        {
                Samples[0][i] = 0.5*(Signal->W[i] + Signal->X[i] +
                  Signal->Y[i] + Signal->Z[i]);
                Samples[1][i] = 0.5*(Signal->W[i] + Signal->X[i] –
                  Signal->Y[i] + Signal->Z[i]);
        }

        for(int i=2;i<NoOfChannels;i++)
        {
                for(int j=0;j<Len;j++)
                Samples[i][j] = 0.0f;
        }
}
void WigSound2::B2Headphones4(AmbiBuffer *Signal,
            AmbiBuffer *Signal2, float **Samples,int NoOfChannels)
{
        const int Len = Signal->Length;
        if(NoOfChannels>=4)
        {
                memcpy(Signal2->W,Signal->W,sizeof(float)*Len);
                memcpy(Signal2->X,Signal->X,sizeof(float)*Len);
                memcpy(Signal2->Y,Signal->Y,sizeof(float)*Len);

                WFf->OverAddFir(Signal->W);
                XFf->OverAddFir(Signal->X);
                YFf->OverAddFir(Signal->Y);
                WFr->OverAddFir(Signal2->W);
                XFr->OverAddFir(Signal2->X);
                YFr->OverAddFir(Signal2->Y);
                for(int i=0;i<Len;i++)
                {
                        Samples[0][i] = Signal->W[i]  + Signal->X[i]
                                  + Signal->Y[i];
                        Samples[1][i] = Signal->W[i]  + Signal->X[i]
                                  - Signal->Y[i];
                        Samples[2][i] = Signal2->W[i] + Signal2->X[i]
                                  + Signal2->Y[i];
                        Samples[3][i] = Signal2->W[i] + Signal2->X[i]
                                  - Signal2->Y[i];
                }

                for(int i=4;i<NoOfChannels;i++)
                {
                        for(int j=0;j<Len;j++)
                                Samples[i][j] = 0.0f;
                }
        }
}
void WigSound2::B2Headphones2D(AmbiBuffer *Signal,
                              float **Samples,int NoOfChannels)
{
        const int Len = Signal->Length;
        const float Wid = Window->m_width;
        if(Window->m_effect==1 || Window->m_effect==2)
        {
                WF2D->OverAddFir(Signal->W,Wid);
                XF2D->OverAddFir(Signal->X,Wid);
                YF2D->OverAddFir(Signal->Y,Wid);
        }
        else
        {
                WF2D->OverAddFir(Signal->W);
                XF2D->OverAddFir(Signal->X);
                YF2D->OverAddFir(Signal->Y);
        }

        for(int i=0;i<Len;i++)
        {
                Samples[0][i] = Signal->W[i]
                              + Signal->X[i]
                              + Signal->Y[i];
                Samples[1][i] = Signal->W[i]
```

```
                              + Signal->X[i]
                              - Signal->Y[i];
        }

        for(int i=2;i<NoOfChannels;i++)
        {
                for(int j=0;j<Len;j++)
                        Samples[i][j] = 0.0f;
        }
}
void WigSound2::B2Trans(AmbiBuffer *Signal,float *Left,
                  float *Right,int NoOfChannels,
                  FastFilter *h1l, FastFilter *h2l,
                  FastFilter *h1r, FastFilter *h2r)
{
        const int Len = Signal->Length;
        const float Width = Window->m_width;
        float *tL = new float[Signal->Length];
        float *tR = new float[Signal->Length];

        memcpy(tL,Left,sizeof(float)*Len);
        memcpy(tR,Right,sizeof(float)*Len);

        h1l->OverAddFir(Left);
        h2l->OverAddFir(tL);
        h1r->OverAddFir(Right);
        h2r->OverAddFir(tR);

        for(int i=0;i<Len;i++)
        {
                Left[i] = Left[i] + (Width * tR[i]);
                Right[i] = Right[i] + (Width * tL[i]);
        }

        delete[] tL;
        delete[] tR;
}
void WigSound2::ChooseFilter(int SRate)
{
        AnsiString h1name,h2name;
        if(SRate==44100)
        {
                switch(Window->TransFilter->ItemIndex)
                {
                case 0:
                        h1name = DIR + "h13.dat";
                        h2name = DIR + "h23.dat";
                        break;
                case 1:
                        h1name = DIR + "h15.dat";
                        h2name = DIR + "h25.dat";
                        break;
                case 2:
                        h1name = DIR + "h110.dat";
                        h2name = DIR + "h210.dat";
                        break;
                case 3:
                        h1name = DIR + "h120.dat";
                        h2name = DIR + "h220.dat";
                        break;
                case 4:
                        h1name = DIR + "h130.dat";
```

```
                                h2name = DIR + "h230.dat";
                                break;
                        case 5:
                                h1name = DIR + "h13b.dat";
                                h2name = DIR + "h23b.dat";
                                break;
                        }
                }
                else if(SRate==48000)
                {
                        switch(Window->TransFilter->ItemIndex)
                        {
                        case 0:
                                h1name = DIR + "h1348.dat";
                                h2name = DIR + "h2348.dat";
                                break;
                        case 1:
                                h1name = DIR + "h1548.dat";
                                h2name = DIR + "h2548.dat";
                                break;
                        case 2:
                                h1name = DIR + "h11048.dat";
                                h2name = DIR + "h21048.dat";
                                break;
                        case 3:
                                h1name = DIR + "h12048.dat";
                                h2name = DIR + "h22048.dat";
                                break;
                        case 4:
                                h1name = DIR + "h13048.dat";
                                h2name = DIR + "h23048.dat";
                                break;
                        case 5:
                                h1name = DIR + "h13b48.dat";
                                h2name = DIR + "h23b48.dat";
                                break;
                        }
                }
                delete h1fl;
                delete h2fl;
                delete h1fr;
                delete h2fr;
                h1fl = new FastFilter(FFTORDER,&h1name,2048);
                h2fl = new FastFilter(FFTORDER,&h2name,2048);
                h1fr = new FastFilter(FFTORDER,&h1name,2048);
                h2fr = new FastFilter(FFTORDER,&h2name,2048);
}
void WigSound2::ChooseRearFilter(int SRate)
{
                AnsiString h1name,h2name;
                if(SRate==44100)
                {
                        switch(Window->RearFilter->ItemIndex)
                        {
                        case 0:
                                h1name = DIR + "h1175.dat";
                                h2name = DIR + "h2175.dat";
                                break;
                        case 1:
                                h1name = DIR + "h1170.dat";
                                h2name = DIR + "h2170.dat";
                                break;
                        case 2:
                                h1name = DIR + "h1160.dat";
                                h2name = DIR + "h2160.dat";
                                break;
                        case 3:
                                h1name = DIR + "h1150.dat";
                                h2name = DIR + "h2150.dat";
                                break;
                        case 4:
                                h1name = DIR + "h1110.dat";
                                h2name = DIR + "h2110.dat";
                                break;
                        }
                }
                else if(SRate==48000)
                {
                        switch(Window->RearFilter->ItemIndex)
                        {
                        case 0:
                                h1name = DIR + "h117548.dat";
                                h2name = DIR + "h217548.dat";
                                break;
                        case 1:
                                h1name = DIR + "h117048.dat";
                                h2name = DIR + "h217048.dat";
                                break;
                        case 2:
                                h1name = DIR + "h116048.dat";
                                h2name = DIR + "h216048.dat";
                                break;
                        case 3:
                                h1name = DIR + "h115048.dat";
                                h2name = DIR + "h215048.dat";
                                break;
                        case 4:
                                h1name = DIR + "h111048.dat";
                                h2name = DIR + "h211048.dat";
                                break;
                        }
                }
                h1rl = new FastFilter(FFTORDER,&h1name,2048);
                h2rl = new FastFilter(FFTORDER,&h2name,2048);
                h1rr = new FastFilter(FFTORDER,&h1name,2048);
                h2rr = new FastFilter(FFTORDER,&h2name,2048);
        }
        #endif
```

```
//---------------------------------------------------------------------
//--------------------------ALLPASS.H-------------------------------
//---------------------------------------------------------------------
#ifndef HALLPASS
#define HALLPASS

#include <math.h>
//-------------------------------------------------------------------
//-------------------------------------------------------------------
class AllPass
{
private:
      float fs,fc,alpha,*Buffer;
        float ff,fb,in,out;
      const int BufLen;
      void DoAllPass(float *signal, int iLen, float aval);
public:
      AllPass(int iLen);
      ~AllPass();
      void SetCutOff(float fcut, float fsam);
      void ProcessAudio(float *signal, float dBLP, float dBHP,
                              bool dummy);
      void ProcessAudio(float *signal, float LinLP, float LinHP);
};
//-------------------------------------------------------------------
//-------------------------------------------------------------------
AllPass::AllPass(int iLen) : BufLen(iLen)
{
      //Constructor - Set Default Cutoff, incase user doesn't ;-)
      SetCutOff(700.0f,44100.0f);
      ff=fb=in=out=0.0f;
      Buffer = new float[BufLen];
}
AllPass::~AllPass()
{
      delete[] Buffer;
}
inline void AllPass::SetCutOff(float fcut,float fsam)
{
      fs = fsam;
      fc = fcut;

      float fcnorm = fc/fs;
      float w = 2*M_PI*fcnorm;
      float cw = cos(w);

      alpha = ((2-sqrt(pow(-2,2) - 4 * cw * cw)))/(2*cw);
}
//-------------------------------------------------------------------
inline void AllPass::DoAllPass(float *signal, int iLen, float aval)
{
        float a,b;
        a = ff;
        b = fb;
      for(int i=0;i<iLen;i++)
      {
            out = (aval * signal[i]) - ff + (aval * fb);
            fb = out;
            ff = signal[i];
            signal[i] = out;
      }
}
```

```
//-------------------------------------------------------------------
void AllPass::ProcessAudio(float *signal, float dBLP, float dBHP
                              , bool dummy)
{
      float LinLP,LinHP,HP,LP;
      LinLP = pow(10,dBLP/20);
      LinHP = pow(10,dBHP/20);

      memcpy(Buffer,signal,sizeof(float) * BufLen);
      DoAllPass(Buffer,BufLen,alpha);

      for(int i=0;i<BufLen;i++)
      {
            HP = 0.5 * (signal[i] + Buffer[i]);
            LP = 0.5 * (signal[i] - Buffer[i]);
            signal[i] = LP * LinLP + HP * LinHP;
      }
}
//-------------------------------------------------------------------
void AllPass::ProcessAudio(float *signal, float LinLP, float LinHP)
{
      float HP,LP;
      memcpy(Buffer,signal,sizeof(float) * BufLen);
      DoAllPass(Buffer,BufLen,alpha);

      for(int i=0;i<BufLen;i++)
      {
            HP = 0.5 * (signal[i] + Buffer[i]);
            LP = 0.5 * (signal[i] - Buffer[i]);
            signal[i] = (LP * LinLP) + (HP * LinHP);
      }
}
//-------------------------------------------------------------------
#endif
```

```
//---------------------------------------------------------------------
//---------------------------FASTFILTER.H----------------------------
//---------------------------------------------------------------------
#ifndef HFASTCONV
#define HFASTCONV

#ifndef nsp_UsesTransform
        extern "C"      {
        #define nsp_UsesTransform
        #include "nsp.h"
        }
#endif

#include <math.h>
#include <fstream.h>

class FastFilter
{
private:
        int order,fftsize,siglen,implen;
        float *OldArray,*Signal,*tconv,*h;
        SCplx *fh,*fSig,*fconv;
public:
        FastFilter(int FFTOrder,AnsiString *FName,int FLength);
        FastFilter(int FFTOrder,AnsiString *FName,
                        int FLength,bool inv);
        void ReLoadFilter(AnsiString *FName,int FLength);
        ~FastFilter();
        void OverAddFir(float *signal);
        void OverAddFir(float *signal,float g);
};
//---------------------------------------------------------------------
FastFilter::FastFilter(int FFTOrder,AnsiString *FName,int FLength)
{
        order = FFTOrder;
        fftsize = pow(2,order);
        siglen = (fftsize/2) + 1;
        implen = fftsize/2;

        OldArray = new float[fftsize];
        Signal = new float[fftsize];
        tconv = new float[fftsize];
        h = new float[fftsize];

        fh = new SCplx[fftsize];
        fSig = new SCplx[fftsize];
        fconv = new SCplx[fftsize];

        ReLoadFilter(FName,FLength);

        nspsRealFftNip(NULL,NULL,order,NSP_Init);
        nspsRealFftNip(h,fh,order,NSP_Forw);
}
//---------------------------------------------------------------------
FastFilter::FastFilter(int FFTOrder,AnsiString *FName,int
FLength,bool inv)
{
        order = FFTOrder;
        fftsize = pow(2,order);
        siglen = (fftsize/2) + 1;
        implen = fftsize/2;
        OldArray = new float[fftsize];
        Signal = new float[fftsize];
        tconv = new float[fftsize];
        h = new float[fftsize];

        fh = new SCplx[fftsize];
        fSig = new SCplx[fftsize];
        fconv = new SCplx[fftsize];

        ReLoadFilter(FName,FLength);
        for(int i=0;i<FLength;i++)
        {
                h[i] = -h[i];
        }
        nspsRealFftNip(NULL,NULL,order,NSP_Init);
        nspsRealFftNip(h,fh,order,NSP_Forw);
}
//---------------------------------------------------------------------
FastFilter::~FastFilter()
{
        delete[] tconv;
        delete[] OldArray;
        delete[] Signal;
        delete[] h;

        delete[] fh;
        delete[] fSig;
        delete[] fconv;
}
//---------------------------------------------------------------------
void FastFilter::ReLoadFilter(AnsiString *FName,int FLength)
{
        FILE *f;
        int c;

        memset(OldArray,0,sizeof(float)*fftsize);
        memset(Signal,0,sizeof(float)*fftsize);
        memset(tconv,0,sizeof(float)*fftsize);
        memset(h,0,sizeof(float)*fftsize);

        memset(fh,0,sizeof(SCplx)*fftsize);
        memset(fSig,0,sizeof(SCplx)*fftsize);
        memset(fconv,0,sizeof(SCplx)*fftsize);

        f = fopen(FName->c_str(),"rb");
        if(f)
        {
                c = fread(h,sizeof(float),FLength,f);
                if(c!=FLength)
                        MessageBox(NULL,FName->c_str(),
                              "Wrong Filter Length",NULL);
                fclose(f);
        }
        else
                MessageBox(NULL,FName->c_str(),"Couldn't Open
File",NULL);
}
//---------------------------------------------------------------------
void FastFilter::OverAddFir(float *signal)
{
        static unsigned int i,j=0,k;
        memcpy(Signal,signal,siglen*sizeof(float));
        //FFT Real Input Signal
        nspsRealFftNip(Signal,fSig,order,NSP_Forw);

        //Do processing in unrolled loop to maximise pipeline
        //usage
        for(i=0;i<implen;i+=4)
        {
                fconv[i].re =   (fh[i].re   * fSig[i].re) -
                                (fh[i].im   * fSig[i].im);
                fconv[i].im =   (fh[i].re   * fSig[i].im) +
                                (fh[i].im   * fSig[i].re);
                fconv[i+1].re = (fh[i+1].re * fSig[i+1].re) -
                                (fh[i+1].im * fSig[i+1].im);
                fconv[i+1].im = (fh[i+1].re * fSig[i+1].im) +
                                (fh[i+1].im * fSig[i+1].re);
                fconv[i+2].re = (fh[i+2].re * fSig[i+2].re) -
                                (fh[i+2].im * fSig[i+2].im);
                fconv[i+2].im = (fh[i+2].re * fSig[i+2].im) +
                                (fh[i+2].im * fSig[i+2].re);
                fconv[i+3].re = (fh[i+3].re * fSig[i+3].re) -
                                (fh[i+3].im * fSig[i+3].im);
                fconv[i+3].im = (fh[i+3].re * fSig[i+3].im) +
                                (fh[i+3].im * fSig[i+3].re);
        }
        fconv[i+1].re = (fh[i+1].re * fSig[i+1].re) -
                        (fh[i+1].im * fSig[i+1].im);
        fconv[i+1].im = (fh[i+1].re * fSig[i+1].im) +
                        (fh[i+1].im * fSig[i+1].re);

        //do inverse FFT
        nspsCcsFftNip(fconv,tconv,order,NSP_Inv);

        //Do overlap add
        for(i=0;i<siglen;i++)
                signal[i]=(tconv[i]+OldArray[i]);

        //update storage of 'old' samples
        for(i=siglen,k=0;i<siglen+implen-1;i++,k++)
        {
                OldArray[k]=tconv[i];
                OldArray[i]=0;
        }
}
//---------------------------------------------------------------------
void FastFilter::OverAddFir(float *signal, float g)
{
        static unsigned int i,j=0,k;
        memcpy(Signal,signal,siglen*sizeof(float));

        //FFT Real Input Signal
        nspsRealFftNip(Signal,fSig,order,NSP_Forw);

        //Do processing in unrolled loop to maximise pipeline
        //usage
        for(i=0;i<implen;i+=4)
        {
                fconv[i].re =   (fh[i].re   * fSig[i].re) -
                                (fh[i].im   * fSig[i].im);
                fconv[i].im =   (fh[i].re   * fSig[i].im) +
                                (fh[i].im   * fSig[i].re);
                fconv[i+1].re = (fh[i+1].re * fSig[i+1].re) -
```

```
                                (fh[i+1].im * fSig[i+1].im);
                fconv[i+1].im = (fh[i+1].re * fSig[i+1].im) +
                                (fh[i+1].im * fSig[i+1].re);
                fconv[i+2].re = (fh[i+2].re * fSig[i+2].re) -
                                (fh[i+2].im * fSig[i+2].im);
                fconv[i+2].im = (fh[i+2].re * fSig[i+2].im) +
                                (fh[i+2].im * fSig[i+2].re);
                fconv[i+3].re = (fh[i+3].re * fSig[i+3].re) -
                                (fh[i+3].im * fSig[i+3].im);
                fconv[i+3].im = (fh[i+3].re * fSig[i+3].im) +
                                (fh[i+3].im * fSig[i+3].re);
        }
        fconv[i+1].re = (fh[i+1].re * fSig[i+1].re) -
                        (fh[i+1].im * fSig[i+1].im);
        fconv[i+1].im = (fh[i+1].re * fSig[i+1].im) +
                        (fh[i+1].im * fSig[i+1].re);

        //do inverse FFT
        nspsCcsFftNip(fconv,tconv,order,NSP_Inv);

        //Do overlap add
        for(i=0;i<siglen;i++)
                signal[i]=((1.0f - g) * signal[i]) +
                  (g * (tconv[i]+OldArray[i]));

        //update storage of 'old' samples
        for(i=siglen,k=0;i<siglen+implen-1;i++,k++)
        {
                OldArray[k]=tconv[i];
                OldArray[i]=0;
        }
}
//---------------------------------------------------------------------
#endif
```

```
//---------------------------------------------------------------------
//----------------------------WIGFILE.H--------------------------------
//---------------------------------------------------------------------
#ifndef WaveFileH
#define WaveFileH

#include <windows.h>
#include <mmsystem.h>

class WigFile
{
private:
        HMMIO FileHandle;
        MMCKINFO FileInfo,CkInfo,CkSubInfo;
        MMIOINFO IoInfo;
        long WaveSize,WavRead,InitialOffset;
        //char FileBuffer[16384];
public:
        WigFile();
        ~WigFile();
        void WaveFile(char *FileName);
        void GetWaveSamples(float *samples, UINT length);
        void SkipIntoFile(long Skip);
        void CloseWaveFile();
        unsigned long GetWaveSize()     {return(WaveSize);};
        unsigned long GetWaveRead()     {return(WavRead);};
        PCMWAVEFORMAT WaveFormat;
};
//---------------------------------------------------------------------
//Function Declarations------------------------------------------------
//---------------------------------------------------------------------
WigFile::WigFile()
{
}
//---------------------------------------------------------------------
WigFile::~WigFile()
{
}
//---------------------------------------------------------------------
void WigFile::WaveFile(char *FileName)
{
                FileHandle = mmioOpen(FileName,NULL,
                              MMIO_READ|MMIO_ALLOCBUF);
                if(FileHandle==NULL){
                        return;
                }

                CkInfo.fccType=mmioFOURCC('W','A','V','E');

                if(mmioDescend(FileHandle,&CkInfo,
                        NULL,MMIO_FINDRIFF))
                {
                        mmioClose(FileHandle,0);
                        ShowMessage("Invalid WaveFormat for file: "
                              + *FileName);
                }
                CkSubInfo.ckid = mmioFOURCC('f','m','t',' ');
                if(mmioDescend(FileHandle,&CkSubInfo,
                                    &CkInfo,MMIO_FINDCHUNK))
                {
                        mmioClose(FileHandle,0);
                        ShowMessage("Invalid Format Chunk for file: "
                      + *FileName);
                }
                unsigned long n = CkSubInfo.cksize;
                mmioRead(FileHandle,(LPSTR)&WaveFormat,n);

                if(WaveFormat.wf.wFormatTag!=WAVE_FORMAT_PCM)
                {
                        mmioClose(FileHandle,0);
                        ShowMessage(*FileName
                              + " is not a Wave File!");
                }

                mmioAscend(FileHandle,&CkSubInfo,0);
                CkSubInfo.ckid = mmioFOURCC('d','a','t','a');

                if(mmioDescend(FileHandle,&CkSubInfo,
                        &CkInfo,MMIO_FINDCHUNK))
                {
                        mmioClose(FileHandle,0);
                        ShowMessage("Could not descend into
                                          data chunk: " + *FileName);
                }
                WavRead = 0;
                WaveSize = CkSubInfo.cksize;
                InitialOffset = CkSubInfo.dwDataOffset;
}
//---------------------------------------------------------------------
void WigFile::GetWaveSamples(float *samples, UINT length)
{
        long c1;
        short *buf = new short[length];
        //Offset file reading by Pos bytes
        if(FileHandle)
        {
                c1 = mmioRead(FileHandle,(char *)buf,length * 2);
                //Increase wavefile position counter
                if(c1<=0)       WavRead=WaveSize;
                else            WavRead+=c1;
                if(WavRead<WaveSize)
                {
                        for(int i=0;i<c1/2;i++)
                        {
                                samples[i] = (float)(buf[i]);
                        }
                        for(int i=c1/2;i<length;i++)
                        {
                                samples[i] = 0.0f;
                        }
                }
                if(c1<=0)
                {
                        if(FileHandle)
                        {
                                mmioClose(FileHandle,0);
                                FileHandle = NULL;
                        }
                }
        }
        else
        {
                for(int i=0;i<length;i++)
                {
                        samples[i] = 0.0f;
                }
        }
        delete[] buf;
}
//---------------------------------------------------------------------
void WigFile::SkipIntoFile(long Skip)
{
        long res = mmioSeek(FileHandle,Skip +
                        InitialOffset,SEEK_SET);
        WavRead = res - InitialOffset;
}
void WigFile::CloseWaveFile()
{
        if(FileHandle)
                mmioClose(FileHandle,0);
        FileHandle=NULL;
}
#endif
```

```
//---------------------------------------------------------------------
//-------------------------WIGAMBI.H-----------------------------------
//---------------------------------------------------------------------
#ifndef WigAmbiH
#define WigAmbiH

#include <math.h>
#include <mmsystem.h>

#ifndef nsp_UsesTransform
        extern "C"      {
        #define nsp_UsesTransform
        #include "nsp.h"
        }
#endif

struct AmbiBuffer
{
        float *W,*X,*Y,*Z,*R,*S,*T,*U,*V;
        int Length;
        bool Order;
};

void DeInterlace(WAVEHDR *,float **,int NoOfChannels);
void ReInterlace(WAVEHDR *,float **,int NoOfChannels);
void BGain(AmbiBuffer *,float Gain);
void BRotate(AmbiBuffer *,float RadAngle);
void BTilt(AmbiBuffer *,float RadAngle);
void Mono2B(float *Mono,AmbiBuffer *,float RadAzim, float RadElev);
void BPlusB(AmbiBuffer *,AmbiBuffer *);
void AssignChannel(AmbiBuffer *,float *,char);
AmbiBuffer * AmbiAllocate(int Length,bool Order,bool WithChannels);
void AmbiFree(AmbiBuffer *);
float ** AllocDecodeArray(int NoOfSpeakers,bool Order);
float ** AllocSampleBuffer(int Channels,int BufferLength);
void FreeDecodeArray(float **,bool Order);
void FreeSampleBuffer(float **,int Channels);
void DecoderCalc(float *Azim,float *Elev,int NoOfSpeakers,bool Order,
                  float WGain,float **Gains);
void B2Speakers(float **SGains,AmbiBuffer *Ambi, float **Samples,
                  int NoOfChannels,int NoOfSpeakers,bool Order);
float MaxSample(float *Samples,int BufferLength);
void MaxSample(WAVEHDR *,float *,int BufferLength,int NoOfChannels);
//------------------------------------------------------------------
float MaxSample(float *Samples,int BufferLength)
{
        float Max=0;
        for(int i=0;i<BufferLength;i++)
                if(Max<Samples[i])  Max=Samples[i];

        return (Max);
}
//------------------------------------------------------------------
void MaxSample(WAVEHDR *pWaveHeader,float *Max,int BufferLength,
                  int NoOfChannels)
{
        for(int i=0;i<NoOfChannels;i++) Max[i]=0;
        short *Data=(short *)pWaveHeader->lpData;
        for(int i=0;i<BufferLength;i++)
        {
                for(int j=0;j<NoOfChannels;j++)
                {
                        if(Max[j]<(float)Data[j])
                              Max[j]=(float)Data[j];
                }
                Data+=NoOfChannels;
        }
}
//------------------------------------------------------------------
void DeInterlace(WAVEHDR *WaveBuffer,float **Samples,
                  int NoOfChannels)
{
        //Sort out channels
        short *Buffer = (short *)WaveBuffer->lpData;
        int count=0;
        for(unsigned int i=0;
                  i<WaveBuffer->dwBufferLength/(2*NoOfChannels);i++)
        {
                for(int j=0;j<NoOfChannels;j++)
                {
                        Samples[j][i]=Buffer[count++];
                }
        }
}
//------------------------------------------------------------------
void ReInterlace(WAVEHDR *WaveBuffer,float **Samples,
                  int NoOfChannels)
{
        //Sort out channels
        short *Buffer = (short *)WaveBuffer->lpData;
        int count=0;
        for(unsigned int i=0;
                  i<WaveBuffer->dwBufferLength/(2*NoOfChannels);i++)
        {
                for(int j=0;j<NoOfChannels;j++)
                {
                        Buffer[count++]=(short)Samples[j][i];
                }
        }
}
//------------------------------------------------------------------
void BRotate(AmbiBuffer *a,float RadAngle)
{
        float x,y;
        float s = sin(RadAngle);
        float c = cos(RadAngle);
        for(int i=0;i<a->Length;)
        {
                x = a->X[i] * c + a->Y[i] * s;
                y = a->Y[i] * c + a->X[i] * s;
                a->X[i] = x;
                a->Y[i] = y;
                i++;
        }
}
void BTilt(AmbiBuffer *a,float RadAngle)
{
        float x,z;
        float s = sin(RadAngle);
        float c = cos(RadAngle);
        for(int i=0;i<a->Length;)
        {
                x = a->X[i] * c - a->Z[i] * s;
                z = a->Z[i] * c + a->X[i] * s;
                a->X[i] = x;
                a->Z[i] = z;
                i++;
        }
}
void BGain(AmbiBuffer *Ambi, float Gain)
{
        if(Ambi->Order)
        {
                for(int i=0;i<Ambi->Length;i++)
                {
                        Ambi->W[i]*=Gain;
                        Ambi->X[i]*=Gain;
                        Ambi->Y[i]*=Gain;
                        Ambi->Z[i]*=Gain;
                        Ambi->R[i]*=Gain;
                        Ambi->S[i]*=Gain;
                        Ambi->T[i]*=Gain;
                        Ambi->U[i]*=Gain;
                        Ambi->V[i]*=Gain;
                }
        }
        else
        {
                for(int i=0;i<Ambi->Length;i++)
                {
                        Ambi->W[i]*=Gain;
                        Ambi->X[i]*=Gain;
                        Ambi->Y[i]*=Gain;
                        Ambi->Z[i]*=Gain;
                }
        }
}
//------------------------------------------------------------------
void Mono2B(float *Mono,AmbiBuffer *Ambi,float RadAzim,
                  float RadElev)
{
        float SinA=sin(RadAzim);
        float CosA=cos(RadAzim);
        float SinE=sin(RadElev);
        float CosE=cos(RadElev);
        float Sin2E=sin(2*RadElev);
        float Sin2A=sin(2*RadAzim);
        float Cos2A=cos(2*RadAzim);
        float Sample,Gain[9];

        Gain[0] = 0.70710678119f;
        Gain[1] = CosA * CosE;
        Gain[2] = SinA * CosE;
        Gain[3] = SinE;
        if(Ambi->Order)
        {
                Gain[4] = 1.5f*SinE*SinE-0.5f;
                Gain[5] = CosA*Sin2E;
                Gain[6] = SinA*Sin2E;
                Gain[7] = Cos2A*CosE*CosE;
                Gain[8] = Sin2A*CosE*CosE;

                for(int i=0;i<Ambi->Length;i++)
                {
                        Sample=Mono[i];
                        Ambi->W[i]=Sample*Gain[0];
                        Ambi->X[i]=Sample*Gain[1];
                        Ambi->Y[i]=Sample*Gain[2];
                        Ambi->Z[i]=Sample*Gain[3];
                        Ambi->R[i]=Sample*Gain[4];
                        Ambi->S[i]=Sample*Gain[5];
                        Ambi->T[i]=Sample*Gain[6];
                        Ambi->U[i]=Sample*Gain[7];
                        Ambi->V[i]=Sample*Gain[8];
                }
        }
        else
        {
                for(int i=0;i<Ambi->Length;i++)
                {
                        Sample=Mono[i];
                        Ambi->W[i]=Sample*Gain[0];
                        Ambi->X[i]=Sample*Gain[1];
                        Ambi->Y[i]=Sample*Gain[2];
                        Ambi->Z[i]=Sample*Gain[3];
                }
        }
}
//------------------------------------------------------------------
void BPlusB(AmbiBuffer *Ambi1,AmbiBuffer *Ambi2)
{
        if(Ambi1->Order && Ambi2->Order)
        {
                for(int i=0;i<Ambi1->Length;i++)
                {
                        Ambi2->W[i]+=Ambi1->W[i];
                        Ambi2->X[i]+=Ambi1->X[i];
                        Ambi2->Y[i]+=Ambi1->Y[i];
                        Ambi2->Z[i]+=Ambi1->Z[i];
                        Ambi2->R[i]+=Ambi1->R[i];
                        Ambi2->S[i]+=Ambi1->S[i];
                        Ambi2->T[i]+=Ambi1->T[i];
                        Ambi2->U[i]+=Ambi1->U[i];
                        Ambi2->V[i]+=Ambi1->V[i];
                }
        }
        else
        {
                for(int i=0;i<Ambi1->Length;i++)
                {
                        Ambi2->W[i]+=Ambi1->W[i];
                        Ambi2->X[i]+=Ambi1->X[i];
                        Ambi2->Y[i]+=Ambi1->Y[i];
                        Ambi2->Z[i]+=Ambi1->Z[i];
                }
        }
}
//------------------------------------------------------------------
AmbiBuffer * AmbiAllocate(int Length,bool Order,bool WithChannels)
{
            AmbiBuffer *Ambi;
            Ambi = new AmbiBuffer;

            if(WithChannels)
            {
                  Ambi->W = new float[Length];
                        memset(Ambi->W,0,sizeof(float)*Length);
                  Ambi->X = new float[Length];
```

```
                        memset(Ambi->X,0,sizeof(float)*Length);
                  Ambi->Y = new float[Length];
                        memset(Ambi->Y,0,sizeof(float)*Length);
                  Ambi->Z = new float[Length];
                        memset(Ambi->Z,0,sizeof(float)*Length);
                  if(Order)
                  {
                              Ambi->R = new float[Length];
                              Ambi->S = new float[Length];
                              Ambi->T = new float[Length];
                              Ambi->U = new float[Length];
                              Ambi->V = new float[Length];
                  }
            }
            Ambi->Length=Length;
            Ambi->Order=Order;
            return(Ambi);
}
//------------------------------------------------------------------
void AmbiFree(AmbiBuffer *Ambi)
{
        if(Ambi->W)     delete [] Ambi->W;
        if(Ambi->X)     delete [] Ambi->X;
        if(Ambi->Y)     delete [] Ambi->Y;
        if(Ambi->Z)     delete [] Ambi->Z;
        if(Ambi->R && Ambi->Order)     delete [] Ambi->R;
        if(Ambi->S && Ambi->Order)     delete [] Ambi->S;
        if(Ambi->T && Ambi->Order)     delete [] Ambi->T;
        if(Ambi->U && Ambi->Order)     delete [] Ambi->U;
        if(Ambi->V && Ambi->Order)     delete [] Ambi->V;

        delete Ambi;
}
//------------------------------------------------------------------
void AssignChannel(AmbiBuffer *Ambi,float *Samples,char Channel)
{
      switch (Channel)
      {
      case 'W':
            Ambi->W=Samples;
            break;
      case 'X':
            Ambi->X=Samples;
            break;
      case 'Y':
            Ambi->Y=Samples;
            break;
      case 'Z':
            Ambi->Z=Samples;
            break;
      case 'R':
            Ambi->R=Samples;
            break;
      case 'S':
            Ambi->S=Samples;
            break;
      case 'T':
            Ambi->T=Samples;
            break;
      case 'U':
            Ambi->U=Samples;
            break;
      case 'V':
            Ambi->V=Samples;
            break;
      default:
            break;
      }
}
//------------------------------------------------------------------
float ** AllocSampleBuffer(int Channels, int BufferLength)
{
        float **Samples;
        int Rows,Cols;
        Rows=Channels;
        Cols = BufferLength;
        Samples = new float*[Rows];
        for (int i=0;i<Rows;i++)
                Samples[i] = new float[Cols];

        return(Samples);
}
//------------------------------------------------------------------
void FreeSampleBuffer(float **Samples,int Channels)
{
        int Rows;
        Rows = Channels;
        for (int i = 0; i < Rows;  i++)
                delete[] Samples[i];
        delete[] Samples;
}
//------------------------------------------------------------------
float ** AllocDecodeArray(int NoOfSpeakers,bool Order)
{
        float **Gains;
        int Rows,Cols;
        Order?Rows=9:Rows=4;
        Cols = NoOfSpeakers;
        Gains = new float*[Rows];
        for (int i=0;i<Rows;i++)
                Gains[i] = new float[Cols];

        return (Gains);
}
//------------------------------------------------------------------
void FreeDecodeArray(float **Gains,bool Order)
{
        int Rows;
        Order?Rows=9:Rows=4;
        for (int i = 0; i < Rows;  i++)
                delete[] Gains[i];
        delete[] Gains;
}
//------------------------------------------------------------------
void DecoderCalc(float *Azim,float *Elev,int NoOfSpeakers,bool Order,
      float WGain, float **Gains)
{
        float SinA,CosA,SinE,CosE,Sin2E,Sin2A,Cos2A;
        if(Order)
        {
                //Create 2 dimensional coefs array
                for(int i=0;i<NoOfSpeakers;i++)
                {
                        SinA=sin(Azim[i]);
                        CosA=cos(Azim[i]);
                        SinE=sin(Elev[i]);
                        CosE=cos(Elev[i]);
                        Sin2E=sin(2*Elev[i]);
                        Sin2A=sin(2*Azim[i]);
                        Cos2A=cos(2*Azim[i]);

                        Gains[0][i] = 0.5*(WGain);
                        Gains[1][i] = 0.5*(CosA * CosE);
                        Gains[2][i] = 0.5*(SinA * CosE);
                        Gains[3][i] = 0.5*(SinE);
                        Gains[4][i] = 0.5*(1.5f*SinE*SinE-0.5f);
                        Gains[5][i] = 0.5*(CosA*Sin2E);
                        Gains[6][i] = 0.5*(SinA*Sin2E);
                        Gains[7][i] = 0.5*(Cos2A*CosE*CosE);
                        Gains[8][i] = 0.5*(Sin2A*CosE*CosE);
                }
        }

        else
        {
                for(int i=0;i<NoOfSpeakers;i++)
                {
                        SinA=sin(Azim[i]);
                        CosA=cos(Azim[i]);
                        SinE=sin(Elev[i]);
                        CosE=cos(Elev[i]);

                        Gains[0][i] = 0.5*(WGain);
                        Gains[1][i] = 0.5*(CosA * CosE);
                        Gains[2][i] = 0.5*(SinA * CosE);
                        Gains[3][i] = 0.5*(SinE);
                }
        }
}
//-----------------------------------------------------------------
void B2Speakers(float **SGains,AmbiBuffer *Ambi, float **Samples,int
NoOfChannels,
      int NoOfSpeakers,bool Order)
{
        for(int i=0;i<Ambi->Length;i++)
        {
                for(int j=0;j<NoOfSpeakers && j<NoOfChannels;j++)
                {
                        if(Order)
                        {
                                Samples[j][i]=Ambi->W[i]*SGains[0][j]
                                +Ambi->X[i]*SGains[1][j]
                                +Ambi->Y[i]*SGains[2][j]
                                +Ambi->Z[i]*SGains[3][j]
                                +Ambi->R[i]*SGains[4][j]
                                +Ambi->S[i]*SGains[5][j]
                                +Ambi->T[i]*SGains[6][j]
                                +Ambi->U[i]*SGains[7][j]
                                +Ambi->V[i]*SGains[8][j];
                        }
                        else
                        {
                                Samples[j][i]=Ambi->W[i]*SGains[0][j]
                                +Ambi->X[i]*SGains[1][j]
                                +Ambi->Y[i]*SGains[2][j]
                                +Ambi->Z[i]*SGains[3][j];
                        }
                }
        }
}
#endif
```