Movements in Binaural Space: Issues in HRTF Interpolation and Reverberation, with applications to Computer Music

Volume 2 of 2

BRIAN CARTY

PhD Dissertation

NUI Maynooth

Music Department

August 2010

Head of Department: Professor Fiona Palmer

Supervisor: Dr Victor Lazzarini

Table of Contents

New Csound Opcodes for Binaural Processing

Victor Lazzarini and Brian Carty
Sound and Digital Music Technology Group,
National University of Ireland, Maynooth,
Co. Kildare,
Ireland
victor.lazzarini, brian.m.carty@nuim.ie

Abstract
Although solutions to the challenge of binaural artificial recreation of audio spatialisation exist in the Computer Music domain, a review of the area suggests that a comprehensive, generic, accurate and efficient toolset is required. A number of Csound opcodes, using a Head Related Transfer Function based approach, are presented to satisfy this necessity. The process is a complex one, with perhaps the most significant difficulty being phase interpolation. Novel approaches (specifically methods using phase truncation and functionally derived itd respectively), as well as a method based on established digital signal processing methods (minimum phase plus delay) are implemented.

Keywords
HRTF, binaural, Csound

1. Sound Localisation

Binaural hearing is the term given to listening with two ears rather than one, and is the main factor involved in sound localisation. The fact that the brain receives an independent signal from each ear allows conclusions to be drawn based on a comparison of the characteristics of each signal.

One such binaural indication of a sound's spatial characteristics is interaural *time* difference (itd): the name given to the time it takes a sound to reach one ear after it has first reached the other. Despite the relatively tiny nature of these time disparities, they can provide very accurate localisation cues.

Interaural *intensity* difference (iid) uses varying respective intensities of a signal at each ear to locate source sounds. Interaural intensity difference is based principally on the head (and to a lesser extent the torso) acting as a barrier to sound. It is generally accepted that interaural time and intensity differences work together to provide a well-defined spatial image, with itd working best for low frequencies and iid for high.

Monaural information (independent information from one ear) also plays an important role in sound localisation. The pinna and concha both have a non-linear frequency response over the audible spectrum, altering incoming sounds. These alterations vary with sound location.

2. Head Related Transfer Functions

Head Related Transfer Functions (HRTFs) are essentially functions that describe how a sound from a specific location is altered from source to tympanic membrane. For any particular source sound, a pair of transfer functions exist (for the left and right ear) for any location relative to the listener. These functions will encompass all localisation cues mentioned above. More generally, Head Related Transfer Functions can be defined as the impulse responses of the left and right ears to sound at a particular location.

The process of simulating an auditory location using HRTFs as a frequency domain operation can be summarised thus:

- Record the response of the left and right ear to an impulse (a frequency rich source) to learn how each ear will treat sound at all frequencies for the desired point in space.
- Analyse the frequency content of the sound you wish to spatialise.
- Impose the HRTF for the left and right ear on the sound (boost or attenuate and delay the frequencies contained in the input in

accordance with how the ear treats the appropriate frequencies), using the process of convolution.

As the physiology of everyone's ears is different, HRTFs vary considerably from subject to subject. Ideally, listeners should use binaural systems customised to their own ears. However, certain consistencies can be observed and generalised/*non-individualised* HRTFs, recorded using a dummy head and torso model or a specific subject are frequently used to remove the necessity for measurements to be taken for each individual user. Results from [18] suggest that although non-individualised HRTFs are certainly a useful tool for binaural simulation, they can result in a distortion of the spectral characteristics used in front/back and elevation resolution when compared to listening in the free field.

It is also important to note that binaurally generated 2 channel (left ear and right ear) signals should be reproduced on headphones to avoid crosstalk and environmental and listener interactions with the source.

3. HRTF Interpolation

HRTF data sets are typically measured at discrete, equidistant points around a listener or dummy head, for example [3]. Therefore some form of interpolation is required for non measured points. The topic of HRTF interpolation is a multi faceted one, with many suggested and possible approaches, for example spatial frequency response surfaces (see [2]) and virtual loudspeaker multichannel approaches (see [15]).

The process of HRTF localisation outlined above describes the localisation of a source sound to one specific area of space. When other locations are required, however, the relevant HRTF data is needed. A fixed amount of points are typically recorded and stored. However, if a location is required that has not been measured, or if a sound is required to move smoothly from one location to another, some kind of averaging or interpolation must be done.

HRTF interpolation can be thought of as taking the two (or more for increased accuracy) nearest HRTF representations to a non-measured point in between, and deriving a new HRTF by averaging the known values with greater relative weighting(/s) on the nearer known point(/s).

Audio, or indeed any type of signals can be represented in several ways. Traditionally, audio is viewed, edited, processed and auditioned in the time domain. However, the frequency domain can provide more useful insights into the properties of the signal. Individual sinusoidal components of a signal, in this case a head related transfer function, can be examined, and their magnitude and phase can be extracted in the frequency domain. The former quantifies the relative strength of the signal at each frequency in the analysis, the latter, the phase/starting point of the component.

Frequency domain interpolation can generally give more accurate results than time domain techniques (see [4] and [14]). However, interpolating in the frequency domain poses the problem of phase interpolation. Phase values are closely related to itd, so are important in the localisation process. The linear interpolation of phase values is flawed. Phase, unlike magnitude, is a periodic quantity, measured in fractions of a full cycle. Uncertainty arises when trying to interpolate phases, as a phase value can be +/- any amount of full cycles. For example, in Figure 1, the first and second points have phase values of 10 and 50 degrees respectively. However, as phase is periodic, these may be 10 or 50 degrees plus any number of full cycles. Therefore interpolated phase may be 30 or 210 degrees, depending on whether the 50 degree phase represents 50 degrees or 410 degrees (50 degrees plus one cycle) respectively, for example.

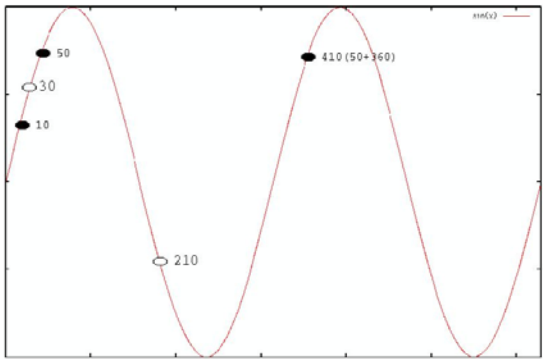


Figure 1: Phase Interpolation

4. Minimum Phase

Oppenheim and Schafer observe that any rational system function can be broken into a minimum phase and an all pass system [16]. An all pass system can de defined as one which has a magnitude response that is absolutely constant with respect to frequency [17]. Therefore, the magnitude of the minimum phase all pass decomposition is represented solely by the minimum phase system and the phase is reconstituted by both the allpass and minimum phase representations.

The system in question can thus be defined as:

$$H(z) = Hmin(z)Hap(z), \qquad (4.1)$$

where Hmin(z) is a minimum phase system, and Hap(z) is an all pass system.

Typically, magnitude and phase spectra are not related. A unique and, in this case, extremely useful property of minimum phase systems, however, is that phase values for each component frequency can be derived from the corresponding magnitude values, see [16] for details.

5. Minimum Phase and HRTFs

The significance of phase information and the auditory system's limitations in responding to changes in phase information has been investigated in depth, for example, see [9]. In [11], it is observed that the auditory system approaches minimum phase. The authors decomposed their measured transfer functions into minimum phase and allpass functions in order to obtain a clear representation of phase without the above mentioned 2 pi ambiguity. While doing this, they realised that the minimum phase function appeared to contain almost all the detail of the phase spectrum and that the allpass phase approached linearity for the free field to ear canal function. The paper goes on to assert that the allpass component of the full HRTF (including the ear canal response, as defined by Begault[1]) exhibits a 'nearly linear' phase response up to 10 kHz. Therefore, the allpass component can be implemented as a simple time delay. This time delay can be realised using a time domain, frequency independent delay line, quite a simple and efficient process to implement. This observation of approximate minimum phase has become a key factor in binaural HRTF based processing, and has been used in several studies of HRTFs, many of which suggest HRTF models based on minimum phase plus delay decomposition, such as principal component analysis [6] and infinite impulse response models [8].

The minimum phase and (assumed linear) all pass decomposition allows a pair of HRTFs (for the left and right ears) to be broken down into 3 parts: a minimum phase representation of each empirical HRTF pair (left and right ear), and an interaural delay. The overall magnitude will be represented by that of the minimum phase filter; the overall phase will be the minimum phase phase spectrum (derivable from the magnitude spectrum) plus a frequency independent, linear delay. Thus phase interpolation is no longer a problem. Figure 2 shows an empirical impulse and its minimum phase representation in the time domain.

[1] Begault defines the hrtf as 'the spectral filtering of a sound source before it reaches the ear drum that is caused primarily by the outer ear' in [1]. However, it is undesirable to use hrtfs that contain the auditory canal response of the dummy head in artificial localisation applications, as the listener, using headphones that transmit audio from the entrance of the ear canal, is then essentially listening through 2 auditory canals, that of the dummy head and their own. This is avoided in the MIT dataset used here through diffuse field equalization.

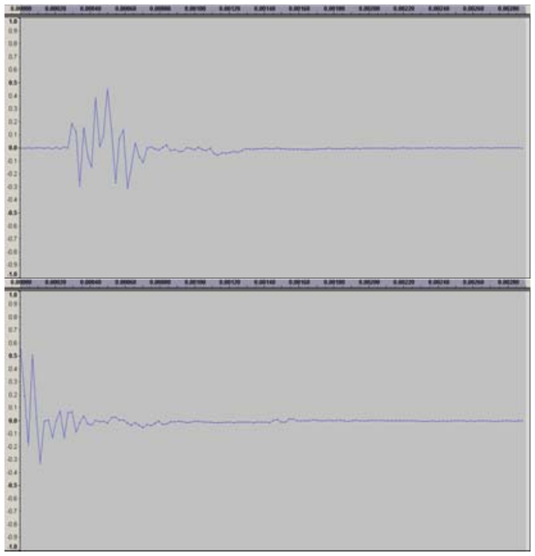


Figure 2: An Empirical HRTF (source at 0 degrees) and its Minimum Phase Representation

The process of HRTF interpolation consequently involves analysing each HRTF pair to find the relevant interaural delay and reducing them to minimum phase representations. The minimum phase magnitude values and extracted delay can then be linearly interpolated. Interpolated minimum phase phase spectra can be derived from interpolated magnitude spectra.

The description of HRTFs as minimum phase filters plus delays above is validated theoretically by work on decomposition of impulses in [11]. However, perhaps a more pertinent validity test from the point of view of a developer of artificial spatialisation tools involves psychophysical testing of a subject group. Kulkarni's seminal paper examining the sensitivity of human subjects to HRTF phase spectra [9] reports high coherence values between empirical and minimum phase plus delay data sets. However, coherence values were systematically worse at lower elevations and extremes of the horizontal plane. It is suggested that this is due to the shadowing effect of the head and interactions with the torso making the allpass delay non linear, a phenomenon discussed in [7]. This is supported by better performance at higher elevations, where there is less obstruction in the path to the contralateral (further from the source) ear. Phase error results enforce this assumption.

Psychophysical results point to a low frequency cue present at extremes of the horizontal plane, aiding the subject in distinguishing between min phase plus delay and empirical impulses. This perhaps suggests that modelling itd as a linear delay is not adequate; however, overall the study concludes that minimum phase plus delay models are sufficient for most locations, and that the finer structures of phase are not overly important, as long as the overall delay is approximated in accordance with that of the empirical.

Practicalities in implementation of a minimum phase based spatialisation system also need to be considered. The tiny delays extracted from the HRTF data set will often fall in between the sample by sample values used in a delay line. This is a consequence of digital delay lines and sampling. To remove abrupt changes in delay for moving sources, interpolated variable delay lines can be employed. As observed in [2], interpolated delay lines attenuate high frequencies, and are therefore not ideal. However, informal listening tests performed both in [2] and by the authors suggest that these artefacts are not significant.

Alternatives to the minimum phase approach are suggested that do not assume the approximations involved in modelling the HRTF as minimum phase plus delay. This essentially involves engaging more directly in the phase ambiguity problem. The minimum phase method employs complex digital signal processing of the HRTF data, and is quite computationally expensive. The approaches outlined below are intended to give spatially accurate and efficient processing without the necessity to perform complex analysis, compression or transformation of the data.

6. Current Csound Solution to Binaural HRTF Based Processing

The HRTFer opcode uses the MIT data set (see [3]) to spatialise the desired source sound.

aleft, aright **HRTFer** ainput, kangle, kelevation, “HRTFcompact”

HRTFer provides accurate spatialisation for static locations which correspond exactly to HRTF measured points. However, if a static point is required that has not been measured, the system simply chooses the nearest measured point. Considering the density of this data set these inaccuracies may be tolerable for certain situations. This approach causes more significant errors when a specific trajectory is desired for a source. A dynamic, rather than static source will skip from one nearest measured point to the next along a user defined trajectory. This staggered movement causes irregularities in the output, manifesting themselves as discontinuities, an undesirable result. The original authors of the opcode suggested a fade out of the old convolution result and a fade in of the new to reduce this noise. However, these crossfades have been disabled, as they cause dropouts in the output, leading to worse irregularities, which are assumed to be caused by an error in the source code. In tests performed by the authors, these crossfades, when implemented, reduce the irregularities to a degree depending on the frequency content of the source.

Another consequence of abruptly changing these complex filters (HRTFs) as a source travels along a defined trajectory is the sudden perceptual change in the output, which can be detrimental even in frequency rich sources (which may mask discontinuities to a certain extent). For example, in a trajectory going from 50 degrees above the listener to directly in front, the source will appear to jump downwards every 10 degrees, as this is the measurement increment. Clearly, this opcode could benefit from the addition of interpolation between measured points.

7. Novel Solutions to HRTF Binaural Processing

Two novel approaches to HRTF binaural processing are presented below.

7.1 Phase Truncation, Magnitude Interpolation

The first suggested approach can be summarised as magnitude interpolation and phase truncation. It can be simply defined as using interpolated spectral magnitude values, and the nearest known phase values to derive the impulse for each block of audio processed. This approach, as well as providing an adequate solution to HRTF interpolation for sources to be placed at non measured points, allows artefact free, user defined source trajectories.

The movement in the program is achieved by updating user defined angle and elevation values, according to where the source is moving from and to, every processing block.

The interpolation algorithm works by storing the four nearest HRTF values to the desired location, left and right below and above. Linear interpolation of the magnitude values is performed. This magnitude interpolation method derives an accurate intermediate/transitional fir filter, essentially boosting/attenuating spectral bands to a level that is proportionate to and sympathetic with the nearest measured points. For source trajectories adhering to minimum audible movement angle constraints (see [13]), noise introduced by filter magnitude values changing as the source moves is inaudible/tolerable.

The nearest measured phase value is used for intermediate filters. As the difference in measured points is often quite minimal, although always significant, it is proposed that choosing the phase of the nearest measured point will not have a significant adverse effect on the final spatialisation quality. As discussed above, studies have shown that phase does indeed play an important role in localisation, but as long as an accurate overall itd is maintained, users frequently cannot distinguish errors.

However, as the source trajectory moves closer to a different measured point, these phase values need to be updated. An abrupt switch of phase values will cause an audible inconsistency in the output. Although the severity of this inconsistency depends on the source material to be spatialised, a method to minimise it is desirable. The crossfade method suggested by the csound HRTFer opcode is considered and developed. Fades are performed when new, nearer *phase* values are available. This approach, coupled with magnitude interpolation, gives much smoother movement. The frequency content of the source defines the audibility of the switch between phase values. If a narrowband source is to be spatialised (i.e. a source with energy focused in a small number of narrow frequency bands), the switch will be quite obvious. However, more noisy, frequency rich sources may be able to mask the inconsistency caused by the new phase values to an extent related to the complexity of the source. It is with this in mind that a user definable, source specific solution is proposed. The user may choose to perform crossfades over a number of processing buffers. One buffer may be enough to mask unwanted inconsistencies for certain sources, whereas others may require up to 16 buffers to mask all artefacts.

The process of crossfading thus involves processing the old HRTF data with the new input data and fading out, while processing the new HRTF data with the input and fading in.

Another point to consider is that for very fast trajectories, the nearest measured phase values may be changing quite swiftly. However, considering the minimum audible movement angle (see [13]), and that only audible trajectory changes are desirable, the system is adequate. A related criticism, however, is that occasionally a path may be required which causes the angle and elevation index to change over the same crossfade. If the path involves a three dimensional trajectory, phase value updates may not be uniform. Users will be warned in this scenario, and can simply reduce the crossfade size. As mentioned previously, the spectral content of the input sound may mask discontinuities, so shorter crossfades will suffice in certain situations. Figure 3 illustrates the magnitude interpolation, phase truncation process for a moving source.

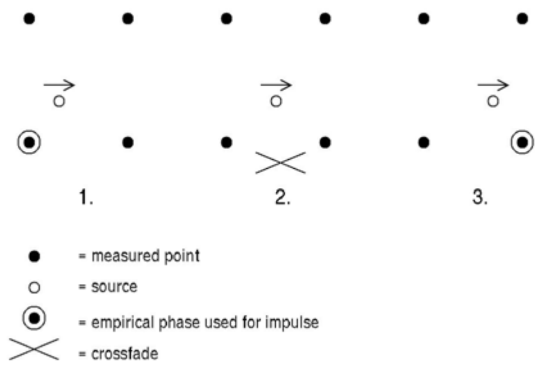


Figure 3: Magnitude Interpolation, Phase Truncation

A minimum phase based implementation is also developed using the model discussed in [9]. Essentially, magnitudes are interpolated as above, and phase is derived from these interpolated magnitudes. A linear allpass delay is inserted using a variable delay line. Minimum phase or phase truncation based processing can be chosen by the user as an optional parameter of the opcode developed.

7.2 Functional Phase Model

Another approach, essentially a hybrid of an empirical and modelled transfer function is presently suggested. As discussed above, spectral magnitude measurement, representation and interpolation is straightforward and easily realisable. Therefore empirical magnitude values are employed here. The difficulty of phase representation and interpolation is approached from a functional modelling point of view.

The main task in functional phase derivation is to model correctly the interaural phase difference, therefore deriving the correct interaural time difference. A basic, yet practical model for the head is to assume it approximates a sphere. The degree to which this phase simplicity will distort the spatial image is closely related to the discussion above on sensitivity to phase differences, which concluded that low frequency itd across frequency is the predominant phase cue (see [9]).

Mathematically, the itd for a particular source location, assuming a spherical head can be defined thus:

$$\frac{r(\vartheta + \sin\vartheta)}{c}\cos\phi\,, \qquad (7.2.1)$$

where r is the head (/sphere) radius, c is the speed of sound, ϑ is the angle and ϕ the elevation of the source. This formula is described as the Extended Woodworth/Schlosberg Formula in [12].

Successful use of this basic Woodworth model for HRTF phase modelling and a magnitude interpolation algorithm is reported in [19]. Some development and improvement is suggested here. As concluded in [9], low frequency consistency of empirical and employed itd is crucial for accurate modelling. Also, it is agreed in both [9] and [7] that higher frequency itd is not as significant, and specified in [7] that a Woodworth model can account for steady state high frequency itds. Also, physiologically, interaural phase difference based localisation breaks down above approximately 1500 Hz (see [13]). Therefore a low frequency, frequency dependent scaling factor is introduced as a more complete solution, requiring minimal extra processing. Essentially, itd is extracted from the empirical HRTFs for each low frequency band of interest. These values are then used as frequency dependent scaling factors in the synthesis of the phase spectrum for the desired HRTFs.

This model provides an accurate average low frequency itd for this particular dataset, and a steady Woodworth based itd for higher frequencies.

The values derived from this Extended Woodworth/Schlosberg non linearly low frequency scaled (/functional) model are then used as phase values. Phase values are calculated per frequency bin, with values of minus and plus half the itd for the ear nearer and further from the source respectively. Practically, negative phase values simply wrap around to the end of the impulse. It therefore appears that the nearer ear impulse happens after the further ear, which is an unnatural result. For this reason, the impulse is shifted in time, by half the size of the buffer. The result is a time and phase accurate filter.

Phase interpolation for dynamic sources has been discussed and a solution presented in the form of phase truncation. However, with the Woodworth functional approach, a new phase can be derived for any location, and can be used and updated for each processing block of a dynamic source. This is an initially exciting prospect; however implementation illustrates undesirable noise, caused by phase updates, and phase not 'matching' magnitudes, as it does in minimum phase implementations.

The short time Fourier transform (stft, see [14]) is employed to avoid the irregularities introduced by changing modelled phase per processing block.

8. Csound Implementations

Three plugin opcodes are designed using the guidelines in [10], one allowing phase truncation or minimum phase binaural processing, and two based on the functional model. The reason for two opcodes based on the functional model is due to the efficiency with which a static source can be processed in comparison to the necessity of stft processing for a dynamic source. The phase truncation/minimum phase model allows the user to choose between minimum phase and phase truncation processing, the latter also allowing user defined crossfade sizes. Functional models allow choice of spherical head radius for itd calculation, and stft overlap for dynamic trajectories. All models allow sampling rates of 44.1, 48 and 96 kilohertz. Data files containing the HRTF data at the appropriate sampling rate, as well as minimum phase delay data are also required.

Despite the addition of magnitude interpolation, and algorithms for appropriate phase representation, the new, optimised opcodes perform favourably in comparison to the HRTFer opcode. For example, the phase truncation process takes an approximate average of .11 seconds of CPU time to spatialise 2 seconds of audio on a 0 to 90 degree trajectory. HRTFer takes an approximate average of .16 CPU seconds to perform the same operation. This figure is comparable with minimum phase processing time for the same trajectory. To place this source statically with the functional model takes just .07 CPU seconds, but to perform the trajectory above with the functional model takes .17 seconds due to the addition of the stft processing. Note: default opcode values were used for the above approximate average csound CPU time tests (crossfade over 8 buffers for phase truncation, head radius of 9cm for the functional models, overlap of 4 for the stft and sampling rate of 44.1 kHz for all).

9. Conclusion and Discussion of Methods Employed

As discussed in [9], HRTF phase data does not require exact accuracy. More specifically, maintaining low frequency interaural time delays appears to provide accurate phase data. The phase truncation method described maintains nearest measured phase data, thus meeting this criterion. The goal of the method: to use the data directly, is also achieved. A generic, user definable model is presented to allow for compromise between complex trajectories, narrow band sources and changing phase noise removal through variable length cross fades.

Minimum phase requires data preparation and knowledge of complex digital signal processing. Furthermore, casual listening tests show there is often an audible discrepancy between minimum phase and empirical data convolution for musical and test sources, although localisation is good for both. Phase truncation output appears to give a result more consistent with the empirical dataset as a whole.

Functional models introduced above assume the head is a sphere and will be accurate to this degree, but adding non linear low frequency scaling factors will reintroduce some of the finer phase detail involved in the non uniformly spherical shape of the head, the pinnae and the torso.

The functional model implementation provides a more mathematical approach, with the addition of the specifics of the data set used, implemented in an efficient and psychoacoustically consistent way. The importance of low frequency phase information is preserved and applied to an efficient, simple model for phase. This provides a speedy solution for static sources; however dynamic sources require stft processing.

The binaural processing capabilities of csound have thus been updated and improved, using existing and novel approaches. Smooth, artefact free dynamic and static binaural processing is now realisable using the various techniques described above.

Acknowledgements

This work is supported by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan and NUI Maynooth.

References

[1] Durand Begault. *3-D Sound for Virtual Reality and Multimedia*. AP Professional, London, 1994.

[2] Cheng and Wakefield. Moving Sound Source Synthesis for Binaural Electroacoustic Music Using Interpolated Head-Related Transfer Functions (HRTFs). *Computer Music Journal*, 25:4: 57–80, 2001.

[3] Gardner and Martin. HRTF Measurements of a KEMAR Dummy Head Microphone (http://sound.media.mit.edu/KEMAR.html, accessed July 2007) MIT, 1994.

[4] Hartung, Braaschand Sterbing. Comparison of Different Methods for the Interpolation of Head Related Transfer Functions. *AES 16th International Conference: Spatial Sound Reproduction*, 319-329, 1999.

[5] Jot, Larcher and Warusfel. Digital Signal Processing Issues in the Context of Binaural and Transaural Stereophony. *AES 98th Convention*, 1995.

[6] Kistler and Wightman. A model of head-related transfer functions based on principal components analysis and minimum-phase reconstruction. *Journal of the Acoustical Society of America* Volume 91(3): 1637-1647, 1992.

[7] Kuhn. Model for the interaural time difference in the azimuthal plane. *Journal of the Acoustical Society of America* Volume 62(1): 157-167, 1977.

[8] Kulkarni and Colburn. Infinite-impulse-response models of the head-related transfer function. *Journal of the Acoustical Society of America* Volume 115(4): 1714-1728, 2004.

[9] Kulkarni, Isabelle and Colburn. Sensitivity of Human Subjects to Head-Related Transfer-Function Phase Spectra. *Journal of the Acoustical Society of America.* Volume 105(5): 2821-2840, 1999.

[10]Lazzarini. Extensions to the Csound Language. *Linux Audio Conference*, 13-19, 2005.

[11]Mehrgardt and Mellert. Transformation Characteristics of the external human ear. *Journal of the Acoustical Society of America.* Volume 61(6), 1977.

[12]Minnaar, Plogsties, Olesen, Christensen and Moller. The Interaural Time Difference in Binaural Synthesis. *AES 108th Convention*, 2000.

[13]Moore. *An Introduction to the Psychology of Hearing* Elsevier Academic Press, London, 1977; 5th edn, 2004.

[14]Moore: *Elements of Computer Music*. Prentice-Hall, New Jersey, 1990.

[15]Noisternig, Musil, Sontacchi and Holdrich. 3D Binaural Sound Reproduction using a Virtual Ambisonic Approach. *IEEE Symposium on Virtual Environments*, 174-178, 2003.

[16]Oppenheim and Schafer: *Discrete-Time Signal Processing*. Prentice Hall , New Jersey, 1989; 2nd edn, 1999.

[17]Steiglitz. *A DSP Primer*. Addison-Wesley, Clifornia, 1996.

[18]Wenzel, Arruda, Kistler, and Wightman. Localization using non-individualized head related transfer functions. *Journal of the Acoustical Society of America* Volume 94(1): 111-123, 1993.

[19]Zotkin, Duraiswami and Davis. Rendering Localized Spatial Audio in a Virtual Auditory Space, *IEEE Transactions on Multimedia*, Volume 6(4), 553-564, 2004.

Audio Engineering Society

Convention Paper

Presented at the 126th Convention
2009 May 7–10 Munich, Germany

Frequency-domain Interpolation of Empirical HRTF Data

Brian Carty[1] and Victor Lazzarini[2]

[1 and 2] Sound and Digital Music Technology Group, National University of Ireland, Maynooth, Co. Kildare, Ireland
[1] brian.m.carty@nuim.ie
[2] victor.lazzarini@nuim.ie

ABSTRACT

This paper discusses Head Related Transfer Function (HRTF) based artificial spatialisation of audio. Two alternatives to the minimum phase method of HRTF interpolation are suggested, offering novel approaches to the challenge of phase interpolation. A phase truncation, magnitude interpolation technique aims to avoid complex preparation, manipulation or transformation of empirical HRTF data, and any inaccuracies that may be introduced by these operations. A second technique adds low frequency non-linear frequency scaling to a functionally based phase model. This approach aims to provide a low frequency spectrum more closely aligned to the empirical HRTF data. Test results indicate favorable performance of the new techniques.

1. INTRODUCTION

1.1 Sound Localisation

Human sound localisation capabilities are based on a number of cues. Of greatest significance is the binaural nature of the hearing system. As the term suggests, listening with two ears allows comparisons to be made between two independent signals. These comparisons are the primary cues used in locating a sound source in a particular spatial location. The difference in time of arrival and intensity of the signals at the ear nearer and farther from the source respectively are the main binaural cues used in sound source localisation. Interaural time difference (ITD) uses the phase difference between signals to locate the source sound. ITD works best at lower frequencies, doe to possible ambiguities in higher frequency phase information. Interaural intensity difference (IID), conversely, works better at higher frequencies, as lower frequencies tend to diffract around the head, reducing IID in the low frequency region, making it a less reliable cue.

Spectral information also plays an important role. The various non-linearities in the path from source to inner ear filter a sound. This filtering varies with source location. Of particular significance here is the complex shape of the pinna. For example, sound from behind a listener will appear duller, as higher frequencies are filtered out by the back of the pinna. Spectral cues facilitate localisation in the median plane, where interaural differences are minimized.

1.2 Head Related Transfer Functions

Head Related Transfer Functions (HRTFs) essentially describe how a source sound is altered from source to inner ear, and encapsulate the localisation cues mentioned above. For any particular location relative to a listener, a HRTF pair exists (for the left and right ear). This HRTF data can be obtained by recording the impulse response of a listener, or dummy head to source sounds at various locations. The dataset used in this study was recorded at MIT [1] using a dummy head.

An immediate application of such a data set is the artificial spatialisation of audio. A non spatialised, mono sound source can be convolved with the HRTF pair for a specific location relative to the listener to artificially locate it to that location. There are, however, a number of caveats to this initially promising process. HRTFs are individual specific, as the physiology of each listener will be different, so inaccuracies will be introduced by using a generic dataset. Finer detail of spectral cues will be thus degraded. However, non-individualized datasets have been reported as a useful tool in artificial spatialisation [2], and are frequently used as the recording of a dataset is a timely and difficult process. The two channel signal derived from such a convolution operation should ideally be played back on headphones, as a loudspeaker reproduction will reintroduce environmental processing, cross talk, and the pinna of the listener will filter the sound again in error, as the pinna is already considered in the HRTF. The challenge of locating source sounds to non measured points or implementing dynamic source trajectories must also be considered. A data interpolation method is needed, which this paper addresses.

This artificial spatialisation of mono audio sounds provides the initial motivation for this study. The paper will therefore discuss the traditional approach to the challenge of implementing moving sources, and suggest, justify and verify the successful implementation and performance of two novel alternatives. The paper will summarize the novel algorithms, and focus on the results of subjective and objective algorithm testing.

2. HRTF INTERPOLATION

Datasets, like [1], will contain a fixed number of HRTFs, measured at static points around a listener. In an artificial spatialisation application, it may be desirable to spatialise a source sound in between measured points, at a location where data is not available. This task becomes more pertinent if a source sound is required to move along a trajectory. Clearly, some form of interpolation is required. This is not a simple task; many approaches have been suggested, including Spatial Frequency Response Surfaces [3], virtual loudspeaker multi-channel approaches [4], numerical methods [5] and infinite impulse response (IIR) representations of HRTF data [6].

In deriving data for non-measured locations, empirical data is used. A typical approach involves combining empirical measurements with relative weightings on those nearer the desired location. Performing this process in the frequency domain is desirable, as it is more accurate [5], [7]. Frequency-domain data involves the magnitude and phase of each spectral component of a signal. To interpolate two HRTFs, therefore, the magnitudes and phases of the HRTFs should be combined in order to derive an intermediate HRTF. Linear interpolation of magnitude data provides an intermediate point that is consistent with empirical data (although perhaps not independently accurate). However, the same cannot be said of linear interpolation of phase data (as mentioned above, ITD is understood as phase data). Phase, by its nature, is a periodic property. Therefore ambiguities arise in the task of phase interpolation. If the phase of a particular spectral component of a HRTF is 10 degrees, and the phase of the same spectral component of an adjacent HRTF is 50 degrees, linear interpolation to derive an interpolated phase for a HRTF half way between these points implies a phase of 30 degrees. The periodic nature of phase, however, introduces ambiguities. The phase of the 50 degree point, for example, may actually represent a phase of 50 degrees plus any number of full cycles of 360 degrees, or 2π. Figure 1, below, illustrates this scenario. A sine wave is illustrated, with approximate phases of 10 degrees and 50 degrees marked. The 30 degree interpolated point, half way between empirical points, may be in error, as the 50 degree phase may represent, to take just one example, 410 degrees (50 + 360).

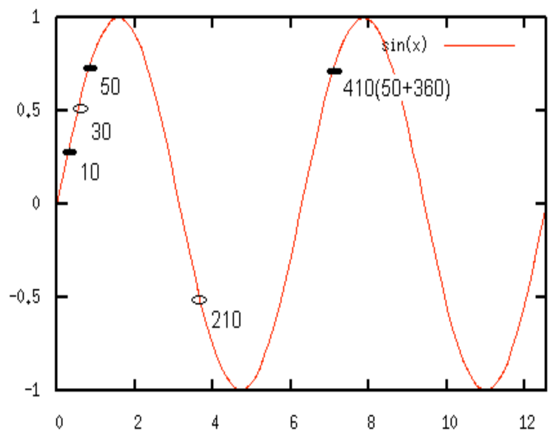


Figure 1: Phase Interpolation

The minimum phase allpass decomposition of the HRTF is the traditionally approach taken to overcome this difficulty.

2.1 Minimum Phase

Digital signal processing can be used to break any rational system function into a minimum phase and all pass component [8]. All pass systems have a flat magnitude response. Consequently, the magnitude of the original function in this alternative representation is contained in the minimum phase system. The phase is represented by both the minimum phase and the all pass systems. The breakdown of the system function *H(z)* is illustrated in equation 1, below.

$$H(z) = H_{\text{min}}(z)H_{\text{ap}}(z) \tag{1}$$

where $H_{\text{min}}(z)$ is a minimum phase system and $H_{\text{ap}}(z)$ is an allpass system.

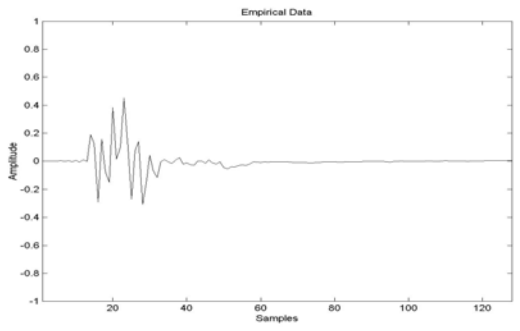
2.2 Minimum Phase Based HRTF Interpolation

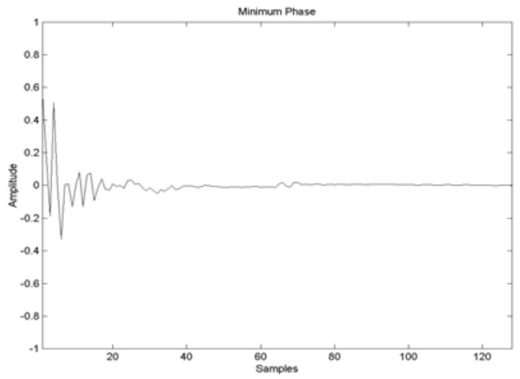
The significance of the minimum phase all pass decomposition with regard to artificial spatialisation using HRTFs will now be discussed. Magnitude and phase spectra of minimum phase systems are related. This unique property means that the phase value for a given component frequency of a minimum phase system can be derived from its corresponding magnitude value. As phase interpolation is the main challenge involved in HRTF interpolation, this property is of immediate interest. If a HRTF dataset is decomposed into minimum phase and all pass components, the minimum phase components can be interpolated to derive the minimum phase portion of non measured points. Magnitude interpolation can be performed, and phase values derived form these interpolated magnitude values. This provides a satisfactory result for the minimum phase portion of the decomposition. However, the all pass component has not yet been considered. Crucially, the all pass component has been shown to be 'nearly linear' up to 10 kHz [9]. Therefore, the HRTF can be understood as a minimum phase system. The all pass function of the minimum phase all pass decomposition can thus be approximated using a linear delay. This is extremely relevant in an artificial HRTF based artificial spatialisation system. As mentioned, the magnitude spectrum of the minimum phase representation of the HRTF can be linearly interpolated. The all pass section, assuming linearity, can be represented by a frequency independent, time domain delay line, which can also be simply interpolated for non measured points.

The HRTF pair for a particular location can thus be decomposed into 2 minimum phase functions (left and right) and an interaural delay. The process of HRTF interpolation can then be performed by interpolating the magnitudes of the minimum phase HRTFs and the interaural delay. The magnitude of the empirical HRTF is represented solely by the minimum phase HRTF. Thus an interpolated magnitude response can be achieved by interpolating the magnitudes of the minimum phase HRTFs. As mentioned above, the phase values of the resulting minimum phase interpolated HRTF can be derived from the magnitude values. The assumed linear all pass component of the decomposition can be treated as a pure delay, and thus interpolated appropriately. This minimum phase approximation has been used in several approaches to HRTF representation, such as principal component analysis [10] and IIR models [11].

In [12], Kulkarni et al. objectively investigate the linearity of the all pass function in the decomposition, reporting high coherence values between empirical and minimum phase plus delay HRTFs. Coherence values were, however, consistently worse at lower elevations and extremes of the horizontal plane. The shadowing effect of the head and interactions of sound with the torso (also discussed in [13]) are suggested as reasons for this. At higher elevations, there is less obstruction to the contralateral (further from source) ear. Perhaps of more relevance to an artificial spatialisation application are the psychophysical tests performed in the same study. Again, extremes of the horizontal plane are highlighted as areas where the minimum phase plus delay decomposition is not reliable. The study does, however, conclude that minimum phase plus delay is adequate for most locations. More specifically, the study illustrates that the delay used in the decomposition should agree closely with low frequency empirical ITD, and that the finer structures of phase are not excessively important in localisation.

Implementation of an artificial spatialisation system using the minimum phase assumption also involves using filters with the lowest possible number of coefficients, as the energy in the impulse is focused at its start. It is also worth noting that minimum phase based artificial spatialisation requires the use of interpolated delay lines, which may colour the output sound. Figure 2 shows an empirical Head Related Impulse Response (HRIR: a time domain HRTF), its minimum phase representation and their common magnitude response.





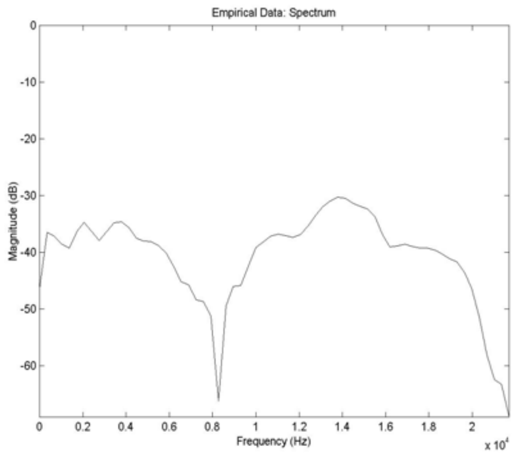


Figure 2: An empirical and minimum phase HRTF for a source directly in front of a listener and the magnitude response of the empirical data, which is the same as that of the minimum phase response.

3. ALTERNATIVE APPROACHES TO HRTF INTERPOLATION IN THE FREQUENCY-DOMAIN

3.1 Motivation

Initial motivation for this study was the development of robust, efficient algorithms for the artificial spatialisation of audio, and an update of HRTF based spatialisation available in open source computer music languages such as Csound. Implementation issues are discussed in [14].

Another main goal of this study is to offer alternatives to the minimum phase assumption. Minimum phase based processing involves complex preparation/online processing of any dataset used. Approaches aiming to use empirical data more directly, bearing in mind the findings of [12] regarding relative insensitivity to the finer detail of phase in localisation, are thus suggested. Although the minimum phase assumption provides good localisation, novel alternatives are presented that approach the problem of phase interpolation more simply and directly. The algorithms developed provide efficient, spatially accurate processing, while minimizing complex data preparation, approximation or compression.

3.2 Phase Truncation

The first proposed algorithm suggests a novel treatment of phase in interpolated HRTFs used for non measured/interpolated sources. Four point linear interpolation is applied to magnitude values. This results in a magnitude spectrum that is consistent with nearest empirical points. Values from the four nearest measured HRTFs are combined, with higher relative weighting given to values nearer the desired location. This method cannot account for local anomalies in the spectrum. However, the dataset used [1] is relatively dense, with measurements every 5 degrees on the horizontal plane, for example. Also, when minimum audible movement constraints [15] and other limitations of the audio spatialisation system are considered, this linear interpolation approach provides a good approximation. It is also employed with a minimum phase approach to phase in [16] and a functional phase model, which will be discussed below, in [17].

The algorithm was originally developed to provide smooth, spatially accurate dynamic source trajectories. This is achieved using frequency domain overlap-add convolution. The azimuth and elevation of the source is updated from a user defined trajectory every processing block. Filters of 128 samples are used.

The novel contribution of this method is the treatment of phase values. Bearing in mind the relative insensitivity to phase reported in [12], nearest empirical phase values are used. Figure 3 illustrates this method. The interpolation algorithm is illustrated for a source moving from left to right. At point 1, the source is nearest the bottom left empirical point. Therefore, magnitude interpolation will derive an intermediate magnitude spectrum with highest relative weighting on this point. The phase values from this HRTF will be combined with the interpolated magnitude values for the HRTF used in the spatialisation of the source. This process will be repeated for each processing block. At point 2, the source will be between the bottom left and bottom right points. This scenario is dealt with as follows: instead of switching abruptly between phase values, which may cause an audible discontinuity in the output, a brief cross fade is introduced. The input is processed with the old phase data and faded out. At the same time, the input processed with the new phase data is faded in. The spectral content of the input will determine the audibility of any discontinuity when phase is abruptly changed. A narrowband source will typically exhibit a more severe discontinuity than a noisy source. Therefore, the crossfade is parametric. Users can define its length. Crossfades as brief as 1 processing buffer of 128 samples may be sufficient for noisy sources, whereas longer fades may be desirable for more narrowband sources. A default value is offered in the Csound implementation of this algorithm, which also offers minimum phase processing [18]. As the source in figure 3 moves closer to the bottom right empirical impulse at point 3, its phase values are used, and magnitude interpolation is weighted more strongly to that point.

The algorithm, when tested subjectively, performs very well (see below), providing smooth source movement. The goal of using empirical data more directly is also achieved and implemented in an efficient and simple algorithm.

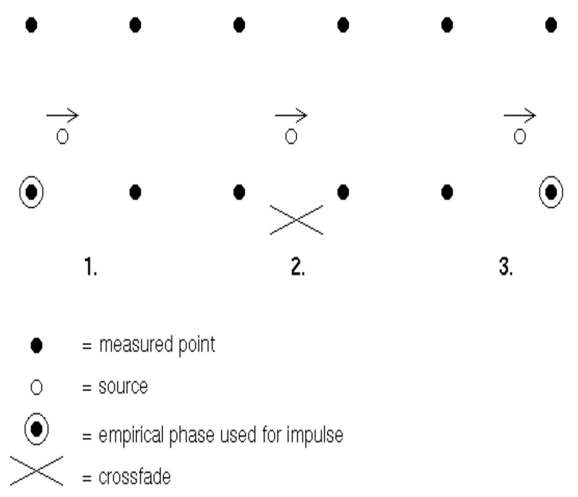


Figure 3: Magnitude Interpolation, Phase Truncation

3.3 Functional Phase Model

A second approach to phase interpolation is presently suggested. Magnitude interpolation, as described above, is employed. Phase can be derived functionally, by assuming the head is a sphere. This will clearly introduce inaccuracies in phase values, which are addressed in a psychoacoustically motivated way here.

To derive an ITD based on a spherical head, the following formula can be used:

$$\frac{r(\theta + \sin\theta)}{c}\cos\varphi \qquad (2)$$

Head (/sphere) radius is represented by *r*, *c* is the speed of sound, *θ* source azimuth and *φ* source elevation. Phase values can be derived from this ITD values. This formula (Woodworth's formula) is used to derive phase values in [17]. An augmentation and improvement of the formula is presently offered, again aiming to use empirical data more directly, in a psychoacoustically motivated manner. As discussed above, accuracy of low frequency ITD is crucial for localisation. Therefore, more accurate low frequency phase values are desirable. The following method aims to introduce phase accuracy into the relevant low frequency end of the spectrum, while utilizing the functional model.

ITD is understood as phase differences between the signals to the left and right ear, and breaks down above 1500 Hz, becoming progressively worse towards this threshold [15]. Interaural phase difference (IPD) is therefore much more significant at lower frequencies. Higher frequencies are not as important for phase based localisation. This is confirmed by the experiments completed in [12], discussed above. Furthermore, it is specified in [13] that this functional model can account for steady state high frequency ITDs.

It is thus proposed that augmenting the functional model with more accurate low frequency phase values will provide a more psychoacoustically accurate model. This is achieved using scaling factors, extracted from the empirical data. IPD becomes ambiguous for phase differences of greater than 180 degrees, leading to unreliable localisation. Therefore, the maximum unambiguous frequency for a specific source location is first calculated. IPDs are then derived for frequency bins below this limit or the 1500 Hz threshold in a 128 sample Fast Fourier Transform of each HRTF pair in the dataset. ITDs are derived from these IPD values, and compared to the ITD derived from the functional model. A scaling factor can thus be calculated for each bin of interest. The values derived from the full dataset are then averaged, and applied in the artificial spatialisation process. This scaling factor increases the accuracy of the functional model, basing it on the empirical data for the dataset used. Figure 4 shows the scaling factors for the frequency bins of interest. It is clear that ITD is largely underestimated by the functional model, an error corrected by scaling the phase values derived from the functional model. When processing dynamic source trajectories using this technique, interpolated HRTFs are derived as the user defined trajectory is updated. Magnitudes are interpolated linearly. Phase values are derived using the functional model. An appropriate IPD is calculated in the frequency domain. Values below 1500 Hz are scaled using the scaling factors extracted from the empirical data. An accurate low frequency ITD is thus derived, with an adequate high frequency ITD, fitting the actual behavior of ITD discussed in [13]. Dynamic sources are processed using the Short Time Fourier Transform (STFT) to avoid noise introduced by changing phase spectra for dynamic sources.

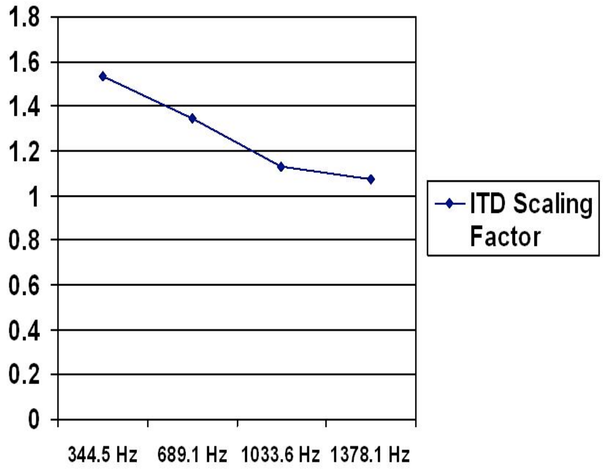


Figure 4: ITD Scaling Factors

As described above, phase values are calculated per frequency bin, with values of plus and minus half the ITD for the ear nearer and further from the source respectively. The low frequency spectrum is scaled accordingly. This results in the HRIR wrapping around its zero time point. It therefore appears that the nearer ear response happens after the further ear, which is an unnatural result. For this reason, the HRIR is shifted in time, by half the size of the buffer. The result is a time and phase accurate filter. Figure 5 shows 2 stereo HRIRs. They both represent the HRTF for 0 degree elevation and 90 degree azimuth. Therefore, the right ear should receive the signal first. In the first HRIR, this is not the case, as the functionally derived phase spectrum wraps around the zero phase point, moving the right ear response to the end. However, shifting the HRIR will clearly result in a time accurate result, as can be seen in the second HRIR. The interaural phase spectrum between the left and right responses is correctly imposed on both examples, but the second HRIR is now correct in peak onset time, i.e. the right ear receives its signal before the left.

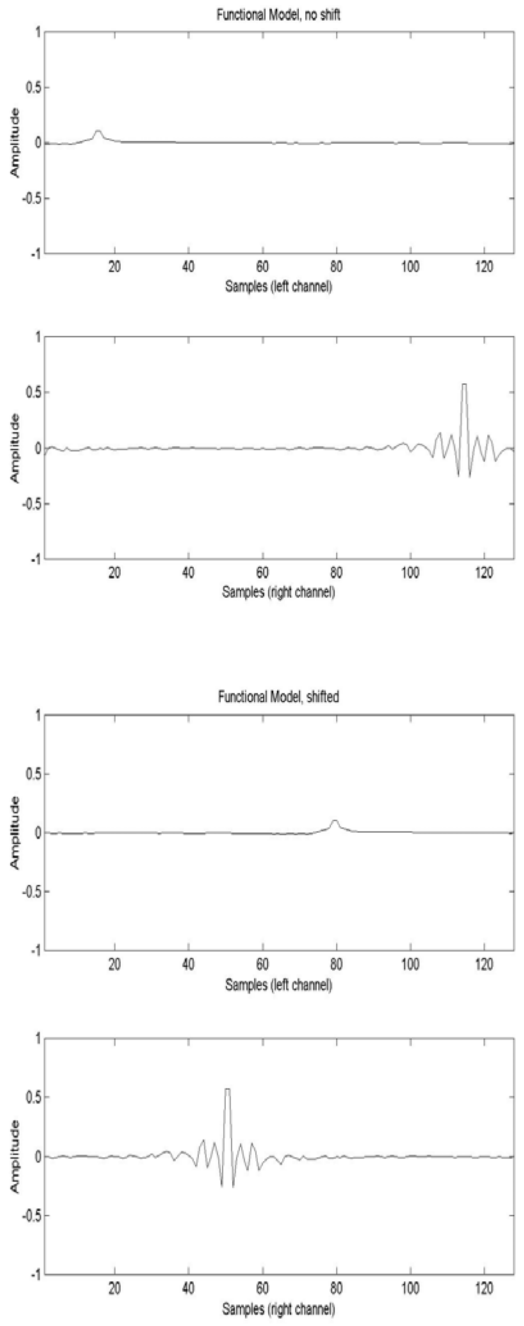


Figure 5: Non-shifted and Shifted Functional Impulses

Interestingly, adding this time alignment provides much better static localisation. It is assumed that different onset times lead to confusions when processing sources with dramatic envelopes.

4. VERIFICATION AND TESTING OF ALGORITHMS

4.1 Objective Testing

An objective test was developed to investigate the accuracy of the functional phase model: the spherical head augmented by non linear low frequency scaling against that of a minimum phase plus delay implementation, prepared as in [12]. The dataset of 368 HRIR data files was initially transformed to a minimum phase plus delay and a functional model dataset. Each HRIR was then upsampled by a factor of 4 to increase the accuracy of the evaluation. As the phase spectrum of the new data files is what we wish to examine, and as low frequency ITD is the psychoacoustically relevant area of the spectrum on which we are focusing [12], each HRIR is low pass filtered. ITDs are then calculated for each pair of filtered responses (left and right), and compared to those of the empirical data. Ideally, these values should agree. However, results from the functional model are expected to deviate due to the averaging of the low frequency scaling factors. To avoid the necessity for an individual curve representing the scaling factors for each data file, all curves are averaged. This means that only one curve of scaling factors needs to be stored and referenced. Any deviations from parity with empirical data in the minimum phase plus delay implementation are due to non linearities in the all pass component of the minimum phase plus delay implementation.

Results for the overall dataset are illustrated in figure 6. The minimum phase plus delay dataset deviates from the empirical data by a total of 1076 samples. The functional model, using a head radius of 8.8 cm, deviates by 827 samples. The elevation 0 degrees subset of the data is of particular interest for artificial spatialisation applications. ITD and IID are most pronounced in this horizontal plane, where artificial spatialisation will be most effective. Applications may choose to only operate in this 2 dimensional space. Deviations from empirical data are given for the datafiles at elevation 0 in figure 7. Once again, a cursory look at the graph shows that the minimum phase plus delay model deviates further from empirical data than the functional phase model. Specifically, minimum phase plus delay processing results in an error of 183 samples, whereas the functional model gives a 156 sample error, for reasons discussed above. Further analysis of the data at individual elevations provides an interesting insight into the nature of the dataset, and the accuracy of the methods under evaluation. At lower elevations and extremes of the horizontal plane, the minimum phase plus delay model performs most poorly. At higher elevations, however, the model performs better. This agrees with previously discussed conclusions reported in [12]. The functional model performs better at lower elevations, suggesting that the averaged data better fits this spatial area.

Overall, results therefore show the inaccuracies introduced in the minimum phase plus delay implementation, as well as the increased accuracy offered by the functional model, despite the averaging of data for efficiency. The goal of this method is therefore validated: to derive a psychoacoustically motivated fit for ITD using an efficient phase model.

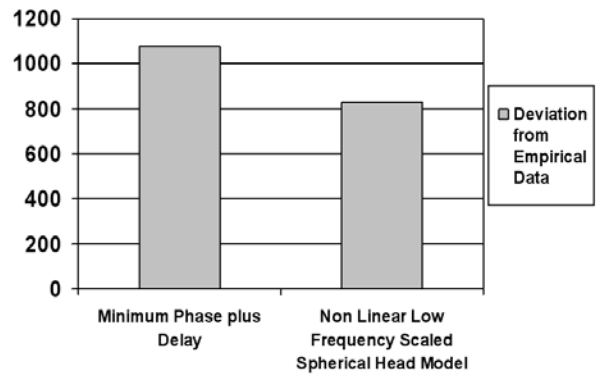


Figure 6: Degree of error in low frequency ITD of minimum phase plus delay and functional model over entire dataset.

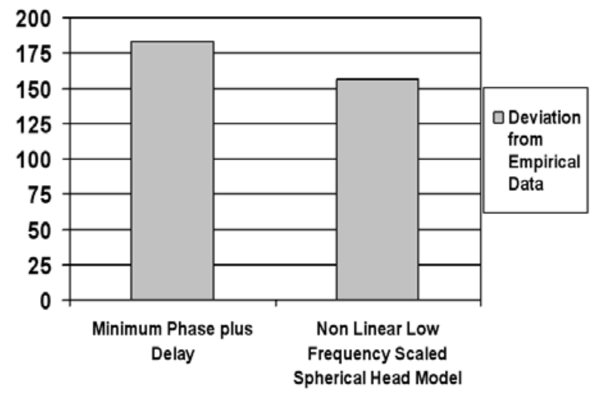


Figure 7: Degree of error in low frequency ITD of minimum phase plus delay and functional model for 0 degree elevation subset of data.

4.2 Subjective Testing

As the main purpose of the novel algorithms is implementation of artificial spatialisation applications, subjective tests were developed to test both new algorithms. Typically, dynamic source trajectories are desirable in spatialisation applications, so moving sources were used as source material. The basic design of the test is similar to the A/B/Ref test defined in the GuineaPig generic testing system as 'Three samples are played. Samples A and B are graded against the reference' [19]. In each test, two moving sources are rated against a reference. The moving sources were prepared using 4 algorithms: the minimum phase plus delay method [12], the phase truncation method, the functional phase method and a method using no magnitude or phase interpolation. The last algorithm is introduced as an anchor condition (as per the MUSHRA method discussed in [20]. Anchors are test signals which are perceptually impaired purposefully to provide a base level. The anchor condition is used in a slightly different way here). No interpolation in moving sources will introduce abrupt changes in the filter used to process the input audio. The anchor condition is therefore expected to perform poorly. The nature of the desired test and the available source material imposes some restrictions. As moving source HRTFs are not available, the reference used involves 2 statically processed files: the source sound spatialised to the start and end positions of the trajectory in question. Start, mid and end points were initially considered to aid listeners in discerning audible changes. However, mid points were deemed unnecessary by listeners in early trials. Subjects were asked to rate each moving file based on smooth, artifact free trajectories. A 5 point grading scale was used, as in [21]. Subjects were informed that spatialisation accuracy was not to be considered in their ratings. The quality scale was also clearly defined to subjects. Using non-individualized datasets can lead to spatialisation inaccuracies [2], and the aim of the experiment was to validate that the new algorithms provide smooth, artifact free processing. The algorithms can then be used with other datasets, for increased accuracy/individualization.

The setup of the test allows subjects to listen to any spatialised source any number of times. A supervised training period consisting of 3 trials is also presented, to familiarize subjects with the sound sources, interface and task. 18 actual trials were then presented, constituting 36 dynamic sources to be rated. 3 source sounds were used: a vocal speech sample, a noise burst and a piano phrase. Thus a range of spectral and temporal data was processed. An equal number of trajectories for each source were tested. 9 subjects performed the test. The interface for the test is illustrated in figure 8.

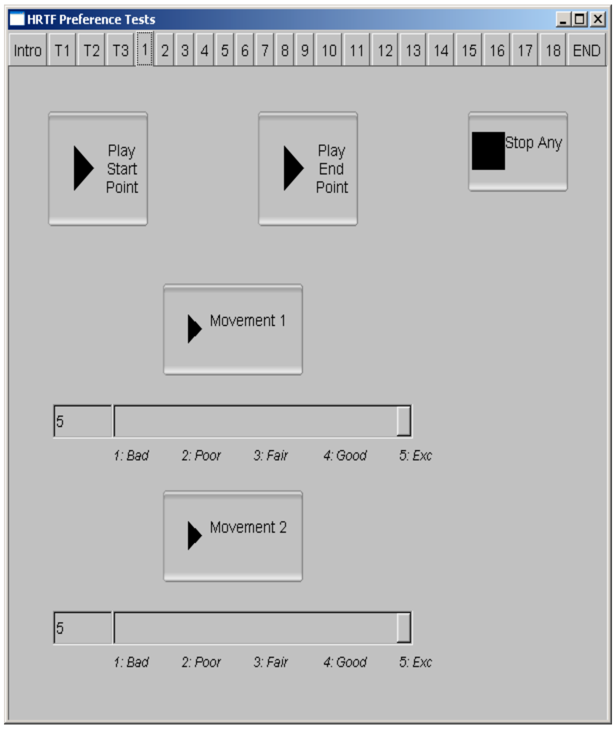

HRTF Preference Tests
Intro T1 T2 T3 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 END
Play Start Point
Play End Point
Stop Any
Movement 1
5
1: Bad 2: Poor 3: Fair 4: Good 5: Exc
Movement 2
5
1: Bad 2: Poor 3: Fair 4: Good 5: Exc

Figure 8: Preference Test Interface

Results of the subjective tests will now be discussed. The mean values for each algorithm are presented in figure 9. The novel algorithms perform better than the minimum phase plus delay method overall. All methods perform well, with the exception of the anchor condition, as expected. The novel algorithms, at 4.639260889 and 4.712208667 for the phase truncation and functional models respectively, perform close to a rating of excellent, defined in the test instructions as ‘Excellent: no distortion or noise: smooth movement that sounds like the sound is convincingly moved from start to end point, without any alterations/clicks/noise added to the sound’. The minimum phase plus delay result of 4.267233333 is a lower rated result, closer to a rating of ‘good’: ‘Good: some slight distortion or noise: Some slight filtering/movement not completely smooth; perhaps the sound is a little altered’. Subjects highlighted the introduction of some artifacts in source movements processed with the minimum phase method. The spectrally rich noise source highlighted these issues. The minimum phase plus delay model performed better with the other sources, whose spectra were more narrow band. The results of the tests performed with the noise source are illustrated in Figure 10.

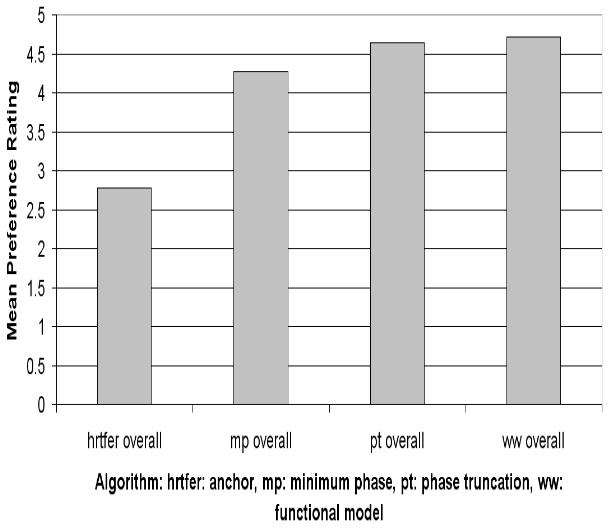


Figure 9: Overall Preference Test Results

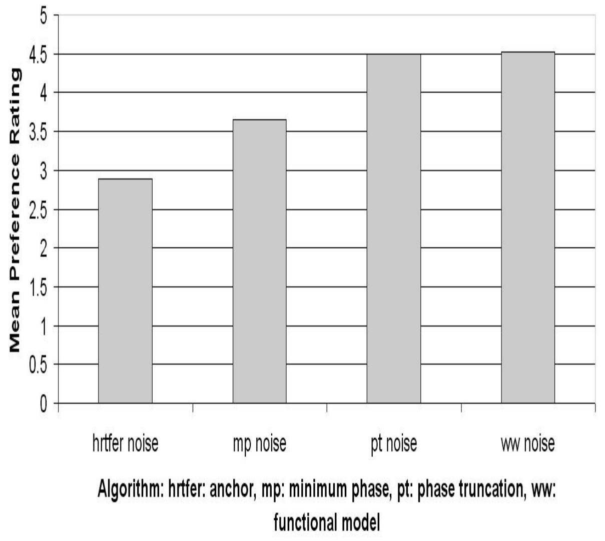


Figure 10: Preference Test Results: Noise Source

5. CONCLUSION

Novel solutions to the challenge of HRTF interpolation and dynamic source processing have been presented. Phase truncation provides an easily realised method which uses empirical data directly, without any necessity for complex data preparation, transformation or compression. Brief crossfades provide a parametric, efficient solution to changing phase values. The method performs very well in subjective tests using dynamic sources.

The functional phase model augments a spherical head approximation of ITD with low frequency, frequency dependent scaling factors based on the empirical data. Empirical low frequency IPDs are used as scaling factors in this psychoacoustically motivated method. Objectively, the method provides a more accurate low frequency ITD than the minimum phase plus delay method over the complete dataset.

The minimum phase plus delay method involves data approximation and complex processing. The novel methods aim to reduce this complexity, and suggest alternative approaches to the challenge of phase interpolation. All 3 methods perform well in objective testing of dynamic sources. However, the novel algorithms both perform significantly closer to a rating of ‘excellent’ than the minimum phase plus delay model.

The most obvious application of the developed algorithms lies in artificial spatialisation processing. The authors are currently developing an artificial reverberation tool, using binaurally accurate early reflections and a binaural diffuse field, both based on the image model [22]. Using the phase truncation algorithm, real time processing of high resolution, dynamic early reflections is possible. It is hoped that this reverb system will be added to a multi channel binaural spatialisation application. Such applications typically assume the listener to be in the sweet spot [4]. Dynamic HRTF processing allows users to move around more freely, auditioning the loudspeaker array. A flexible loudspeaker setup is also desirable, allowing various algorithms and setups, such as Ambisonics, Vector Base Amplitude Panning or Wave Field Synthesis. The previously mentioned reverb tool may be useful in such an application if the inherent multichannel reverberation of the desired method is deemed unsuitable for reasons of accuracy or complexity.

6. ACKNOWLEDGEMENTS

This work was supported by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan and NUI Maynooth.

7. REFERENCES

[1] B. Gardner and K. Martin, “HRTF Measurements of a KEMAR Dummy Head Microphone,” Available at http://sound.media.mit.edu/resources/KEMAR.html , Accessed January 31, 2009.

[2] E. Wenzel, M. Arruda, D. Kistler and F. Wightman, “Localization using non-individualized head related transfer functions,” *Journal of the Acoustical Society of America*, vol. 94, no. 1, pp.111-123, July 1993.

[3] C. Cheng and G. Wakefield, “Moving Sound Source Synthesis for Binaural Electroacoustic Music Using Interpolated Head-Related Transfer Functions (HRTFs),” *Computer Music Journal*, vol. 25, no. 4, pp.57–80, Winter 2001.

[4] M. Noisternig, T. Musil, A. Sontacchi and R. Höldrich, “3D Binaural Sound Reproduction using a Virtual Ambisonic Approach,” *IEEE Symposium on Virtual Environments*, Lugano, Switzerland, July 27-29, 2003, pp. 174-178.

[5] K. Hartung, J. Braasch and S. Sterbing, “Comparison of Different Methods for the Interpolation of Head Related Transfer Functions,” *AES 16th International Conference: Spatial Sound Reproduction*, Rovaniemi, Finland, March 1999, pp.319-329.

[6] J. Jot, V Larcher and O. Warusfel, “Digital Signal Processing Issues in the Context of Binaural and Transaural Stereophony,” *AES 98th Convention*, Paris, France, February 25-28, 1995.

[7] F. R. Moore, *Elements of Computer Music*, Prentice Hall, New Jersey, USA, 1990.

[8] A. Oppenheim, and R. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, New Jersey, USA, second edition, 1999.

[9] S. Mehrgardt and V. Mellert, "Transformation characteristics of the external human ear," *Journal of the Acoustical Society of America*, vol. 61, no. 6, pp. 1567-1576, June 1977.

[10] D. Kistler and F. Wightman, "A model of head-related transfer functions based on principal components analysis and minimum-phase reconstruction," *Journal of the Acoustical Society of America*, vol. 91, no. 3, pp.1637-1647, March 1992.

[11] A. Kulkarni and H. Colburn, "Infinite-impulse-response models of the head-related transfer function," *Journal of the Acoustical Society of America*, vol. 115, no. 4, pp.1714-1728, April 2004.

[12] A. Kulkarni, S. Isabelle and H. Colburn, "Sensitivity of Human Subjects to Head-Related Transfer-Function Phase Spectra," *Journal of the Acoustical Society of America*, vol. 105, no. 5, pp. 2821-2840, May 1999.

[13] G. Kuhn, "Model for the interaural time difference in the azimuthal plane," *Journal of the Acoustical Society of America*, vol. 62, no. 1, pp.157-167, July 1977.

[14] V. Lazzarini and B. Carty, "New Csound Opcodes for Binaural Processing," *Proc. 6th Linux Audio Conference*, Cologne, Germany, February 2008, pp. 28-35.

[15] B. Moore, An *Introduction to the Psychology of Hearing*, Elsevier Academic Press, London, UK, fifth edition, 2004.

[16] L. Savioja, J. Huopaniemi, T. Lokki and R. Väänänen, "Creating Interactive Acoustic Environments," *Journal of the Audio Engineering Society*, Vol. 47, No. 9, pp. 675-705, September 1999.

[17] D. Zotkin, R. Duraiswami and L. Davis, "Rendering Localized Spatial Audio in a Virtual Auditory Space," *IEEE Transactions on Multimedia*, vol. 6, no. 4, pp.553-564, August 2004.

[18] http://www.csounds.com/manual/html/hrtfmove.html.

[19] J. Hynninen and N. Zacharov, "GuineaPig – A generic subjective test system for multichannel audio," *AES 106th Convention*, Munich, Germany, May 1999.

[20] ITU-R. Recommendation BS.1534, "Method for the Subjective Assessment of Intermediate Quality Level of Coding Systems," *International Telecommunications Union Radiocommunications Assembly*, 2001.

[21] ITU-R. Recommendation BS.1284, "General methods for the subjective assessment of sound quality," *International Telecommunications Union Radiocommunications Assembly*, 1997.

[22] J. Allen and D. Berkley, "Image method for efficiently simulating small-room acoustics," *Journal of the Acoustical Society of America*, vol. 65, no. 4, pp.943-950, April 1979.

BINAURAL HRTF BASED SPATIALISATION: NEW APPROACHES AND IMPLEMENTATION

Brian Carty and Victor Lazzarini

Sound and Digital Music Technology Group,
National University of Ireland, Maynooth
Co. Kildare, Ireland
`brian.m.carty@nuim.ie`,
`victor.lazzarini@nuim.ie`

ABSTRACT

New approaches to Head Related Transfer Function (HRTF) based artificial spatialisation of audio are presented and discussed in this paper. A brief summary of the topic of audio spatialisation and HRTF interpolation is offered, followed by an appraisal of the existing minimum phase HRTF interpolation method. Novel alternatives are then suggested which essentially approach the problem of phase interpolation more directly. The first technique, based on magnitude interpolation and phase truncation, aims to use the empirical HRTFs without the need for complex data preparation or manipulation, while minimizing any approximations that may be introduced by data transformations. A second approach augments a functionally based phase model with low frequency non-linear frequency scaling based on the empirical HRTFs, allowing a more accurate phase representation of the more relevant lower frequency end of the spectrum. This more complex approach is deconstructed from an implementation point of view. Testing of both algorithms is then presented, which highlights their success, and favorable performance over minimum phase plus delay methods.

1. INTRODUCTION

Our ability to locate sound sources in our spatial environment depends primarily on the binaural nature of our auditory system. We can use interaural time and intensity differences (ITD and IID respectively) to help us in this task, often very accurately under favorable conditions. These interaural cues have frequency limitations. Generally, ITD performs best at low frequencies and IID at high. Monaural information can also provide important localisation cues. The pinna filters audible incoming sound in a non-linear manner due to its complex shape.

These cues will all be evident in the Head Related Transfer Function (HRTF) of the left and right ear of a specific listener, with regard to a specific source location relative to this listener. HRTFs essentially define how a sound from a particular location is altered from source to tympanic membrane. An arbitrary mono, non-localized source can then be artificially spatialised to the location of this HRTF pair by convolving it with the left and right ear HRTFs and playing the resulting stereo file in headphones.

Such a system appears promising for artificial spatialisation; however, limitations must be recognized. HRTFs are individual specific, for physiological reasons. Consistencies can however be observed in external ear characteristics, leading to the frequent use of generalised/non-individualized HRTF data sets in artificial binaural spatialisation scenarios. The finer detail of localisation ability, for example elevation resolution and front/back confusions in areas where interaural cues will be similar can be degraded in this scenario, but it is suggested in [1] that non-individualized data sets are certainly a useful tool in artificial spatialisation applications.

HRTF datasets typically record and store a fixed number of responses around a subject, for various azimuths and elevations, for example [2]. If sources are required to be spatialised to a non-measured point, or move smoothly from point to point, an interpolation algorithm is required.

Several approaches to this complex task have been suggested. Essentially, the interpolation process can be thought of as the derivation of a new HRTF by combining values from known empirical HRTF measurements. Known points in the vicinity of the desired non-measured point can thus be read and combined with relative weightings with regard to the desired point.

This interpolation process is more accurately performed in the frequency domain, which immediately raises the issue of phase interpolation. As ITD uses phase differences in locating sounds, phase values in HRTFs are clearly significant. Phase is, however, a periodic quantity, therefore phase interpolation is problematic.

Traditionally, this difficulty has been overcome using a minimum phase allpass decomposition of the HRTF. By assuming the allpass component is linear, this becomes a minimum phase plus delay decomposition. This paper will first present a review of the standard minimum phase method. Following this, we will introduce two novel approaches to the problem, considering their motivation and implementation. Finally, we will present test results illustrating their favorable performance.

2. MINIMUM PHASE HRTF ASSUMPTION AND INTERPOLATION

Any rational system function can be broken into a minimum phase and an allpass system [3]. The magnitude of the minimum phase all pass decomposition is represented solely by the minimum phase system and the phase is reconstituted by both the allpass and minimum phase representations. The system in question can thus be defined as:

$$H(z) = H_{\min}(z)H_{ap}(z) \qquad (1)$$

where $H_{\min}(z)$ is a minimum phase system and $H_{ap}(z)$ is an allpass system.

Typically, magnitude and phase spectra are not related. A unique and, in this case, extremely useful property of minimum phase systems, however, is that phase values for each component frequency can be derived from their corresponding magnitude values.

It has been asserted that the allpass component of the decomposition of the HRTF into minimum phase and allpass components approximates linearity in [4]. In this study, the authors performed the decomposition of measured transfer functions in order to avoid the above mentioned phase uncertainty. They wished to obtain an unambiguous representation of the phase of their functions. In so doing, they realised that almost all of the fine detail in the phase of their free field to ear canal transfer functions was contained in the minimum phase components. They concluded that the allpass component thus approaches linearity.

Furthermore, the allpass component of the full HRTF (including the ear canal response) exhibits a 'nearly linear' phase response up to 10 KHz. Consequently, the external ear can be thought of as a minimum phase system within this range. This implies that the allpass component can be approximated using a time domain, frequency independent delay line. Thus phase interpolation is no longer a problem, as phase can be derived from magnitude for the minimum phase part of the decomposition, and delay lines can be interpolated. This observation has become the basis for many HRTF based binaural processing algorithms.

Each empirical HRTF pair is thus analysed to extract an appropriate interaural delay and reduced to a minimum phase representation. Interpolation can be performed on the delay and magnitude values. Interpolated minimum phase phase values can then be derived from these interpolated magnitude values.

The decomposition of impulses in [4] theoretically validates the description of HRTFs as minimum phase filters (the transfer function can be thought of as a filter operation) plus delays. A typical motivation regarding the study of HRTFs is the implementation of an artificial spatialisation system. Such an application is perhaps more concerned with more subjective testing. Therefore, the seminal paper by Kulkarni et al. examining the sensitivity of human subjects to HRTF phase spectra [5], which details psychophysical tests performed on a subject group is of great significance. Initially, while objectively investigating the validity of the minimum phase assumption, the study reports high coherence values between empirical and minimum phase plus delay data sets. However, coherence values were found to be systematically worse at lower elevations and extremes of the horizontal plane. It is suggested that this is due to the shadowing effect of the head and interactions with the torso making the allpass delay non-linear, a phenomenon also discussed in [6]. This is supported by better performance at higher elevations, where there is less obstruction in the path to the contralateral (further from the source) ear. Phase error results enforce this assumption. These specific cases when minimum phase plus delay may not be valid are also mentioned in [7], where some possible solutions are discussed.

The psychophysical results from [5] further clarify this issue, highlighting a low frequency cue present at extremes of the horizontal plane, helping the subject to distinguish between minimum phase plus delay and empirical impulses. Therefore, the suitability of modeling the interaural delay as a linear delay is brought into question. The study, however, concludes that minimum phase plus delay models are sufficient for most locations (and therefore adequate), and that the finer structures of phase are not excessively important, as long as the overall delay is approximated in accordance with that of the low frequency empirical ITD. The benefits of minimum phase plus delay, specifically its ability to deal with phase interpolation and efficiently express the filter with the lowest possible number of coefficients (as the energy in a minimum phase impulse will be focused at its start) typically justify its use.

To conclude this analysis of the minimum phase plus delay HRTF representation, practicalities of implementation of the desired real time artificial spatialisation system need to be considered. In such an application, (the design of which is based on the minimum phase assumption) delay lines need to be interpolated, which adds complexity and possible spectral distortions to the output signal. The method of delay extraction is also pertinent. Several methods have been suggested, again adding to the processing and preparation required.

3. NOVEL APPROACHES TO EMPIRICAL DATA INTERPOLATION

3.1 Motivation

The initial and primary aim of this study is to provide a toolset for the artificial recreation of audio spatialisation using HRTF based binaural techniques for open source computer music languages. Tools recently developed by the authors are discussed in [8] from a point of view of implementation for a particular computer music programming language, Csound. The developed algorithms are also introduced in [9] (more detail is given here). Further insight into algorithm testing is also given in [9].

Secondary to this goal, alternatives to the minimum phase approach are suggested that do not assume the approximations involved in modeling the HRTF as minimum phase plus delay. This essentially involves engaging more directly with the phase ambiguity problem. Thus approaches are developed that remove the approximation involved in the minimum phase assumption, as well as the complex data preparation/online processing necessary in minimum phase implementation, while exploiting the apparent insensitivity to phase spectra reported in [5]. The approaches outlined below are also intended to give spatially accurate and efficient processing while dealing more directly with the empirical data. Complex data analysis, compression or transformation necessary in other approaches is thus purposefully minimized to enable convenient, immediate use of HRTFs. The two novel approaches suggested are discussed below.

3.2 Phase Truncation, Magnitude Interpolation

The first of the two new methods proposed introduces phase truncation as a novel addition to linear interpolation methods. The spectrum of the employed HRTF is derived from interpolated magnitude values and the nearest available empirical phase values. An impulse is thus derived for each block of audio processed in the case of a dynamic source. The method provides a simple, intuitive solution to HRTF interpolation for non measured points and performs particularly well in subjective tests.

A user defined, dynamic source trajectory is implemented by updating angle (azimuth) and elevation values for each processing block. The HRTF data in the employed data set [2] is stored as a group of values for particular angles at various elevation increments. Linear interpolation is performed on the magnitude values. This method derives an intermediate/transitional FIR filter that is consistent with the local empirical data, boosting or attenuating spectral bands appropriately.

Possible anomalies in the impulse response of non-measured points are not addressed by this method, although in a dense dataset (the MIT dataset has a resolution of 5 degrees in the horizontal plane for the 0 degree elevation subset of measurements), bearing in mind minimum audible movement constraints [10] and other limitations of the auditory spatialisation system, the technique provides a good approximation. The preference tests discussed below also attest to the perceived smooth movement of sources dynamically spatialised using the method. Filters used are 128 samples long, and are processed using overlap add (rectangular window) convolution in the frequency domain. Noise introduced by filter magnitude values changing as the source moves through a trajectory adhering to minimum audible movement angle limits is inaudible/tolerable. This linear interpolation method is utilized in Savioja et al. [11], who use a minimum phase approach to phase interpolation (as discussed above), Xiang et al. [12], who use time domain processing (which is not efficient and can introduce errors) and Zotkin et. al [13], whose approach will be discussed below.

A novel addition to this interpolation algorithm is the truncation of phase values, and subsequent processing. Intermediate filters use nearest measured phase values. It is proposed that choosing the phase of the nearest measured point in a dense dataset will not have a significant effect on the perceived spatial quality of the result. As discussed above, it has been shown that phase does indeed play an important role in localisation, but exact phase accuracy is not essential [5].

Of immediate concern is the update of these phase values as a source moves closer to the next empirical HRTF on a desired trajectory. Abruptly switching between phase values is undesirable, as it could potentially cause inconsistencies in the output. Brief crossfades are suggested to avoid this. The frequency content of the source defines the audibility of the switch. Frequency rich sources may be able to mask any artifacts caused by a switch in phase values. However, sources with energy focused on one spectral region/narrowband sources will typically not perform as well in this scenario, leading to inconsistencies in the output.

Therefore, in the Csound implementation of this algorithm, the user can simply define the length of crossfades required depending on the source they are working with, if they wish to deviate from a suggested default. Buffers of 128 samples are processed in each iteration. A crossfade over one such buffer may be enough to mask inconsistencies for frequency rich sources. Users may find that other sources may require crossfades lasting up to 16 buffers to mask all artifacts. The old HRTF data is processed with the input data and faded out. Simultaneously, the new HRTF data is processed with the input and faded in. Thus inconsistencies are removed in a simple, source specific (if required) manner. These brief crossfades will typically be infrequent. For static/slow moving sources, no/very occasional crossfades will be needed. For more quickly moving sources, more crossfades will be required, however in all cases, only very brief periods of crossfade are needed.

Figure 1 gives an overview of the algorithm. Three snapshots in time are illustrated. In the first, the source is nearest to the bottom left empirical value, so uses its phase spectrum. In the second, a crossfade occurs as the source moves from being closer to one empirical point to another. Finally, the source is closer to the bottom right point, so uses its phase spectrum. Relatively weighted magnitude values will be used in accordance with source location.

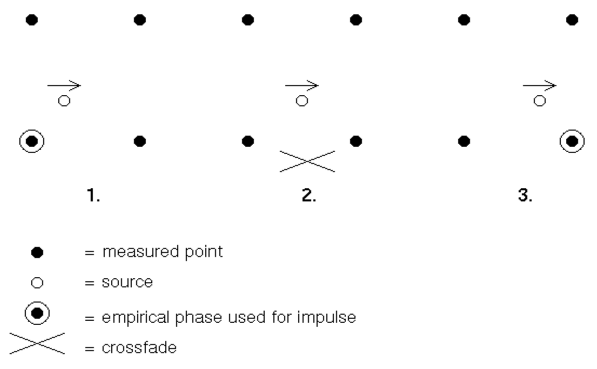


Figure 1: *Magnitude interpolation, phase truncation.*

3.3 Functional Phase Model

3.3.1 Woodworth/Schlosberg Formula

Spectral magnitude interpolation, as discussed above and in the literature, is straightforward, easily realizable and performs adequately, and is employed again in the second suggested novel approach. Again, the derivation of the phase spectrum constitutes the novel aspect of this approach. Essentially, empirical magnitude interpolation is coupled with a functionally modeled phase spectrum. Interaural Phase Difference (IPD) is essential in the derivation of a correct ITD. When endeavoring to functionally model the phase spectrum, the head can be roughly approximated to a sphere. This simplification can be practically implemented mathematically: the ITD for a particular source location, assuming a spherical head can be defined thus:

$$ITD(\theta,\varphi) = \frac{r(\theta + \sin\theta)}{c}\cos\varphi \qquad (2)$$

where r is the head (/sphere) radius, c is the speed of sound, θ is the angle (azimuth) and φ the elevation of the source. This formula is described as the Extended Woodworth/Schlosberg Formula in [14]. Successful use of this basic Woodworth model for HRTF phase modeling and a magnitude interpolation algorithm is reported in [13], and is augmented and advanced here. The formula is also successfully utilized in [11]. Simplifying the complex shape of the head to that of a sphere will distort the HRTF. This distortion is closely related to the discussion above on sensitivity to phase spectra, which concluded that low frequency ITD is the predominant phase cue [5]. Therefore the novel addition to the method aims to reproduce more accurately this low frequency ITD.

3.3.2 Low Frequency Scaling

Accurate ITD modeling involves maintaining a modeled low frequency ITD that is consistent with empirical values [5]. This is done by improving the Woodworth/Schlosberg formula. Higher frequency ITD is not as significant, as agreed in [5] and [6], which specifies that a Woodworth model can account for steady state high frequency ITDs. From a physiological point of view, IPD based localisation breaks down above approximately 1500 Hz [10], becoming progressively less accurate towards this threshold. A low frequency, frequency dependent scaling factor is therefore suggested as an addition to the Woodworth/Schlosberg formula. It is proposed that this provides a more complete, psychoacoustically based solution, with minimal extra processing required.

Primarily, psychoacoustically based parameters are imposed on the range of the spectrum to be scaled. As mentioned above, IPD breaks down above approximately 1500 Hz; therefore this value is used as the upper boundary for scaling. Physical IPD restrictions for sinusoidal sources can be further quantified by finding the maximum unambiguous frequency for a specific source location. At IPDs of 180 degrees and greater, the source location is uncertain. The right signal may be leading the left, or vice versa. As with phase interpolation, this uncertainty is a result of the periodic nature of phase. As IPDs get larger, a greater number of perceived source locations are possible, as a number of full phase cycles may be incorporated into the reported IPD. The maximum frequency for a specific source location can be calculated thus:

$$f_{\max} = \frac{c}{2r(\theta + \sin\theta)(\cos\varphi)} \tag{3}$$

where r is the head radius (again assuming a spherical head), c is the speed of sound, θ is the angle (azimuth) and φ the elevation of the source. This essentially represents the frequency that corresponds to half the distance around the head to the opposite ear.

This formula is used, where appropriate, to reduce this 1500 Hz threshold. The radius used here is that of the largest radius derivable from the KEMAR [15] mannequin measurements to minimize the value used. This reduction is maximized at the horizontal extreme of the half of the spatial hemisphere used (the left hemisphere is simply an inverted copy of the right in the dataset used [2]). A maximum IPD of π is implied by this methodology, which is the highest realizable resolution without phase ambiguities in a typical situation. However, although unnecessary here, resolution to 2π is possible, as the source location direction is known. ITD is, in these circumstances, a vectorial quantity. In relation to the ear nearest to the source position, the ITD will have positive orientation, whereas the other ear will have a negative ITD.

In practical terms, impulses will always come from the right if the angle is less than 180 degrees (with the exception of 0 and 180 degrees, where there is no IPD). As the right phase is positively oriented and the left negatively in this scenario, IPD can be defined as right phase minus left. If there is an anomaly in this calculation (if the phase difference has passed onto a new cycle), the right phase is augmented by 2π.

ITDs are derived from empirical IPDs and compared to Woodworth/Schlosberg ITDs. Scaling factors are then calculated. The average of all derived scaling factors for each bin of the low frequency spectra of the HRTFs are shown in Figure 2, for a Fast Fourier Transform (FFT) size of 128 samples (we are using the compact, diffuse field filtered HRTF data from [2]). The bins of interests are shown in the figure, up to the 1500 Hz threshold.

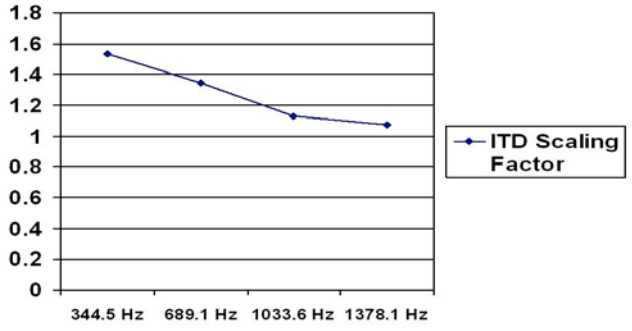


Figure 2: *ITD scaling factors.*

A larger sample block FFT, giving more spectral resolution, reveals some interesting characteristics of this particular dataset. The curve is predominantly > 1, as expected [6], and illustrated in Figure 2. Some anomalies do appear on closer inspection, however. For example the curve falls below 1 at angle 150 degrees, elevation -30 degrees. However, the curve generally fits Figure 2 well, so the averaged model is used across location for efficiency.

The values derived from this Extended Woodworth /Schlosberg Non-linearly Low Frequency Scaled Spherical Head (functional) Model are then used in the re-synthesis of the phase spectrum. Essentially, an appropriate ITD is derived from the Woodworth/Schlosberg formula. In the frequency domain, the appropriate phase is then calculated. For frequencies below 1500 Hz, the ITD value is scaled in accordance with the averaged scaling factor, which is derived from the empirical data. This model provides an accurate average low frequency ITD for this particular dataset, and a steady Woodworth based ITD for higher frequencies, providing a psychoacoustically derived fit of the actual behavior of ITD [6]. Overlap-add convolution leads to undesirable noise when processing dynamic source trajectories, due to derived phase values not 'matching' amplitude values, so Short-time Fourier Transform (STFT) processing is used.

4. ALGORITHM TESTING

4.1 Objective Tests

The non linear low frequency scaling of the functional model was tested numerically to compare it to the minimum phase plus delay model. Primarily, all 368 data files in the empirical dataset were transformed into minimum phase plus delay and functional model datasets. The minimum phase plus delay dataset was prepared as in [5]. We wish to highlight not only that the novel algorithm performs well, but also the approximations involved in assuming that the allpass component is linear in the minimum phase plus delay algorithm. Datasets were then upsampled to 4 times their sampling rate (44,100 * 4 Hz) to provide a more accurate evaluation. Each HRTF pair was run through a low pass filter, to focus on the lower end of the spectrum, where ITD is more significant as a spatial localisation cue [10]. ITDs for each filtered HRTF pair were then calculated, by finding the maximum of their interaural cross correlation. The filtered minimum phase and functional model ITDs were then compared to the filtered empirical ITDs.

Ideally, both algorithms should agree with the empirical data. However, this is not always the case. When all data is considered, the minimum phase plus delay ITDs deviate from the empirical data by a total of 1076 samples over the entire dataset. This is due to the non-linearities involved in the allpass component of the minimum phase allpass deconstruction. The functional model deviates by 827 samples for a head radius of 8.8 cm. This deviation is due primarily to inaccuracies introduced by averaging of the scaling factors over the whole dataset, which was performed for efficiency. Therefore the novel suggested method is validated, as its main goal is to provide a more accurate low frequency ITD, due to its importance in localisation [5]. This result is illustrated in Figure 3, which shows that the minimum phase plus delay model involves a greater deviation from empirical data.

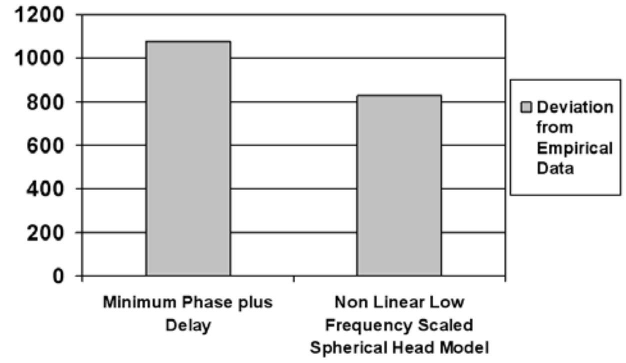


Figure 3: *Objective test illustrating that ITD of introduced functional model is closer to empirical data than minimum phase plus delay for low frequencies.*

4.2 Subjective Tests

Subjective tests were also performed to rate both of the novel algorithms. Due to the nature of the novel algorithms and the desire for source movement being the motivation for the study, a moving source A/B/Ref based test was developed.

The GUI for the test was developed using Csound's FLTK opcodes. Due to the restriction of not having a true reference signal (a moving source recorded under the exact conditions of the dataset), the source in question processed with static start and end point empirical HRTFs constitutes the reference. The minimum phase (as prepared in [5], using overlap add convolution, as the phase truncation algorithm does), phase truncation, functional model and an anchor condition were tested. The anchor condition uses the same dataset to spatialise sounds, but no interpolation. Therefore it was expected to perform poorly.

Subjects were asked to rate the dynamic samples according to a 5 point quality grading scale [16]. These ratings were based on smooth, artifact free movement from start to end point. Note that non-individualized HRTFs were used here, which can lead to front-back confusion and localisation inaccuracies [1]. Therefore, spatial location is not being assessed in this test. This is also explicitly confirmed as a note to participants in the test's instructions. Subjects were permitted to repeat playback of reference and sample files, as desired. Also, subjects could stop samples if required, and could not play more than one sample at a time.

Three sample tests were presented, constituting a training period, followed by 36 dynamic sources to be judged. The purpose of the training period was to familiarize subjects with the sound samples, task and interface. A screenshot of the interface is given in Figure 4. It shows the reference signals (start point and endpoint) and two movements to be rated.

The three different sound sources used were: a vocal sample, a noise burst and a brief musical figure played on piano, representing a range of spectral and temporal changes in sources. Nine subjects were tested, all of whom had experience with critical headphone listening. Overall results are illustrated in Figure 5. The mean values for each algorithm are presented. As expected, the anchor algorithm performs significantly worse than the others. Interestingly, the means indicate that the novel algorithms introduced here perform better than the minimum phase plus delay method. All 3 algorithms are within the range from good to excellent, however, the novel algorithms are closer to excellent, at 4.6 and 4.7 for the phase truncation and functional models respectively. The minimum phase plus delay method, at 4.3, clearly has a lower mean. Results of a Friedman test show a statistically significant difference between algorithm ratings.

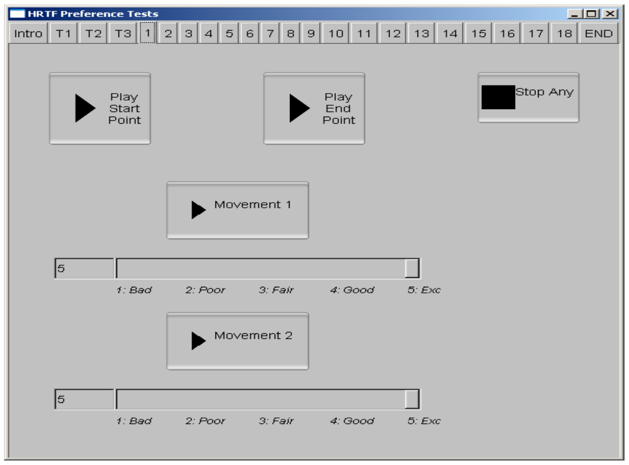

HRTF Preference Tests
Intro T1 T2 T3 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 END
Play Start Point
Play End Point
Stop Any
Movement 1
5
1: Bad 2: Poor 3: Fair 4: Good 5: Exc
Movement 2
5
1: Bad 2: Poor 3: Fair 4: Good 5: Exc

Figure 4: *Preference test interface.*

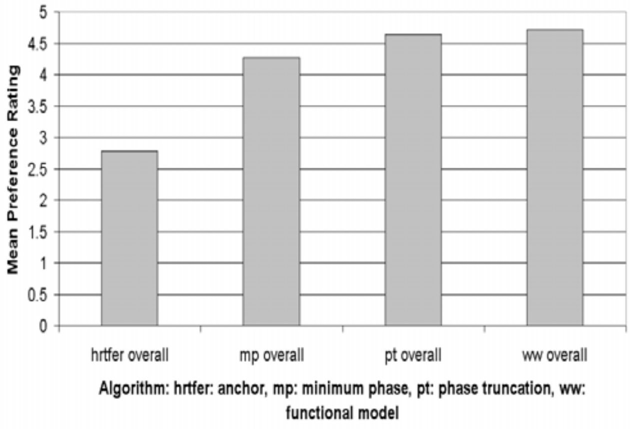


Figure 5: *Preference test results.*

5. CONCLUSION

A critique of the minimum phase plus delay method of dynamic binaural spatialisation is offered. It requires complex data preparation and digital signal processing, as well as data approximations. Novel methods for the interpolation of HRTFs have been presented and discussed. The phase truncation method described maintains nearest measured phase data, thus meeting the criterion of using empirical data directly. Smooth, artifact free, user definable complex trajectories are possible with this method. Change of phase information is dealt with using brief crossfades, which users may tailor to the spectral content of their source sound if desired.

As discussed in [5], HRTF phase data does not require exact accuracy. More specifically, maintaining low frequency interaural time delays appears to provide accurate phase data. The more complex functional model introduced works on this assertion. Augmenting the simplification of the head to a sphere with non linear frequency scaling factors for the psychoacoustically relevant low frequency end of the spectrum will reintroduce some of the more significant finer phase detail of the head, pinnae and torso. The algorithms involved are discussed in detail, and some insight is given into the phase response of the particular dataset used, as well as the vectorial nature of ITD. The importance of low frequency phase information is preserved and applied to an efficient, simple model for phase.

Both objective and subjective tests are presented. The functional model is numerically validated by examining the lower end of the spectrum for all impulses in the dataset. This shows a low frequency ITD that agrees more closely with the empirical data than a minimum phase plus delay model. Subjectively, both the phase truncation and functional model perform better than the minimum phase plus delay algorithm.

The novel methods mentioned above, as well as the minimum phase based method have been implemented as Csound opcodes [8, 17]. A HRTF based reverb system is currently being completed, adding HRTF accurate early reflections and a binaural statistical diffuse field to sources spatialised using the opcodes developed.

6. ACKNOWLEDGEMENTS

This research is supported by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan and NUI Maynooth. V. Lazzarini's work was supported by An Foras Feasa.

7. REFERENCES

[1] E. Wenzel, M. Arruda, D. Kistler and F. Wightman, "Localization using non-individualized head related transfer functions," *Journal of the Acoustical Society of America*, vol. 94, no. 1, pp.111-123, July 1993.

[2] B. Gardner and K. Martin, "HRTF Measurements of a KEMAR Dummy Head Microphone," Available at http://sound.media.mit.edu/resources/KEMAR.html, Accessed June 23, 2009.

[3] A. Oppenheim, and R. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, New Jersey, USA, second edition, 1999.

[4] S. Mehrgardt and V. Mellert, "Transformation characteristics of the external human ear," *Journal of the Acoustical Society of America*, vol. 61, no. 6, pp. 1567-1576, June 1977.

[5] A. Kulkarni, S. Isabelle and H. Colburn, "Sensitivity of Human Subjects to Head-Related Transfer-Function Phase Spectra," *Journal of the Acoustical Society of America*, vol. 105, no. 5, pp. 2821-2840, May 1999.

[6] G. Kuhn, "Model for the interaural time difference in the azimuthal plane," *Journal of the Acoustical Society of America*, vol. 62, no. 1, pp.157-167, July 1977.

[7] S. Busson, R. Nicol and B. Katz, "Subjective investigations of the interaural time difference in the horizontal plane," *AES 118th Convention*, Barcelona, Spain, May 2005.

[8] V. Lazzarini and B. Carty, "New Csound Opcodes for Binaural Processing," *Proc. 6th Linux Audio Conference*, Cologne, Germany, February 2008, pp. 28-35.

[9] B. Carty and V. Lazzarini, "Frequency-domain Interpolation of Empirical HRTF Data," *AES 126th Convention*, Munich, Germany, May 7-10, 2009.

[10] B. Moore, An *Introduction to the Psychology of Hearing*, Elsevier Academic Press, London, UK, fifth edition, 2004.

[11] L. Savioja, J. Huopaniemi, T. Lokki and R. Väänänen, "Creating Interactive Acoustic Environments," *Journal of the Audio Engineering Society*, Vol. 47, No. 9, pp. 675-705, September 1999.

[12] P. Xiang, D. Camargo and M. Puckette, "Experiments on Spatial Gestures in Binaural Sound Display," in *Proc. 11th International Conference on Auditory Display (ICAD 05)*, Limerick, Ireland, July 2005, pp. 1-4.

[13] D. Zotkin, R. Duraiswami and L. Davis, "Rendering Localized Spatial Audio in a Virtual Auditory Space," *IEEE Transactions on Multimedia*, vol. 6, no. 4, pp.553-564, August 2004.

[14] P. Minnaar, J. Plogsties, S. Olesen, F. Christensen and H. Møller, "The Interaural Time Difference in Binaural Synthesis", *AES 108th Convention*, Paris, France, February 2000.

[15] M. Burkhard and R. Sachs, "Anthropometric manikin for acoustic research," *Journal of the Acoustical Society of America*, vol. 58, no. 1, pp.214-222, July 1975.

[16] ITU-R. Recommendation BS.1284, "General methods for the subjective assessment of sound quality," *International Telecommunications Union Radiocommunications Assembly*, 1997.

[17] http://www.csounds.com/manual/html/hrtfmove.html, http://www.csounds.com/manual/html/hrtfmove2.html, http://www.csounds.com/manual/html/hrtfstat.html, Accessed June 23 2009.

HRTFEARLY & HRTFREVERB: FLEXIBLE BINAURAL REVERBERATION PROCESSING

Brian Carty *Victor Lazzarini*

Sound and Digital Music Technology Group,
National University of Ireland, Maynooth,
Co. Kildare,
Ireland

ABSTRACT

A binaural reverberation processor is presented, based on location accurate processing of early reflections and a Feedback Delay Network (FDN) approach to the later diffuse field. Recently developed Head Related Transfer Function (HRTF) dynamic processing algorithms, which have been shown to perform favorably when compared to typically employed methods, are used to allow dynamic direct sources and early reflections. A flexible binaural FDN, which considers interaural coherence, provides an efficient and robust later reverberation model. The overall system is designed to work parametrically, and requires no measured room impulses. This paper introduces the area and gives implementation details of each section of the overall reverberation algorithm from the point of view of recently developed Csound opcodes.

1. INTRODUCTION

In most natural listening circumstances, sound arrives at a listener not only directly from the source, but also after reflecting off obstacles/boundaries (reverberation). Several approaches to reverberation processing have been presented in the literature, varying greatly in computational cost and accuracy (for example, computationally efficient recursive filters [12] to more accurate yet significantly more costly digital waveguide meshes [10]). The impulse response of a room can be split into early reflections and a later, more diffuse reverberant tail. Several artificial reverberation models are based on this decomposition [2, 8, 11], which is also employed here.

The binaural nature of the human hearing system provides our main sound localization cues. Two signals, with possible timing and intensity differences, arrive at the ears and are processed by the brain. HRTFs describe how a sound is altered as it travels from a source at a particular location to the eardrum. HRTF pairs thus inherently include the above interaural differences, as well as other localization cues. The HRTF should thus be considered when endeavoring to recreate artificial reverberation binaurally.

2. HRTF PROCESSING: DYNAMIC TRAJECTORIES

HRTFs can be used to artificially spatialize any source sound to a desired location for headphone listening using convolution. HRTFs are typically measured at discrete points around a listener (the dataset used here is the MIT HRTF dataset [5]). Therefore, to move a source sound from point to point, interpolation is required. Interpolation in the frequency domain is desirable, but poses the difficulty of phase interpolation (phase is a periodic quantity). Decomposing the HRTF into a minimum phase plus linear delay system offers a potential solution to this problem. However, this transformation also potentially introduces inaccuracies [3] at certain locations and involves complex data processing. Novel approaches to the challenge of HRTF interpolation were recently suggested by the authors, which aim to minimize any complex data transformation, preparation or compression, thus allowing empirical data to be used more directly [3, 4].

Briefly, the two approaches suggested both involve interpolating HRTF magnitudes directly and novel approaches to phase interpolation. The first truncates phase, using brief, user definable cross fades to allow for phase changes. The second augments a spherical head model for phase with low frequency scaling, allowing for accurate low frequency interaural phase difference: a psychoacoustically motivated approach. Significantly, the algorithms perform very well in subjective and objective tests. When tested subjectively, the phase truncation and augmented spherical head model both provide a more convincing and artifact free dynamic source trajectory than a minimum phase model (although all perform well) [4]. An anchor condition, with HRTF switching as opposed to smooth interpolation performs less successfully. These results are illustrated in Figure 1. Phase truncation is used in this reverberation application as it is both efficient and performs excellently for dynamic source trajectories.

HRTF processing can offer very convincing results, but has limitations, which should be considered. HRTFs are individual specific, due to the individual nature of outer ear physiology. The binaural nature of HRTFs implies optimal reproduction on headphones. Also, artificial spatialisation can be difficult in a vision centric perceptual system.

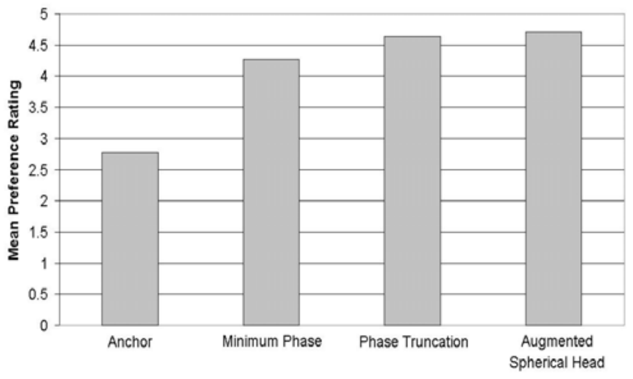


Figure 1. Preference test results for HRTF dynamic source algorithms.

3. EARLY REFLECTIONS

The image model [1] is a geometric model which can be used for early reflection processing. It uses virtual sources in virtual rooms adjacent to the actual room to be modeled. The geometrical paradigm is illustrated in Figure 2. Each virtual source can be spatialized, filtered, delayed and attenuated accordingly, aiding with the perception of the listening environment.

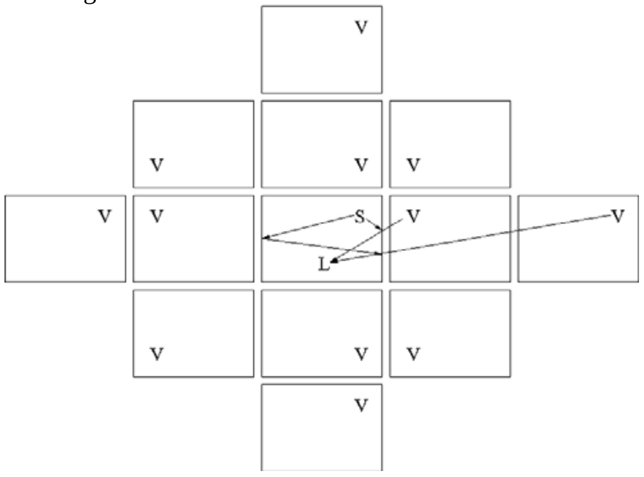


Figure 2. The image model for early reflections (2 dimensional, 2nd order), S represents the actual source, L the listener and V virtual sources.

The implementation of the early reflections processor as the opcode **`hrtfearly`** for the computer music language Csound uses phase truncation HRTF processing to spatialize and move the direct sound source and early reflections in accordance with the image model. HRTF processing can be costly, particularly when compared to the more efficient later diffuse field algorithm. Therefore, the user can choose the order of the early reflections. Order 0 processes just the direct source, order 1 the first reflections, etc. It may be desirable to simplify early reflection spatial accuracy if higher order/more efficient processing is required [8]. Three dimensional processing is also optional, whereby reflections from the floor and the ceiling are considered. In a natural environment, a user may move his/her head, thus reorienting sound sources. This is also possible in the presented opcode. Source and listener location are also dynamic parameters.

The room parameters define the nature of the reverberant sound. A value for the low and high frequency absorption coefficients of each surface of the (presumed rectangular) room are used to calculate the cutoff frequency of a simple low pass filter which models the surface's response. A three band equalizer is also offered for each surface to allow for implementation of multiband reflective surfaces. Source and reflection location are dealt with using HRTFs. Distance processing is implemented using an interpolated delay line (as well as attenuation), which also provides any appropriate Doppler Effect. As well as the processed input, the opcode outputs the rooms mean free path and low and high frequency reverberation times, based on the Norris-Eyring reverberation equation. These outputs can then be used in the diffuse field binaural reverberation opcode.

4. DIFUSE FIELD

After the period of discrete early reflections in the evolution of the reverberation, sound begins to arrive from all around a listener, in a diffuse manner. Therefore, spatially accurate single reflections are no longer required. Without the necessity of considering each reflection individually, a much more efficient late reverberation can be implemented. The opcode **`hrtfreverb`** offers efficient binaural reverberation processing. It can be used with **`hrtfearly`** to provide spatially accurate source location as well as reverberation, or independently as an efficient, more general binaural reverberator.

A reverberant tail can be captured/modeled and convolved with the input sound to impose the reverberant characteristics of the space onto the input. However, convolution, although optimizable, can introduce processing delays and be computationally expensive for long impulses. A feedback system is perhaps a more appropriate solution. FDNs offer a subtle yet effective approach. Jot's comprehensive treatment of the topic covers the scenario of modeling a measured impulse response [6] and using parametric inputs [7]. Jot also discusses binaural output of the results.

Recently, the binaural element of Jot's measured impulse model was improved by considering interaural coherence [9]. The current model furthers this work by considering the parametric scenario, as well as independent early reflection processing.

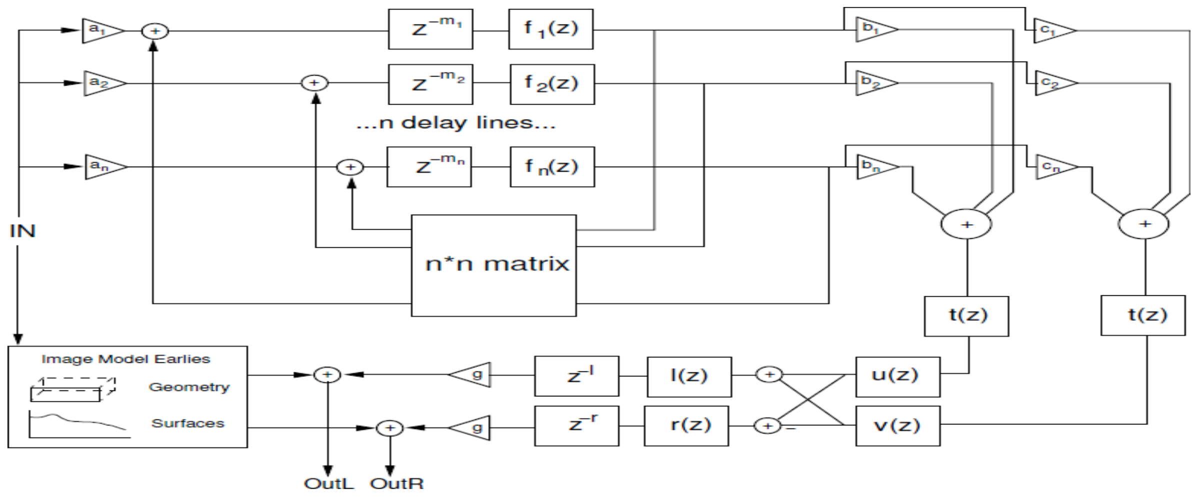


Figure 3: Schematic of Overall Process: Input is processed by the **`hrtfearly`** opcode, which outputs details on frequency dependent reverberation time, which are used by the **`hrtfreverb`** opcode (FDN adapted from [9]).

Briefly, the Jot FDN works using a number of mutually prime delay lines, with frequency dependent feedback, in accordance with the desired frequency dependent reverberation time. The feedback loop includes a matrix which increases the density of the diffuse tail. In **`hrtfreverb`**, the frequency dependent reverberation time can be chosen by the user, or values calculated by an instance of the **`hrtfearly`** opcode, based on the inputted room geometry and surface characteristics, can be used.

The suggested reverberation frequency density of 0.15 modes per Hz [12] is achieved using a sufficiently long overall delay (the sum of the delay lines used). The mean free path of the environment provides a suitable average delay line length (it is also important that the shortest delay line is shorter than the overall diffuse onset delay for real time processing). A flexible number of delay lines is thus required for various reverberation times. 6, 12 or 24 are used in **`hrtfreverb`**, based on the input reverberation properties. Frequency dependent reverberation times are achieved using simple first order Infinite Impulse Response (IIR) filters [7]. These filters ($f_n(z)$ in Figure 3, which represents the overall process) also reduce the spectral energy for low reverberation times, which is compensated for using a tone correction filter (t(z)). Two outputs of the FDN are taken, which should be uncorrelated (vectors **b** and **c** ensure this). Coherence of the HRTF dataset is calculated using equation 1 [9]. The coherence matching filters u(ω) and v(ω), which provide accurate interaural coherence, are defined by equations 2 and 3 [9].

$$\phi(\omega) = \frac{\sum_{i=1}^{N} L_i(\omega) R_i^*(\omega)}{\sqrt{\sum_{i=1}^{N} |L_i(\omega)|^2 \sum_{i=1}^{N} |R_i(\omega)|^2}} \tag{1}$$

where $L_i(\omega)$ and $R_i(\omega)$ represent the ith left and right HRTF in a dataset of *N*.

$$u(\omega) = \sqrt{\frac{1+\phi(\omega)}{2}} \tag{2}$$

$$v(\omega) = \sqrt{\frac{1-\phi(\omega)}{2}} \tag{3}$$

Left and right HRTF dataset average power filters (l(z) and r(z)) are then used to make the output binaural. The opcode outputs the appropriate delay time (according to the mean free path, processing order and inherent delay) for the late reverberation, as well as the processed audio. This delay, followed by a scaling factor, completes the process, which is illustrated in Figure 3.

5. IMPLEMENTATION

The opcodes are designed to balance efficiency, accuracy and usability. They must also be robust to a large number of scenarios. As discussed, **`hrtfreverb`** is designed to be used with **`hrtfearly`**, or equally, as an independent, efficient binaural reverberator. If a number of sources exist in the same environmental infrastructure, they can be summed and processed with the same instance of **`hrtfreverb`**. Individual trajectories/locations can be dealt with using multiple instances of **`hrtfearly`**. For ease of use, a number of default rooms are available, with standard surfaces. Processing at various sampling rates is also offered. Csound provides suitable opcodes to implement the appropriate **`hrtfreverb`** delay (calculated in a straightforward manner, appropriate to the parametric nature of the processing), as well as other opcodes which add a further dimension of user control (e.g. further low pass filtering).

As mentioned above, HRTF processing can be computationally costly, particularly when processing multiple sources with multiple reflections each. Therefore, several code optimizations have been implemented, for example, optimal real FFT processing is used, and interpolation is only performed when relative source location changes. Real time performance in most typical scenarios is therefore achievable.

Overall, it is hoped that the opcodes offer an intuitive, flexible approach to binaural reverberation. The desirable balance between ease of use (default values, the standalone nature of the intuitive **`hrtfreverb`** opcode, etc.) and advanced processing (expert optional parameters, giving a fine degree of control) allow for both immediate and detailed use.

6. APPLICATIONS

The most obvious application of the HRTF reverberation opcodes is the binaural spatialisation of audio with accurate localization and environmental processing. More specifically, they allow for binaural multi-channel audition (the opcodes were developed with this application in mind). As any source can be spatialized to any location in any desired room, it is possible to place virtual loudspeakers in virtual listening rooms. The phase truncation interpolation algorithm allows a listener to move around this virtual environment without jeopardizing audio quality. Therefore, multi-channel algorithms can be auditioned on headphones. A composer/sound designer can thus work with a multi-channel setup using only headphones.

Using the infrastructure available in Csound, the application under development will allow a user to setup a room with a desired number of virtual loudspeakers. Each of these virtual loudspeakers is then fed with an appropriate audio stream (for example, the outputs of an Ambisonic/VBAP/Wave Field Synthesis mix). The overall output can then be auditioned. The user can also move through the listening space, for example to investigate any sweet spot issues. This approach is meant as an audition tool, as opposed to offering a general, optimized binaural solution.

Compositionally, the reverberation tools can also be useful. Indeed, their development was part motivated by the need for an accurate reverberation when composing using HRTFs. Interestingly, the parametric approach to the algorithms involved allows non natural scenarios, within the boundaries of stability, which may provide appealing compositional material. For example, massive rooms with very reflective walls, non natural levels of late reverberation and distant sources.

7. CONCLUSIONS

New tools for binaural reverberation are presented. The algorithms developed constitute a consolidation of classic and more recent approaches to reverberation. Several updates are suggested to allow for a flexible implementation, including recently developed HRTF interpolation algorithms and more accurate early reflections. The resulting algorithms are presented as efficient Csound opcodes, allowing both immediate application and a fine degree of control.

8. ACKNOWLEDGEMENTS

This Research is supported by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan and NUI Maynooth.

9. REFERENCES

[1] Allen, J. and Berkley, D. “Image method for efficiently simulating small-room acoustics,” *JASA*, 65(4), 1979, pp. 943-950.

[2] Begault, D. *3-D Sound for Virtual Reality and Multimedia*, Maryland: NASA, 2000.

[3] Carty, B. and Lazzarini, V. “Binaural HRTF Based Spatialisation: New Approaches and Implementation,” *Proc. DAFx*, Como, Italy, September 2009, pp. 49-54.

[4] Carty, B. and Lazzarini, V. “Frequency-domain Interpolation of Empirical HRTF Data,” *126th AES Convention*, Munich, Germany, May 2009.

[5] Gardner, B. and Martin, K. “HRTF Measurements of a KEMAR Dummy Head Microphone,” available at http://sound.media.mit.edu/resources/KEMAR.html, accessed December 22, 2009.

[6] Jot, J. “An Analysis/Synthesis Approach to Real-time Artificial Reverberation,” *Proc. ICASSP*, 2, March 1992, pp. 221-224.

[7] Jot, J. “Digital delay networks for designing artificial reverberators,” *90th AES Convention*, Paris, France, February 1991.

[8] Jot, J., Larcher, V. and Warusfel, O. “Digital Signal Processing Issues in the Context of Binaural and Transaural Stereophony,” *98th AES Convention*, Paris, France, February 1995.

[9] Menzer, F. and Faller, C. “Binaural reverberation using a modified Jot reverberator with frequency-dependent interaural coherence matching,” *126th AES Convention*, Munich, Germany, May 2009.

[10] Murphy, D., Beeson, M., Shelley, S. and Moore, A. “Hybrid Room Implulse Response Synthesis in Digital Waveguide Mesh Based Room Acoustics Simulation,” *Proc. DAFx*, Espoo, Finland, 2008, pp. 129-136.

[11] Savioja, L., Huopaniemi, J., Lokki, T. and Väänänen, R. “Creating Interactive Acoustic Environments,” *JAES*, 47(9), 1999, pp. 675-705.

[12] Schroeder, M. “Natural Sounding Artificial Reverberation,” *JAES*, 10(3), 1962, pp. 219-223.

MULTIBIN: A BINAURAL AUDITION TOOL

Brian Carty *Victor Lazzarini*

Sound and Digital Music Technology Group,
National University of Ireland, Maynooth
Co. Kildare, Ireland
brian.m.carty@nuim.ie victor.lazzarini@nuim.ie

ABSTRACT

MultiBin is a new tool for binaural audition of multiple sound sources in a user definable environment. Although designed to be flexible in its application, its primary function is to provide dynamic multi-channel binaural simulation. It is built upon 2 new Csound binaural reverberation opcodes. An early reflection opcode, based on an image source method and a HRTF interpolation algorithm previously introduced by the authors provides dynamic source and listener location. This is complemented by a later reverberation opcode which provides a diffuse reverb based on a parametric FDN model which considers interaural coherence.

1. INTRODUCTION

Head Related Transfer Functions (HRTFs) are essentially frequency domain functions which describe how a sound is altered from a particular source location to the ear [1]. Pairs of HRTFs inherently consider sound localisation cues such as interaural differences and spectral transformations. HRTFs are typically used in binaural processing tools, and can be used to artificially spatialise sound in a virtual listening environment [1, 2].

Typically, environmental processing is desirable in this scenario. Reverberation thus needs to be considered [2, 3].

This paper discusses issues involved with the development of a flexible binaural audition tool, MultiBin. Each key aspect of the application is dealt with in turn. The first challenge is providing smooth, artefact free source trajectories: dynamic HRTF processing. Binaural early reflections will then be discussed in the context of HRTF spatialisation. A later reverberant tail which considers interaural coherence completes the environmental processing. Finally, an intuitive Python GUI is outlined.

2. CSOUND OPCODES

MultiBin uses the Csound API to allow Python to send dynamic, user generated information to an instance of Csound from the host application [4]. Csound thus deals with the low level DSP required to dynamically represent sound sources and a listener in a user defined sonic environment. The relevant Csound opcodes and their background will now be discussed.

2.1 HRTF Processing

Two novel approaches to HRTF interpolation and dynamic source trajectory processing were recently introduced by the authors. The approaches, as well as background and validation of the algorithms are discussed in previous publications [5, 6]. Briefly, the algorithms aim to provide accurate, efficient HRTF processing while minimising data analysis, compression or transformation. The algorithms developed are realised in the Csound opcodes `hrtfmove`, `hrtfmove2` and `hrtfstat` [4].

`hrtfmove` employs magnitude interpolation and phase truncation to allow dynamic spatialisation of input audio using overlap add convolution processing. Alternatively, a more traditional minimum phase plus delay process is implemented by this opcode. An optional flag allows the user to switch processing modes.

`hrtfmove2` takes a more functional approach, augmenting a spherical head model for interaural phase with low frequency scaling factors based on empirical HRTF data. This psychoacoustically motivated approach models phase in accordance with the limitations and frequency dependant nature of the auditory localisation system [7, 8]. `hrtfstat` implements the same algorithm, but exploits potential optimisations for static source processing.

The algorithms (Minimum Phase based processing, Phase Truncation, the Augmented Spherical Head and an anchor condition with no interpolation) were tested subjectively with regard to ability to provide smooth, artefact free source trajectories. The novel algorithms performed extremely well. Interestingly, the results also highlight both the need for interpolation (the anchor condition performed in the poor-fair range) and the good performance but potential problems with minimum phase processing [7].

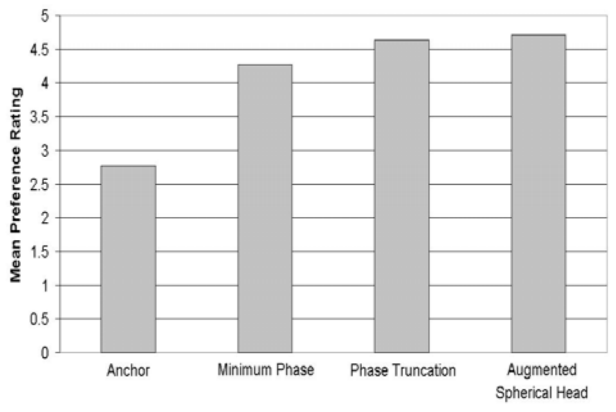


Figure 1: *Preference Test Results.*

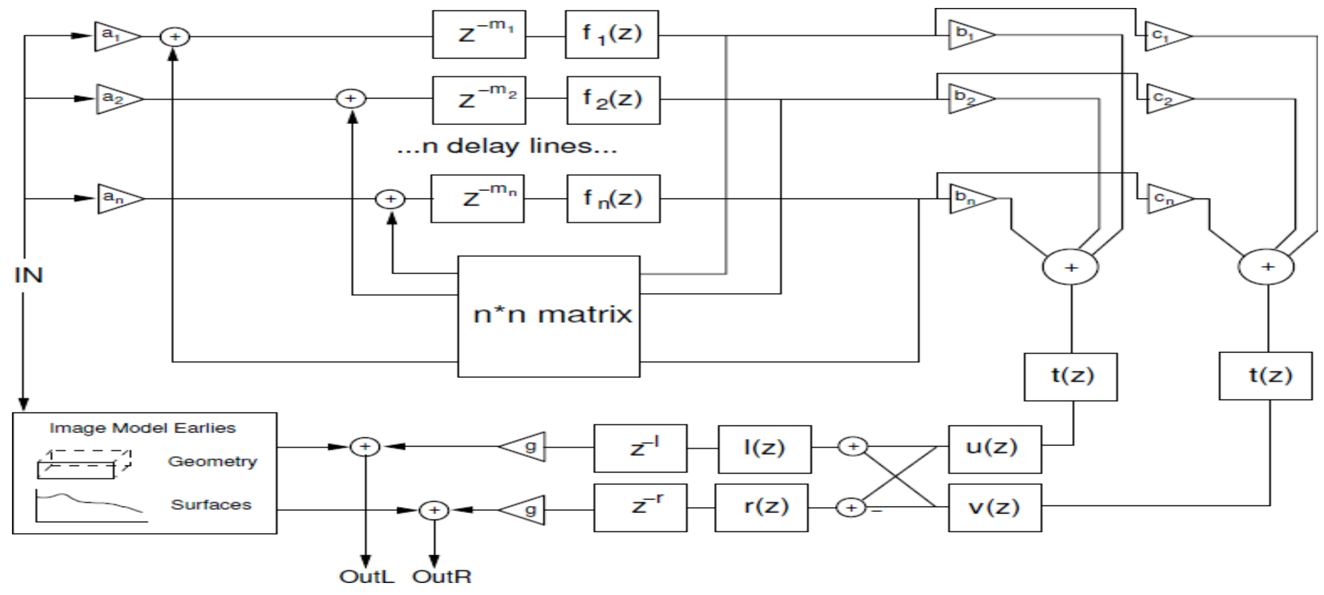


Figure 2: *Schematic of the Reverb Model.*

2.2 Early Reflections

MultiBin does not use the HRTF opcodes directly, but re-implements the Phase Truncation algorithm for early reflection processing. The success of the algorithm in subjective tests and its efficiency make it a suitable candidate. A brief code example is perhaps an appropriate introduction to the opcode.

```
aearlyl,  aearlyr,  ilow,  ihigh,  imfp
hrtfearly ain, srcx, srcy, srcz, lx, ly, lz,
"datal.raw", "datar.raw", 1
```

The opcode, `hrtfearly`, outputs the left and right processed audio, a low and high frequency reverb time, calculated using the Norris-Eyring reverb formula, and the mean free path for the room in question. The latter 3 values are intended to be used as inputs to the later reverberant field opcode, discussed below. Dynamic, control rate source and listener x, y and z geometric location values are the main inputs, following the mono audio input. Left and right HRTF data files are the next arguments. Finally, a default room can be chosen: small, medium or large. An image source model [9, 10] is used to process the early reflections dynamically.

Optional parameters for more advanced use are also available. These include the Phase Truncation fade length [5, 6], the sampling rate, the order of reflection processing, whether floor and ceiling reflections are considered, a dynamic head rotation value, and room parameters. The size of the room, as well as the high and low frequency absorption coefficients and parameters of 3 band pass filters for each surface can be set.

Processing a number of sources, each with a (user definable) number of reflections can quickly become computationally costly, so optimisation is crucial. Interpolation is only performed if source orientation with respect to the listener changes. . Memory allocation, Fourier Transform Processing and dynamic source trajectory processing are also optimised.

2.3 Later Reverb

`hrtfreverb` is a later, reverberant field opcode, which employs a dynamic Feedback Delay Network (FDN) and various filters to process the output binaurally [10]. The opcode builds on the interaural coherence [11] addition to the Jot FDN [12] by considering the parametric scenario, as opposed to measured impulses. It also considers flexible early reflection processing.

The FDN is illustrated in Figure 2, as well as frequency dependent reverb filters ($f_n(z)$), compensating tone correction filters (t(z)), vectors to ensure 2 uncorrelated output (**b** and **c**), coherence matching filters (u(z), v(z)), average HRTF filters (l(z), r(z)) appropriate delay and gain factors for the later reverberant tail and finally early reflection processing (discussed in more detail in [10]).

Once again, the opcode can be initialized and used in a flexible manner:

```
alatel, alater, idel hrtfreverb ain, ilow,
ihigh, "datal.raw", "datar.raw"
```

Outputs are the left and right channels of the binaural reverberant tail and its appropriate delay time. The low and high frequency reverb time [12], HRTF data files and audio input are the only required inputs. Sampling rate, mean free path and order of early reflection processing are optional inputs. The latter 2 arguments are used to derive the appropriate delay time for the late reverberant field.

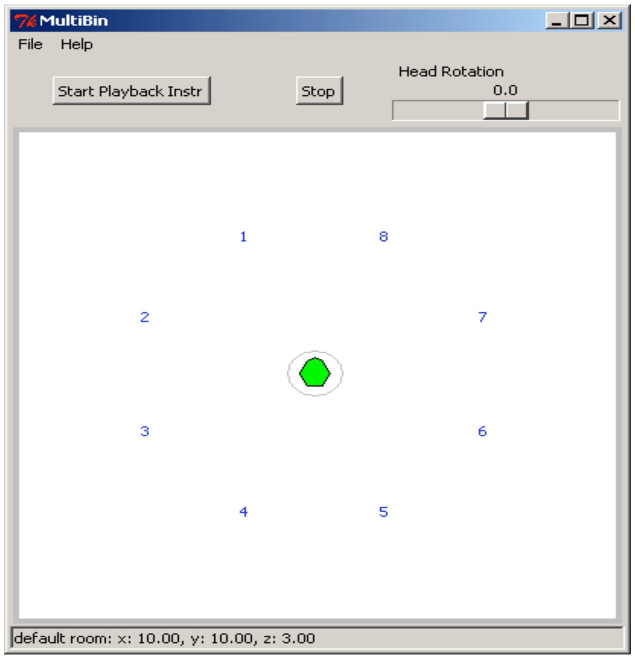
3. MULTIBIN

The MultiBin application is built on the above binaural reverberation opcodes. Essentially, the goal of the application is to allow flexible virtual spatial environments, with a particular focus on virtual multi-channel (sources constituting loudspeakers). Figure 3 illustrates a typical instance of MultiBin. Upon running the application, the canvas [13] shows a centred listener with no sources in their sonic environment. A default medium room is setup. Users can add sources to the canvas in a simple and flexible way. Each source is numbered, which links it to a specific Csound channel. Sources can be added and removed using simple menu options. Therefore, the user may setup a flexible virtual environment.

Sources can be moved around the environment by selecting and dragging. The HRTF interpolation algorithms discussed above allow for real time realisation of these movements. The listener may also move using similar intuitive user control. Head rotation is also implemented. Playback of the Csound controlled source material is also a simple and intuitive process (play and stop buttons). The File menu allows a user to setup an 8 channel ambisonic room, as well as a similar VBAP setup. The parameters of these loudspeaker setups are based on the existing Csound opcodes for ambisonic and VBAP processing. All sources can also be easily cleared.

A user may therefore audition a multi-channel file in headphones. The approach taken here differs from previous approaches that use sweet spot virtual multi-channel binaural processing [14]. The user may move out of the sweet spot to audition off centre listening. Perhaps a loudspeaker in a specific room cannot be physically placed at the optimum location. The loudspeaker can be dynamically moved in real time to audition any potential problems. Any multi-channel algorithm can thus be auditioned in a dynamic fashion. The ambisonic and VBAP opcodes in Csound allow for convenient preparation of appropriate source material, motivating their inclusion as defaults. Equally, however, other setups, such as simple Wave Field Synthesis arrays could be auditioned.

From a design point of view, the system allows optimal dynamic multi-channel audition, but also caters for other virtual spatial applications. The ease with which sources can be added, removed and dragged around the canvas creates a flexible and creative workspace for spatial audio. Implementation of the application will now be discussed in more detail.



default room: x: 10.00, y: 10.00, z: 3.00

Figure 3: *A Typical Instance of MultiBin.*

3.1 Implementation Detail

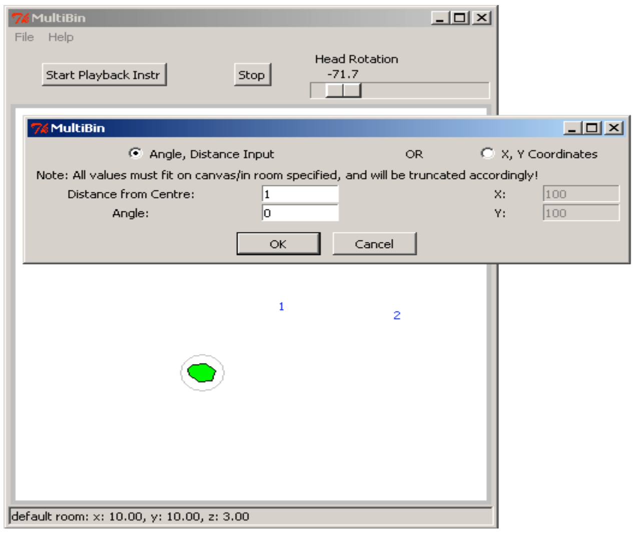
The Tkinter Python module [13], which uses the Tk GUI toolkit is the cross platform tool used to create an appropriate GUI for the MultiBin application. A simple Csound file is controlled by sending user triggered messages from the GUI. This Csound file consists of a playback instrument, which essentially plays back the source audio. Csound offers several possible options to do this, including playing back sound samples stored in tables, or directly reading multi-channel files. Each stream of audio is played back on a numerically labelled channel. A global parameter reading instrument reads head position in the GUI.

Reverberation processing is then dealt with. An early reflection instrument is assigned to each source, reading the dynamic source trajectory, and taking parameters from user inputs. When the user adds a new source, it appears on the canvas of the GUI as a number which relates to the channel it is reading from. Therefore, the user has direct control over the audio linked to each source. Finally, a late reverb instrument processes all sources using the `hrtfreverb` opcode. Reverb time and room outputs of `hrtfearly` can be used as inputs to `hrtfreverb`.

The Python code defines the GUI, which will now be discussed. The main interactive element of the application is the canvas widget, which allows user input and control. The application is designed as a class, with methods for movement of objects on the canvas and other control functions. Control of items on the canvas is maintained using the Tkinter item methods. The application class constructor initiates an instance of Csound, whose parameters are updated by the user. For example, if a source is moved, updated x and y values are sent to Csound, which updates the location of the source relative to the listener at the control rate. The constructor also sets up the menus and GUI, initialises variables and binds class methods/callbacks to user operations/events, such as mouse button clicks.

A number of methods are used to allow for adding sources in a well defined manner. A generic function adds an item to the canvas, increments the control variable which keeps track of the number of active sources and turns on an instance of the `hrtfearly` instrument. A second method calls this generic method, and deals with the user defined location of the source, using data from the dialog window illustrated in figure 4. This method also ensures the source location is legal (inside the defined room). Location data is stored in an instance of another class, which defines the location dialog window and data. As can be seen in figure 4, direct pixel location can be entered, or a polar approach can be taken, motivated by typical multi-channel setups. An angle and distance from room centre can be entered to define source location. The tkSimpleDialog module is used to create the source location class/dialog window.

Figure 5 shows the 'new scene' dialog window, similarly realised as a class developed using the tkSimpleDialog module. Users can either simply define the room size, or choose to enable complex parameters, and enter surface parameters for high and low frequency reverb time and wall response band pass filter gain factors. The canvas will always be 400 pixels across, with room geometry ratios dictating the height. The size of the room dictates the size of the grey circle centred at the listener position. Inside this circle, HRTF processing is not performed, as near field HRTF modelling is not considered [15].



Head Rotation -71.7

(●) Angle, Distance Input OR ( ) X, Y Coordinates

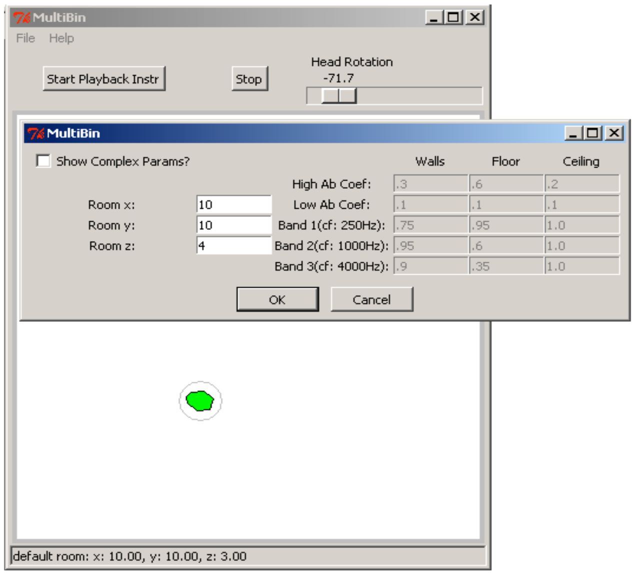
Note: All values must fit on canvas/in room specified, and will be truncated accordingly!

Distance from Centre: 1 X: 100

Angle: 0 Y: 100

default room: x: 10.00, y: 10.00, z: 3.00

Figure 4: *New Source Dialog.*



Head Rotation -71.7

☐ Show Complex Params?

Room x: 10

Room y: 10

Room z: 4

	Walls	Floor	Ceiling
High Ab Coef:	.3	.6	.2
Low Ab Coef:	.1	.1	.1
Band 1(cf: 250Hz):	.75	.95	1.0
Band 2(cf: 1000Hz):	.95	.6	1.0
Band 3(cf: 4000Hz):	.9	.35	1.0

default room: x: 10.00, y: 10.00, z: 3.00

Figure 5: *New Scene Dialog.*

4. DISCUSSION & APPLICATIONS

Multibin is designed to be a flexible binaural tool, allowing dynamic source and listener behaviour in user definable virtual environments. Its primary intention as a binaural multi-channel audition application allows flexible listener scenarios. Unlike sweet spot multi-channel binaural approaches [14], off centre listening is purposefully allowed and indeed intended, as the listener may move around the sonic environment. Flexible loudspeaker setups are also possible, allowing real world audition. It is also important to highlight that MultiBin does not aim to optimise multi-channel binaural processing [16]; it purposefully allows complete user control.

Standard binaural limitations apply, including HRTF individualisation. Real time performance is prioritised over additional processing such as source directivity.

5. CONCLUSIONS

MultiBin is a flexible tool developed using Python and the Csound API. Csound binaural reverberation opcodes are utilized, which are based on previous HRTF interpolation research by the authors. MultiBin allows dynamic control over user definable virtual environments, with specific application to virtual multi-channel.

6. ACKNOWLEDGMENTS

This research is supported by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan and NUI Maynooth.

7. REFERENCES

[1] Begault, D., *3-D Sound for Virtual Reality and Multimedia*, Maryland: NASA, 2000.

[2] Jot, J., Larcher, V. and Warusfel, O. “Digital Signal Processing Issues in the Context of Binaural and Transaural Stereophony,” *98th AES Convention*, Paris, France, 1995.

[3] Savioja, L., Huopaniemi, J., Lokki, T. and Väänänen, R. “Creating Interactive Acoustic Environments,” *JAES*, 47(9), pp. 675-705, 1999.

[4] http://www.csounds.com.

[5] Carty, B. and Lazzarini, V. “Binaural HRTF Based Spatialisation: New Approaches and Implementation,” *Proc. DAFx*, Como, Italy, pp. 49-54, September 2009.

[6] Carty, B. and Lazzarini, V. “Frequency-domain Interpolation of Empirical HRTF Data,” *126th AES Convention*, Munich, Germany, May 2009.

[7] Kulkarni, A., Isabelle, S. and Colburn, H., “Sensitivity of Human Subjects to Head-Related Transfer-Function Phase Spectra,” *JASA*, 105(5), pp. 2821-2840, May 1999.

[8] G. Kuhn, “Model for the interaural time difference in the azimuthal plane,” *JASA*, 62(1), pp.157-167, July 1977.

[9] Allen, J. and Berkley, D. “Image method for efficiently simulating small-room acoustics,” *JASA*, 65(4), pp. 943-950, 1979.

[10] Carty, B. and Lazzarini, V. “hrtfearly & hrtfreverb: Flexible Binaural Reverberation Processing,” *ICMC*, New York, USA, June 2010.

[11] Menzer, F. and Faller, C. “Binaural reverberation using a modified Jot reverberator with frequency-dependent interaural coherence matching,” *126th AES Convention*, Munich, Germany, May 2009.

[12] Jot, J. “Digital delay networks for designing artificial reverberators,” *90th AES Convention*, Paris, France, 1991.

[13]http://www.pythonware.com/library/tkinter/introduction/index.htm

[14] Noisternig, Musil, Sontacchi and Holdrich. 3D Binaural Sound Reproduction using a Virtual Ambisonic Approach. *IEEE Symposium on Virtual Environments*, pp. 174-178, 2003.

[15] Duda, R. and Martens, W., “Range dependence of the response of a spherical head model,” *JASA*, 104(5), pp. 3048-3058, 1998.

[16] Goodwin, M. and Jot, J., “Binaural 3-D audio rendering based on spatial audio scene coding,” *123rd AES Convention*, New York, USA, October 2007.

Appendix 1: Command-line

1.1 defs.h

```
/*
Brian Carty PhD Code 2010
Chapter 4,
defs.h
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sndfile.h>
#include <fftw3.h>

/* correct length for mono impulse */
#define irlength 128
/* padded impulse */
#define irpadlength 256
/* convolution overlap */
#define overlapsize 127
/* maxumum points in a trajectory */
#define maxbrkpts 101

void bkpt(int *pers, double *els, double *angs, int *noofpoints, int maxpts);
```

1.2 datapreparation.cpp

```
/*
Brian Carty PhD Code 2010
Chapter 4,
datapreparation.cpp
*/

#include "defs.h"
#define SQUARE(X) (X)*(X)

int main()
{
  /* setup variables */
  /* min elev, angle, increment, iterators */
  int el = -40, az = 0, inc, i, j, k;
  /* input from HRTF file */
  double input[2 * irlength];
  /* separate input into left and right */
  double inl[irlength], inr[irlength], fftl[irlength],
         fftr[irlength];
  /* file pointers */
  FILE *foutl, *foutr;
  /* fft plans */
  fftw_plan forwardl, forwardr;
  /* strings for filename */
  char filename[14];
  char hrtffile[22];
  /* file in pointer */
  SNDFILE *finhrtf;
  /* file info */
  SF_INFO *psfinfohrtf;
  /* memory for file info */
  psfinfohrtf = new SF_INFO;

  /* setup fft plans (see fftw documentation) */
  forwardl = fftw_plan_r2r_1d(irlength, inl, fftl, FFTW_R2HC,
          FFTW_ESTIMATE);
  forwardr = fftw_plan_r2r_1d(irlength, inr, fftr, FFTW_R2HC,
         FFTW_ESTIMATE);

  /* open outfiles for writing */
  foutl = fopen("datal.raw", "wb");
  foutr = fopen("datar.raw", "wb");

  /* loop for 368 files */
  for(j = 0; j < 368; j++)
  {
    /* prep for file open string */
    strcpy(hrtffile,"diffuse/");

    /* prep file names */
    if(az < 10)
      sprintf(filename, "H%de00%da.wav", el, az);
    else if(az >= 10 && az < 100)
      sprintf(filename, "H%de0%da.wav", el, az);
    else if(az >= 100)
      sprintf(filename, "H%de%da.wav", el, az);

    /* sort out incrementation based on elev */
    if(el == -40)
    {
      if(inc != 6 || j % 7 == 0)
        inc = 6;
      else inc = 7;
    }
    else if(el == -30 || el == 30)
      inc = 6;
```

```c
else if(el == -20 || el == -10 || el == 0 || el == 10 || el == 20)
  inc = 5;
else if(el == 40)
{
  if(inc != 6 || (j - 276) % 7 == 0)
    inc = 6;
  else inc = 7;
}
else if(el == 50)
  inc = 8;
else if(el == 60)
  inc = 10;
else if(el == 70)
  inc = 15;
else if(el == 80)
  inc = 30;
else if(el == 90)
  inc = 0;

/* put together for full name */
strcat(hrtffile, filename);

/* open appropriate file */
if(!(finhrtf = sf_open(hrtffile, SFM_READ, psfinfohrtf)))
{
  printf("error opening file\n");
  exit(1);
}

/* read in file */
sf_readf_double(finhrtf, input, irlength);
/* close file */
sf_close(finhrtf);

/* put (double: -1.0 to +1.0) input into seperate left and right buffers, scale a
   little */
for(i = 0; i < irlength; i++)
{
  inl[i] = input[2 * i] * .65;
  inr[i] = input[(2 * i) + 1] * .65;
}

/* fft */
fftw_execute(forwardl);
fftw_execute(forwardr);

/* 0Hz and nyq */
inl[0] = fftl[0];
inl[1] = fftl[irlength / 2];
inr[0] = fftr[0];
inr[1] = fftr[irlength / 2];

/* mag/phase: polar */
for(i = 2, k = 1; i < irlength; k++, i += 2)
{
  inl[i] = sqrt(SQUARE(fftl[k]) + SQUARE(fftl[irlength – k]));
  inl[i+1] = atan2(fftl[irlength-k],fftl[k]);
  inr[i] = sqrt(SQUARE(fftr[k]) + SQUARE(fftr[irlength – k]));
  inr[i+1] = atan2(fftr[irlength-k],fftr[k]);
}

/* write outputs, one by one, to large spectral file */
fwrite(inl,sizeof(double), irlength, foutl);
fwrite(inr,sizeof(double), irlength, foutr);

/* incrementation */
az = az + inc;

if(j == 28 || j == 59 || j == 96 || j == 133 || j == 170 || j == 207 || j == 244 ||
   j == 275 || j == 304 || j == 327 || j == 346 || j == 359 || j == 366)
{
      /* change elevation,reset variables */
      el = el + 10;
      az = 0;
      inc = 0;
    }
  }
  /* clear memory, close files */
  delete psfinfohrtf;
  fclose(foutl);
  fclose(foutr);
  fftw_destroy_plan(forwardl);
  fftw_destroy_plan(forwardr);

  return 0;
}
```

1.3 binauralmover.cpp

```
/*
binauralmover.cpp:
Copyright (C) Brian Carty 2009
Binaural sound source movement using magnitude interpolation and phase truncation: main
program
See license.txt for a disclaimer of all warranties and licensing information
*/

#include "defs.h"

int main()
{
  /* MIT Kemar info. */
  int elevationarray[14] = {56, 60, 72, 72, 72, 72, 72, 60, 56, 45, 36, 24, 12, 1};
  int minelev = -40, elevincrement = 10;
  /* iterators */
  int i, j;

  /* data file pointers */
  FILE *hrtfleft, *hrtfright;

  /* arrays to store addresses of where all left and right hrtfs are stored: arrays of
     pointers to double. */
  double *hrtfarrayl[14][37], *hrtfarrayr[14][37];
  /* pointers to read arrays */
  double *hrtfpl, *hrtfpr;

  /* declarations for movement */
  int countbkp = 0;
  int percentages[maxbrkpts];
  double elevs[maxbrkpts], angles[maxbrkpts], elev, angle;
  int k = 0;
  int x,start = 0,sum = 0;

  /* crossfade preparation and checks */
  double elevindexstore, angleindexlowstore, angleindexhighstore;
  int elevindex, angleindex, oldelevindex = -1, oldangleindex = -1;
  int fade, fadebuffer;
  int crossfade, crossout = 0, cross = 0, l = 0;

  /* interpolation variable declaration */
  int elevindexlow, elevindexhigh, angleindex1, angleindex2, angleindex3, angleindex4;
  double elevindexhighper, angleindex2per, angleindex4per;
  double magllow, magrlow, maglhigh, magrhigh, magl, magr, phasel, phaser;
  double lowl1[irlength], lowr1[irlength], lowl2[irlength], lowr2[irlength];
  double highl1[irlength], highr1[irlength], highl2[irlength], highr2[irlength];
  double hrtflinterp[irlength], hrtfrinterp[irlength], hrtfltd[irlength],
         hrtfrtd[irlength];
  double hrtflpadtd[irpadlength], hrtfrpadtd[irpadlength], hrtflpadspec[irpadlength],
         hrtfrpadspec[irpadlength];

  /* convolution/in/output buffers */
  double inbuf[irpadlength], inspec[irpadlength];
  double outlspec[irpadlength], outrspec[irpadlength], outl[irpadlength] = {0.0},
         outr[irpadlength] = {0.0};
  double overlapl[overlapsize], overlapr[overlapsize];
  sf_count_t count=0;
  double lrout[2 * irlength];

  /* various buffers for fades */
  double currentphasel[irlength], currentphaser[irlength];
  double hrtflpadspecold[irpadlength], hrtfrpadspecold[irpadlength];
  double outlspecold[irpadlength], outrspecold[irpadlength];
  double overlaplold[overlapsize], overlaprold[overlapsize];
  double outlold[irpadlength] = {0.0}, outrold[irpadlength] = {0.0};
```

```
/* file pointers, file info */
char filename[100];
SNDFILE *fin, *fout;
SF_INFO *psfinfoout, *psfinfoin;

/* fftw plans */
fftw_plan invhrtfl, invhrtfr, forhrtflpad, forhrtfrpad, forin;
fftw_plan invoutl, invoutr, invoutlold, invoutrold;

invhrtfl = fftw_plan_r2r_1d(irlength, hrtflinterp, hrtfltd, FFTW_HC2R, FFTW_ESTIMATE);
invhrtfr = fftw_plan_r2r_1d(irlength, hrtfrinterp, hrtfrtd, FFTW_HC2R, FFTW_ESTIMATE);
forhrtflpad = fftw_plan_r2r_1d(irpadlength, hrtflpadtd, hrtflpadspec, FFTW_R2HC,
                               FFTW_ESTIMATE);
forhrtfrpad = fftw_plan_r2r_1d(irpadlength, hrtfrpadtd, hrtfrpadspec, FFTW_R2HC,
                               FFTW_ESTIMATE);
forin = fftw_plan_r2r_1d(irpadlength, inbuf, inspec, FFTW_R2HC, FFTW_ESTIMATE);
invoutl = fftw_plan_r2r_1d(irpadlength, outlspec, outl, FFTW_HC2R, FFTW_ESTIMATE);
invoutr = fftw_plan_r2r_1d(irpadlength, outrspec, outr, FFTW_HC2R, FFTW_ESTIMATE);
invoutlold = fftw_plan_r2r_1d(irpadlength, outlspecold, outlold, FFTW_HC2R,
                              FFTW_ESTIMATE);
invoutrold = fftw_plan_r2r_1d(irpadlength, outrspecold, outrold, FFTW_HC2R,
                              FFTW_ESTIMATE);

/* memory for SF_INFO structures */
psfinfoin = new SF_INFO;
psfinfoout = new SF_INFO;

printf("\nBinaural Processing Application\n\n");

/* setup crossfades: over user defined number of convolution cycles */
printf("enter number of processing buffers for fades (>1),8 is good for musical
        source,less for noisy sources:\n");
scanf("%d",&fade);
if(fade <= 0)
{
  printf("fade number must be positive, exiting\n");
  exit(1);
}
if(fade > 24)
  fade = 24;
fadebuffer = fade * irlength;

printf("enter mono (wav) sound file,include.wav extension(<100
       characters):\n");
scanf("%s", filename);

/* open files */
if(!(fin = sf_open(filename, SFM_READ, psfinfoin)))
{
  printf("error opening in file, exiting\n");
  exit(1);
}

if(psfinfoin->channels != 1)
{
  printf("input should be mono, exiting\n");
  exit(1);
}

if(!(hrtfleft = fopen("datal.raw", "rb")))
{
  printf("error opening hrtf file,exiting\n");
  exit(1);
}

if(!(hrtfright = fopen("datar.raw", "rb")))
{
  printf("error opening hrtf file,exiting\n");
  exit(1);
}
```

```
/* store files */
for(i = 0; i < 14; i++)
  for(j = 0; j < elevationarray[i] / 2 + 1; j++)
  {
    /* hrtfarray[i][j] = &hrtfarray[i][j][0] */
    hrtfarrayl[i][j] = new double [irlength];
    hrtfarrayr[i][j] = new double [irlength];
    fread(hrtfarrayl[i][j],sizeof(double), irlength, hrtfleft);
    fread(hrtfarrayr[i][j],sizeof(double), irlength, hrtfright);
  }

/* initialise the SF_INFO structure (need to do this before opening file!), same as
   input but stereo */
psfinfoout->samplerate = psfinfoin->samplerate;
psfinfoout->channels = 2;
psfinfoout->format = psfinfoin->format;

if(!(fout = sf_open("mover.wav", SFM_WRITE, psfinfoout)))
{
  printf("error opening out file\n");
  exit(1);
}

/* function to read, check and store trajectory */
bkpt(percentages, elevs, angles, &countbkp, maxbrkpts);

printf("...\nprocessing\n...\n");

/* main loop */
for(x = 0; x < countbkp; x++)
{
  start = sum;
  /* run to full length of convolved output */
  sum = (int)((psfinfoin->frames + irlength - 1) * percentages[x + 1] / 100.0);

  do
  {
    crossout = 0;
    crossfade = 0;

    /* change elev and angle according to bkpt file */
    elev = elevs[x] + (elevs[x + 1] - elevs[x]) * (double)(k - start) / (sum - start);
    angle = angles[x] + (angles[x + 1] - angles[x]) * (double)(k - start) / (sum –
            start);

    /* two nearest elev indices */
    /* to avoid recalculating  */
    elevindexstore = (elev - minelev) / elevincrement;
    elevindexlow = (int)elevindexstore;

    if(elevindexlow < 13)
      elevindexhigh = elevindexlow + 1;
    else
      elevindexhigh = elevindexlow;        /* highest index reached */

    /* get percentage value for interpolation */
    elevindexhighper = elevindexstore - elevindexlow;

    while(angle < 0.0)
      angle += 360.0;
    while(angle >= 360.0)
      angle -= 360.0;

    /* as above,lookup index, used to check for crossfade */
    elevindex = (int)(elevindexstore + 0.5);

    angleindex = (int)(angle / (360.0 / elevationarray[elevindex]) + 0.5);
    angleindex = angleindex % elevationarray[elevindex];

    /* crossfade happens if index changes:nearest measurement changes */
    if(oldelevindex != elevindex || oldangleindex != angleindex)
```

```
{
  if(k > 0)
  {
    /* warning on overlapping fades */
    if(cross)
    {
      printf("\nwarning: fades are overlapping: this could lead to noise: reduce
              fade size or change trajectory");
      cross = 0;
    }
    /* reset l */
    l = 0;
    crossfade = 1;
    for(i = 0; i < irpadlength; i++)
    {
      hrtflpadspecold[i] = hrtflpadspec[i];
      hrtfrpadspecold[i] = hrtfrpadspec[i];
    }
  }

  if(angleindex > elevationarray[elevindex] / 2)
  {
    hrtfpl = hrtfarrayl[elevindex][elevationarray[elevindex] - angleindex];
    hrtfpr = hrtfarrayr[elevindex][elevationarray[elevindex] - angleindex];
    for(i = 0; i < irlength; i++)
    {
      currentphasel[i]=hrtfpr[i];
      currentphaser[i]=hrtfpl[i];
    }
  }
  else
  {
    hrtfpl = hrtfarrayl[elevindex][angleindex];
    hrtfpr = hrtfarrayr[elevindex][angleindex];
    for(i = 0; i < irlength; i++)
    {
      currentphasel[i]=hrtfpl[i];
      currentphaser[i]=hrtfpr[i];
    }
  }
}

/* avoid recalculation */
angleindexlowstore = angle / (360.0 / elevationarray[elevindexlow]);
angleindexhighstore = angle / (360.0 / elevationarray[elevindexhigh]);

/* 4 closest indices, 2 low and 2 high */
angleindex1 = (int)angleindexlowstore;

angleindex2 = angleindex1 + 1;
angleindex2 = angleindex2 % elevationarray[elevindexlow];

angleindex3 = (int)angleindexhighstore;

angleindex4 = angleindex3 + 1;
angleindex4 = angleindex4 % elevationarray[elevindexhigh];

/* angle percentages for interp */
angleindex2per = angleindexlowstore - angleindex1;
angleindex4per = angleindexhighstore - angleindex3;

/* read 4 nearest HRTFs  */
/* switch l and r */
if(angleindex1 > elevationarray[elevindexlow] / 2)
{
  hrtfpl = hrtfarrayl[elevindexlow][elevationarray[elevindexlow] - angleindex1];
  hrtfpr = hrtfarrayr[elevindexlow][elevationarray[elevindexlow] - angleindex1];
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = hrtfpr[i];
    lowr1[i] = hrtfpl[i];
  }
}
else
{
  hrtfpl = hrtfarrayl[elevindexlow][angleindex1];
  hrtfpr = hrtfarrayr[elevindexlow][angleindex1];
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = hrtfpl[i];
    lowr1[i] = hrtfpr[i];
  }
}

if(angleindex2 > elevationarray[elevindexlow] / 2)
{
  hrtfpl = hrtfarrayl[elevindexlow][elevationarray[elevindexlow] - angleindex2];
  hrtfpr = hrtfarrayr[elevindexlow][elevationarray[elevindexlow] - angleindex2];
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = hrtfpr[i];
    lowr2[i] = hrtfpl[i];
  }
}
else
{
  hrtfpl = hrtfarrayl[elevindexlow][angleindex2];
  hrtfpr = hrtfarrayr[elevindexlow][angleindex2];
  for(i = 0; i < irlength; i++)
 {
   lowl2[i] = hrtfpl[i];
   lowr2[i] = hrtfpr[i];
 }
}

if(angleindex3 > elevationarray[elevindexhigh] / 2)
{
 hrtfpl = hrtfarrayl[elevindexhigh][elevationarray[elevindexhigh] - angleindex3];
 hrtfpr = hrtfarrayr[elevindexhigh][elevationarray[elevindexhigh] - angleindex3];
 for(i = 0; i < irlength; i++)
 {
   highl1[i] = hrtfpr[i];
   highr1[i] = hrtfpl[i];
 }
}
else
{
 hrtfpl = hrtfarrayl[elevindexhigh][angleindex3];
 hrtfpr = hrtfarrayr[elevindexhigh][angleindex3];
 for(i = 0; i < irlength; i++)
 {
   highl1[i] = hrtfpl[i];
   highr1[i] = hrtfpr[i];
 }
}

if(angleindex4 > elevationarray[elevindexhigh] / 2)
{
 hrtfpl = hrtfarrayl[elevindexhigh][elevationarray[elevindexhigh] - angleindex4];
 hrtfpr = hrtfarrayr[elevindexhigh][elevationarray[elevindexhigh] - angleindex4];
 for(i = 0; i < irlength; i++)
 {
   highl2[i] = hrtfpr[i];
   highr2[i] = hrtfpl[i];
 }
}
else
{
  hrtfpl = hrtfarrayl[elevindexhigh][angleindex4];
  hrtfpr = hrtfarrayr[elevindexhigh][angleindex4];
 for(i = 0; i < irlength; i++)
 {
    highl2[i] = hrtfpl[i];
    highr2[i] = hrtfpr[i];
  }
}

/* magnitude interpolation */
/* 0hz and Nyq real values */
/* organised in format of fftw */
magllow = fabs(lowl1[0]) + (fabs(lowl2[0]) – fabs(lowl1[0])) * angleindex2per;
maglhigh = fabs(highl1[0]) + (fabs(highl2[0]) – fabs(highl1[0])) * angleindex4per;
magrlow = fabs(lowr1[0]) + (fabs(lowr2[0]) – fabs(lowr1[0])) * angleindex2per;
magrhigh = fabs(highr1[0]) + (fabs(highr2[0]) – fabs(highr1[0])) * angleindex4per;
magl = magllow + (maglhigh - magllow) * elevindexhighper;
magr = magrlow + (magrhigh - magrlow) * elevindexhighper;
if(currentphasel[0] < 0.0)
  hrtflinterp[0] = -magl;
else
  hrtflinterp[0] = magl;
if(currentphaser[0] < 0.0)
  hrtfrinterp[0] = -magr;
else
  hrtfrinterp[0] = magr;

magllow = fabs(lowl1[1]) + (fabs(lowl2[1]) – fabs(lowl1[1])) * angleindex2per;
maglhigh = fabs(highl1[1]) + (fabs(highl2[1]) – fabs(highl1[1])) * angleindex4per;
magrlow = fabs(lowr1[1]) + (fabs(lowr2[1]) – fabs(lowr1[1])) * angleindex2per;
magrhigh = fabs(highr1[1]) + (fabs(highr2[1]) – fabs(highr1[1])) * angleindex4per;
magl = magllow + (maglhigh - magllow) * elevindexhighper;
magr = magrlow + (magrhigh - magrlow) * elevindexhighper;
if(currentphasel[1] < 0.0)
  hrtflinterp[irlength/2] = -magl;
else
  hrtflinterp[irlength/2] = magl;
if(currentphaser[1] < 0.0)
  hrtfrinterp[irlength/2] = -magr;
else
  hrtfrinterp[irlength/2] = magr;

/* other values are complex, in fftw format */
for(i = 2, j=1; i < irlength; j++, i+=2)
{
  /* interpolate high and low magnitudes */
  magllow = lowl1[i] + (lowl2[i] - lowl1[i]) * angleindex2per;
  maglhigh = highl1[i] + (highl2[i] - highl1[i]) * angleindex4per;

  magrlow = lowr1[i] + (lowr2[i] - lowr1[i]) * angleindex2per;
  magrhigh = highr1[i] + (highr2[i] - highr1[i]) * angleindex4per;

  /* interpolate high and low results, use current phase */
  magl = magllow +  (maglhigh - magllow) * elevindexhighper;
  phasel = currentphasel[i + 1];

  /* polar to rectangular, organised in fftw order */
  hrtflinterp[j] = magl * cos(phasel);
  hrtflinterp[irlength - j] = magl * sin(phasel);

  magr = magrlow + (magrhigh - magrlow) * elevindexhighper;
  phaser = currentphaser[i + 1];

  hrtfrinterp[j] = magr * cos(phaser);
  hrtfrinterp[irlength - j] = magr * sin(phaser);
}

fftw_execute(invhrtfl);
fftw_execute(invhrtfr);

/* scale and pad */
for(i = 0; i < irlength; i++)
{
  hrtflpadtd[i] = (hrtfltd[i] / irlength);
  hrtfrpadtd[i] = (hrtfrtd[i] / irlength);
```

```
}

for(i = irlength; i < irpadlength; i++)
{
 hrtflpadtd[i] = 0.0;
 hrtfrpadtd[i] = 0.0;
}

/* execute fft on padded hrtfs */
fftw_execute(forhrtflpad);
fftw_execute(forhrtfrpad);

/* look after overlap add */
for(i = 0; i < overlapsize ; i++)
{
 overlapl[i] = outl[i+irlength];
 overlapr[i] = outr[i+irlength];
  if(crossfade)
 {
   overlaplold[i] = outl[i+irlength];
   overlaprold[i] = outr[i+irlength];
 }
 /* overlap will be previous fading out signal */
 if(cross)
 {
   overlaplold[i] = outlold[i+irlength];
   overlaprold[i] = outrold[i+irlength];
 }
}

/* read input */
count = sf_readf_double(fin, inbuf, irlength);

/* zero pad */
/* fills last one with zeros from count */
for(i = (int)count; i < irpadlength; i++)
  inbuf[i] = 0.0;

/* fft input */
fftw_execute(forin);

/* convolution: spectral multiplication */
/* 0hz and Nyq */
outlspec[0] = inspec[0] * hrtflpadspec[0];
outrspec[0] = inspec[0] * hrtfrpadspec[0];
outlspec[irpadlength/2] = inspec[irpadlength/2] * hrtflpadspec[irpadlength/2];
outrspec[irpadlength/2] = inspec[irpadlength/2] * hrtfrpadspec[irpadlength/2];

/* complex multiplication according to fftw layout */
/* (a + i b)(c + i d) */
/* = (a c - b d) + i(a d + b c) */
for(i = 2, j = 1; i < irpadlength; j++, i+=2)
{
  /* real */
 outlspec[j] = inspec[j] * hrtflpadspec[j] – inspec[irpadlength - j] *
                hrtflpadspec[irpadlength - j];
 outrspec[j] = inspec[j] * hrtfrpadspec[j] – inspec[irpadlength - j] *
                hrtfrpadspec[irpadlength - j];
 /* imaginary */
 outlspec[irpadlength - j] = inspec[j] * hrtflpadspec[irpadlength - j] +
                            inspec[irpadlength - j] * hrtflpadspec[j];
 outrspec[irpadlength - j] = inspec[j] * hrtfrpadspec[irpadlength - j] +
                             inspec[irpadlength - j] * hrtfrpadspec[j];
}

fftw_execute(invoutl);
fftw_execute(invoutr);

/* scaled, as fftw is a sum */
for(i = 0; i < irpadlength; i++)
{
 outl[i] = outl[i] / irpadlength;
 outr[i] = outr[i] / irpadlength;
}

/* setup for fades */
if(crossfade || cross)
{
  crossout = 1;

  /* convolution */
 /* 0hz and Nyq */
 outlspecold[0] = inspec[0] * hrtflpadspecold[0];
 outrspecold[0] = inspec[0] * hrtfrpadspecold[0];
 outlspecold[irpadlength/2] = inspec[irpadlength/2] *
                               hrtflpadspecold[irpadlength/2];
 outrspecold[irpadlength/2] = inspec[irpadlength/2] *
                               hrtfrpadspecold[irpadlength/2];

 /* complex multiplication */
 for(i = 2, j = 1; i < irpadlength; j++, i+=2)
 {
   /* real */
   outlspecold[j] = inspec[j] * hrtflpadspecold[j] - inspec[irpadlength – j] *
                    hrtflpadspecold[irpadlength - j];
   outrspecold[j] = inspec[j] * hrtfrpadspecold[j] – inspec[irpadlength - j] *
                    hrtfrpadspecold[irpadlength - j];
   /* imaginary */
   outlspecold[irpadlength - j] = inspec[j] * hrtflpadspecold[irpadlength - j] +
                                  inspec[irpadlength - j] * hrtflpadspecold[j];
   outrspecold[irpadlength - j] = inspec[j] * hrtfrpadspecold[irpadlength - j] +
                                  inspec[irpadlength - j] * hrtfrpadspecold[j];
 }

 /* ifft, back to time domain */
 fftw_execute(invoutlold);
 fftw_execute(invoutrold);

 /* scaling */
 for(i = 0; i < irpadlength; i++)
 {
   outlold[i] = outlold[i] / irpadlength;
   outrold[i] = outrold[i] / irpadlength;
 }

 cross++;
 cross = cross % fade;
}

/* for next check */
oldelevindex = elevindex;
oldangleindex = angleindex;

if(crossout)
  for(i = 0; i < irlength; i++)
 {
   lrout[2 * i] = (outlold[i] + (i < overlapsize ? overlaplold[i] : 0.0)) * (1.0 –
                  (double)l / fadebuffer) + (outl[i] + (i < overlapsize ?
                  overlapl[i] : 0.0)) * (double)l / fadebuffer;
   lrout[(2 * i) + 1] = (outrold[i] + (i < overlapsize ? overlaprold[i] : 0.0)) *
                        (1.0 - (double)l / fadebuffer) + (outr[i] + (i <
                        overlapsize ? overlapr[i] : 0.0)) * (double)l /
                        fadebuffer;
   l++;
 }
else
  for(i = 0; i < irlength; i++)
 {
   lrout[2 * i] = outl[i] + (i < overlapsize ? overlapl[i] : 0.0);
   lrout[(2 * i) + 1] = outr[i] + (i < overlapsize ? overlapr[i] : 0.0);
 }
```

```
      /* do every irlength samples! */
      k += irlength;

      /* if on last run, only write output length mod irlength frames */
      if(k > psfinfoin->frames + irlength - 1)
        sf_writef_double(fout, lrout, (psfinfoin->frames + irlength - 1) % irlength);
      else
        sf_writef_double(fout, lrout, irlength);
    }
    while (k < sum);
  }

  /* clear dynamic memory, close files */
  delete psfinfoin;
  delete psfinfoout;
  sf_close(fin);
  sf_close(fout);
  fclose(hrtfleft);
  fclose(hrtfright);

  for(i = 0; i < 14; i++)
    for(j = 0; j < elevationarray[i] / 2 + 1; j++)
    {
      delete[] hrtfarrayl[i][j];
      delete[] hrtfarrayr[i][j];
    }

  fftw_destroy_plan(invhrtfl);
  fftw_destroy_plan(invhrtfr);
  fftw_destroy_plan(forhrtflpad);
  fftw_destroy_plan(forhrtfrpad);
  fftw_destroy_plan(forin);
  fftw_destroy_plan(invoutl);
  fftw_destroy_plan(invoutr);
  fftw_destroy_plan(invoutlold);
  fftw_destroy_plan(invoutrold);

  return 0;
}
```

1.4 binauralmoverfunctions.cpp

```
/*
Brian Carty PhD Code 2010
Chapter 4,
binauralmoverfunctions.cpp
*/

#include "defs.h"

void bkpt(int *pers, double *els, double *angs, int *noofpoints, int maxpts)
{
  /* file details */
  FILE *finbkp;
  char bkpfilename[100];
  int i;

  printf("enter breakpoint file (integer value percentages),include.txt extension (<100
          characters):\n");
  scanf("%s",bkpfilename);

  if(!(finbkp = fopen(bkpfilename,"r")))
  {
    printf("error opening breakpoint file, exiting\n");
    exit(1);
  }

  for(i = 0; i < maxpts; i++)
  {
    /* read input from file */
    if(!feof(finbkp))
    {
      fscanf(finbkp,"%d",&pers[i]);
      fscanf(finbkp,"%lf",&els[i]);
      if(els[i] > 90.0)
        els[i] = 90.0;
      if(els[i] < -40.0)
        els[i] = -40.0;
      fscanf(finbkp,"%lf",&angs[i]);

      /* do checks */
      /* legal % values ? */
      if(pers[i] > 100 || pers[i] < 0)
      {
        printf("error, breakpoint file must run from 0 to 100, exiting\n");
        exit(1);
      }
      /* percentage accumulation */
      if(i > 0 && pers[i] <= pers[i - 1])
      {
        printf("error, percentage values must accumulate...%d is not > %d,
               exiting\n",pers[i],pers[i-1]);
        exit(1);
      }

      /* end at 100% */
      if(pers[i] == 100)
      break;

    *noofpoints = *noofpoints + 1;
    }
    else
    break;
  }

  /* check last value is 100 */
  if(pers[*noofpoints] != 100)
  {
    printf("error, percentage values must conclude with 100, not %d, exiting",pers[i]);
    exit(1);
  }

  /* close file */
  fclose(finbkp);
}
```

Appendix 2: HRTF Opcodes

```
/*
hrtfopcodes.c: new HRTF opcodes

(c) Brian Carty, 2010

This file is part of Csound.

The Csound Library is free software; you can redistribute it
and/or modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

Csound is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with Csound; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
02111-1307 USA
*/

#include "csdl.h"
#include <math.h>
/* definitions */
/* from mit */
#define minelev -40
#define elevincrement 10

/* max delay for min phase: a time value: multiply by sr to get no of samples for memory
   allocation */
#define maxdeltime 0.0011

/* additional definitions for woodworth models */
#define c 34400.0

/* hrtf data sets were analysed for low frequency phase values, as it is the important
   part of the spectrum for phase based localisation cues. The values below were
   extracted and are used to scale the functional phase spectrum. radius: 8.8 cm see
   nonlinitd.cpp */
static const float nonlinitd[5] = {1.570024f, 1.378733f, 1.155164f, 1.101230f, 1.0f};
static const float nonlinitd48k[5] = {1.549748f, 1.305457f, 1.124501f, 1.112852f, 1.0f};
static const float nonlinitd96k[5] = {1.550297f, 1.305671f, 1.124456f, 1.112818f, 1.0f};

/* number of measurements per elev: mit data const:read only, static:exists for whole
   process... */
static const int elevationarray[14] = {56, 60, 72, 72, 72, 72, 72, 60, 56, 45, 36, 24,
                                        12, 1 };

/* assumed mit hrtf data will be used here. Otherwise delay data would need to be
   extracted and replaced here... */
static const float minphasedels[368] =
{
0.000000f, 0.000045f, 0.000091f, 0.000136f, 0.000159f, 0.000204f,
0.000249f, 0.000272f, 0.000295f, 0.000317f, 0.000363f, 0.000385f,
0.000272f, 0.000408f, 0.000454f, 0.000454f, 0.000408f, 0.000385f,
0.000363f, 0.000317f, 0.000295f, 0.000295f, 0.000249f, 0.000204f,
0.000159f, 0.000136f, 0.000091f, 0.000045f, 0.000000f, 0.000000f,
0.000045f, 0.000091f, 0.000136f, 0.000181f, 0.000227f, 0.000249f,
0.000272f, 0.000317f, 0.000363f, 0.000385f, 0.000454f, 0.000476f,
0.000454f, 0.000522f, 0.000499f, 0.000499f, 0.000476f, 0.000454f,
0.000408f, 0.000408f, 0.000385f, 0.000340f, 0.000295f, 0.000272f,
0.000227f, 0.000181f, 0.000136f, 0.000091f, 0.000045f, 0.000000f,
0.000000f, 0.000045f, 0.000091f, 0.000113f, 0.000159f, 0.000204f,
0.000227f, 0.000272f, 0.000317f, 0.000317f, 0.000363f, 0.000408f,
0.000363f, 0.000522f, 0.000476f, 0.000499f, 0.000590f, 0.000567f,
0.000567f, 0.000544f, 0.000522f, 0.000499f, 0.000476f, 0.000454f,
0.000431f, 0.000408f, 0.000385f, 0.000363f, 0.000317f, 0.000295f,
0.000249f, 0.000204f, 0.000181f, 0.000136f, 0.000091f, 0.000045f,
0.000000f, 0.000000f, 0.000045f, 0.000091f, 0.000113f, 0.000159f,
0.000204f, 0.000249f, 0.000295f, 0.000317f, 0.000363f, 0.000340f,
0.000385f, 0.000431f, 0.000476f, 0.000522f, 0.000544f, 0.000612f,
0.000658f, 0.000658f, 0.000635f, 0.000658f, 0.000522f, 0.000499f,
0.000476f, 0.000454f, 0.000408f, 0.000385f, 0.000363f, 0.000340f,
0.000295f, 0.000272f, 0.000227f, 0.000181f, 0.000136f, 0.000091f,
0.000045f, 0.000000f, 0.000000f, 0.000045f, 0.000091f, 0.000136f,
0.000159f, 0.000204f, 0.000249f, 0.000295f, 0.000340f, 0.000385f,
0.000431f, 0.000476f, 0.000522f, 0.000567f, 0.000522f, 0.000567f,
0.000567f, 0.000635f, 0.000703f, 0.000748f, 0.000748f, 0.000726f,
0.000703f, 0.000658f, 0.000454f, 0.000431f, 0.000385f, 0.000363f,
0.000317f, 0.000295f, 0.000272f, 0.000227f, 0.000181f, 0.000136f,
0.000091f, 0.000045f, 0.000000f, 0.000000f, 0.000045f, 0.000091f,
0.000113f, 0.000159f, 0.000204f, 0.000249f, 0.000295f, 0.000340f,
0.000385f, 0.000408f, 0.000454f, 0.000499f, 0.000544f, 0.000522f,
0.000590f, 0.000590f, 0.000635f, 0.000658f, 0.000680f, 0.000658f,
0.000544f, 0.000590f, 0.000567f, 0.000454f, 0.000431f, 0.000385f,
0.000363f, 0.000317f, 0.000272f, 0.000272f, 0.000227f, 0.000181f,
0.000136f, 0.000091f, 0.000045f, 0.000000f, 0.000000f, 0.000045f,
0.000068f, 0.000113f, 0.000159f, 0.000204f, 0.000227f, 0.000272f,
0.000317f, 0.000340f, 0.000385f, 0.000431f, 0.000454f, 0.000499f,
0.000499f, 0.000544f, 0.000567f, 0.000590f, 0.000590f, 0.000590f,
0.000590f, 0.000567f, 0.000567f, 0.000476f, 0.000454f, 0.000408f,
0.000385f, 0.000340f, 0.000340f, 0.000295f, 0.000249f, 0.000204f,
0.000159f, 0.000136f, 0.000091f, 0.000045f, 0.000000f, 0.000000f,
0.000045f, 0.000091f, 0.000113f, 0.000159f, 0.000204f, 0.000249f,
0.000295f, 0.000340f, 0.000363f, 0.000385f, 0.000431f, 0.000454f,
0.000499f, 0.000522f, 0.000522f, 0.000522f, 0.000499f, 0.000476f,
0.000454f, 0.000431f, 0.000385f, 0.000340f, 0.000317f, 0.000272f,
0.000227f, 0.000181f, 0.000136f, 0.000091f, 0.000045f, 0.000000f,
0.000000f, 0.000045f, 0.000091f, 0.000136f, 0.000159f, 0.000204f,
0.000227f, 0.000249f, 0.000295f, 0.000340f, 0.000363f, 0.000385f,
0.000408f, 0.000431f, 0.000431f, 0.000431f, 0.000431f, 0.000408f,
0.000385f, 0.000363f, 0.000317f, 0.000317f, 0.000272f, 0.000227f,
0.000181f, 0.000136f, 0.000091f, 0.000045f, 0.000000f, 0.000000f,
0.000045f, 0.000091f, 0.000136f, 0.000181f, 0.000204f, 0.000227f,
0.000272f, 0.000295f, 0.000317f, 0.000340f, 0.000340f, 0.000363f,
0.000363f, 0.000340f, 0.000317f, 0.000295f, 0.000249f, 0.000204f,
0.000159f, 0.000113f, 0.000068f, 0.000023f, 0.000000f, 0.000045f,
0.000068f, 0.000113f, 0.000159f, 0.000181f, 0.000204f, 0.000227f,
0.000249f, 0.000249f, 0.000249f, 0.000227f, 0.000227f, 0.000181f,
0.000159f, 0.000113f, 0.000091f, 0.000045f, 0.000000f, 0.000000f,
0.000045f, 0.000091f, 0.000136f, 0.000159f, 0.000181f, 0.000181f,
0.000181f, 0.000159f, 0.000136f, 0.000091f, 0.000045f, 0.000000f,
0.000000f, 0.000045f, 0.000068f, 0.000091f, 0.000068f, 0.000045f,
0.000000f, 0.000000f
};


#ifdef WORDS_BIGENDIAN
static int swap4bytes(CSOUND* csound, MEMFIL* mfp)
{
  char c1, c2, c3, c4;
  char *p = mfp->beginp;
  int  size = mfp->length;

  while (size >= 4)
  {
    c1 = p[0]; c2 = p[1]; c3 = p[2]; c4 = p[3];
    p[0] = c4; p[1] = c3; p[2] = c2; p[3] = c1;
    size -= 4; p +=4;
  }

  return OK;
}
#else
static int (*swap4bytes)(CSOUND*, MEMFIL*) = NULL;
```

```
#endif

/* Csound hrtf magnitude interpolation, phase truncation object */

/* aleft,aright hrtfmove asrc, kaz, kel, ifilel, ifiler [, imode = 0, ifade = 8, sr =
   44100]... */
/* imode: minphase/phase truncation, ifade: no of buffers per fade for phase trunc., sr
   can be 44.1/48/96k */

typedef struct
{
  OPDS  h;
  /* outputs and inputs */
  MYFLT *outsigl, *outsigr;
  MYFLT *in, *kangle, *kelev, *ifilel, *ifiler, *omode, *ofade, *osr;

  /* check if relative source has changed! */
  MYFLT anglev, elevv;

  float *fpbeginl,*fpbeginr;

  /* see definitions in INIT */
  int irlength, irlengthpad, overlapsize;

  MYFLT sr;

  /* old indices for checking if changes occur in trajectory. */
  int oldelevindex, oldangleindex;

  int counter;

  /* initialfade used to avoid fade in of data...if not,'old' data faded out with zero
     hrtf,'new' data faded in. */
  int cross,l,initialfade;

  /* user defined buffer size for fades. */
  int fadebuffer, fade;

  /* flags for process type */
  int minphase,phasetrunc;

  /* hrtf data padded */
  AUXCH hrtflpad,hrtfrpad;
  /* old data for fades */
  AUXCH oldhrtflpad,oldhrtfrpad;
  /* in and output buffers */
  AUXCH insig, outl, outr, outlold, outrold;

  /* memory local to perform method */
  /* insig fft */
  AUXCH complexinsig;
  /* hrtf buffers (rectangular complex form) */
  AUXCH hrtflfloat, hrtfrfloat;
  /* spectral data */
  AUXCH outspecl, outspecr, outspecoldl, outspecoldr;

  /* overlap data */
  AUXCH overlapl, overlapr;
  /* old overlap data for longer crossfades */
  AUXCH overlapoldl, overlapoldr;

  /* interpolation buffers */
  AUXCH lowl1, lowr1, lowl2, lowr2, highl1, highr1, highl2, highr2;
  /* current phase buffers */
  AUXCH currentphasel, currentphaser;

  /* min phase buffers */
  AUXCH logmagl,logmagr,xhatwinl,xhatwinr,expxhatwinl,expxhatwinr;
  /* min phase window: a static buffer */
  AUXCH win;
  MYFLT delayfloat;
  /* delay */
  AUXCH delmeml, delmemr;
  int ptl, ptr, mdtl, mdtr;
}
hrtfmove;

static int hrtfmove_init(CSOUND *csound, hrtfmove *p)
{
  /* left and right data files: spectral mag, phase format. */
  MEMFIL *fpl = NULL,*fpr = NULL;
  int i;
  char filel[MAXNAME],filer[MAXNAME];

  int mode = (int)*p->omode;
  int fade = (int)*p->ofade;
  MYFLT sr = *p->osr;

  MYFLT *win;

  /* time domain impulse length, padded, overlap add */
  int irlength, irlengthpad, overlapsize;

  /* flag for process type: default phase trunc */
  if(mode == 1)
  {
    p->minphase = 1;
    p->phasetrunc = 0;
  }
  else
  {
    p->phasetrunc = 1;
    p->minphase = 0;
  }

  /* fade length: default 8, max 24, min 1 */
  if(fade < 1 || fade > 24)
    fade = 8;
  p->fade = fade;

  /* sr, defualt 44100 */
  if(sr != 44100 && sr != 48000 && sr != 96000)
    sr = 44100;
  p->sr = sr;

  if(UNLIKELY(csound->esr != sr))
    csound->Message(csound, Str("\n\nWARNING!!:\nOrchestra SR not compatible with HRTF
                                 processing SR of: %.0f\n\n"), sr);

  /* setup as per sr */
  if(sr == 44100 || sr == 48000)
  {
    irlength = 128;
    irlengthpad = 256;
    overlapsize = (irlength - 1);
  }
  else if(sr == 96000)
  {
    irlength = 256;
    irlengthpad = 512;
    overlapsize = (irlength - 1);
  }

  /* copy in string name */
  strcpy(filel, (char*) p->ifilel);
  strcpy(filer, (char*) p->ifiler);

  /* reading files, with byte swap */
  if(UNLIKELY((fpl = csound->ldmemfile2withCB(csound, filel, CSFTYPE_FLOATS_BINARY,
     swap4bytes)) == NULL))
    return
    csound->InitError(csound, Str("\n\n\nCannot load left data file, exiting\n\n"));

if (UNLIKELY((fpr = csound->ldmemfile2withCB(csound, filer, CSFTYPE_FLOATS_BINARY,
    swap4bytes)) == NULL))
  return
    csound->InitError(csound, Str("\n\n\nCannot load right data file, exiting\n\n"));

p->irlength = irlength;
p->irlengthpad = irlengthpad;
p->overlapsize = overlapsize;

/* the amount of buffers to fade over. */
p->fadebuffer = (int)fade*irlength;

/* file handles */
p->fpbeginl = (float *) fpl->beginp;
p->fpbeginr = (float *) fpr->beginp;

/* common buffers (used by both min phase and phasetrunc) */
if (!p->insig.auxp || p->insig.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->insig);
if (!p->outl.auxp || p->outl.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outl);
if (!p->outr.auxp || p->outr.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outr);
if (!p->hrtflpad.auxp || p->hrtflpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->hrtflpad);
if (!p->hrtfrpad.auxp || p->hrtfrpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->hrtfrpad);
if (!p->complexinsig.auxp || p->complexinsig.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->complexinsig);
if (!p->hrtflfloat.auxp || p->hrtflfloat.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->hrtflfloat);
if (!p->hrtfrfloat.auxp || p->hrtfrfloat.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->hrtfrfloat);
if (!p->outspecl.auxp || p->outspecl.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outspecl);
if (!p->outspecr.auxp || p->outspecr.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outspecr);
if (!p->overlapl.auxp || p->overlapl.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize*sizeof(MYFLT), &p->overlapl);
if (!p->overlapr.auxp || p->overlapr.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize*sizeof(MYFLT), &p->overlapr);

memset(p->insig.auxp, 0, irlength * sizeof(MYFLT));
memset(p->outl.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outr.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtflpad.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtfrpad.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->complexinsig.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtflfloat.auxp, 0, irlength * sizeof(MYFLT));
memset(p->hrtfrfloat.auxp, 0, irlength * sizeof(MYFLT));
memset(p->outspecl.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outspecr.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->overlapl.auxp, 0, overlapsize * sizeof(MYFLT));
memset(p->overlapr.auxp, 0, overlapsize * sizeof(MYFLT));

/* interpolation values */
if (!p->lowl1.auxp || p->lowl1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowl1);
if (!p->lowr1.auxp || p->lowr1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowr1);
if (!p->lowl2.auxp || p->lowl2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowl2);
if (!p->lowr2.auxp || p->lowr2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowr2);
if (!p->highl1.auxp || p->highl1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highl1);
if (!p->highr1.auxp || p->highr1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highr1);
if (!p->highl2.auxp || p->highl2.size < irlength * sizeof(MYFLT))
```

```
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highl2);
if (!p->highr2.auxp || p->highr2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highr2);
if (!p->currentphasel.auxp || p->currentphasel.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->currentphasel);
if (!p->currentphaser.auxp || p->currentphaser.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->currentphaser);

memset(p->lowl1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowr1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowl2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowr2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highl1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highl2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highr1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highr2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->currentphasel.auxp, 0, irlength * sizeof(MYFLT));
memset(p->currentphaser.auxp, 0, irlength * sizeof(MYFLT));

  /* phase truncation buffers and variables */
if (!p->oldhrtflpad.auxp || p->oldhrtflpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->oldhrtflpad);
if (!p->oldhrtfrpad.auxp || p->oldhrtfrpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->oldhrtfrpad);
if (!p->outlold.auxp || p->outlold.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outlold);
if (!p->outrold.auxp || p->outrold.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outrold);
if (!p->outspecoldl.auxp || p->outspecoldl.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outspecoldl);
if (!p->outspecoldr.auxp || p->outspecoldr.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outspecoldr);
if (!p->overlapoldl.auxp || p->overlapoldl.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize*sizeof(MYFLT), &p->overlapoldl);
if (!p->overlapoldr.auxp || p->overlapoldr.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize*sizeof(MYFLT), &p->overlapoldr);

memset(p->oldhrtflpad.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->oldhrtfrpad.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outlold.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outrold.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outspecoldl.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outspecoldr.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->overlapoldl.auxp, 0, overlapsize * sizeof(MYFLT));
memset(p->overlapoldr.auxp, 0, overlapsize * sizeof(MYFLT));

/* initialize counters and indices */
p->counter = 0;
p->cross = 0;
p->l = 0;
p->initialfade = 0;

/* need to be a value that is not possible for first check to avoid phase not being
   read. */
p->oldelevindex = -1;
p->oldangleindex = -1;

/* buffer declaration for min phase calculations */
if (!p->logmagl.auxp || p->logmagl.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->logmagl);
if (!p->logmagr.auxp || p->logmagr.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->logmagr);
if (!p->xhatwinl.auxp || p->xhatwinl.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->xhatwinl);
if (!p->xhatwinr.auxp || p->xhatwinr.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->xhatwinr);
if (!p->expxhatwinl.auxp || p->expxhatwinl.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->expxhatwinl);
if (!p->expxhatwinr.auxp || p->expxhatwinr.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->expxhatwinr);
  memset(p->logmagl.auxp, 0, irlength * sizeof(MYFLT));
  memset(p->logmagr.auxp, 0, irlength * sizeof(MYFLT));
  memset(p->xhatwinl.auxp, 0, irlength * sizeof(MYFLT));
  memset(p->xhatwinr.auxp, 0, irlength * sizeof(MYFLT));
  memset(p->expxhatwinl.auxp, 0, irlength * sizeof(MYFLT));
  memset(p->expxhatwinr.auxp, 0, irlength * sizeof(MYFLT));

      /* delay buffers */
  if (!p->delmeml.auxp || p->delmeml.size < (int)(sr * maxdeltime) * sizeof(MYFLT))
    csound->AuxAlloc(csound, (int)(sr * maxdeltime) * sizeof(MYFLT), &p->delmeml);
  if (!p->delmemr.auxp || p->delmemr.size < (int)(sr * maxdeltime) * sizeof(MYFLT))
    csound->AuxAlloc(csound, (int)(sr * maxdeltime) * sizeof(MYFLT), &p->delmemr);

  memset(p->delmeml.auxp, 0, (int)(sr * maxdeltime) * sizeof(MYFLT));
  memset(p->delmemr.auxp, 0, (int)(sr * maxdeltime) * sizeof(MYFLT));

  if (!p->win.auxp && p->win.size < irlength * sizeof(MYFLT))
    csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->win);

  win = (MYFLT *)p->win.auxp;

  /* min phase win defined for irlength point impulse! */
  win[0] = FL(1.0);
  for(i = 1; i < (irlength / 2); i++)
    win[i] = FL(2.0);
    win[(irlength / 2)] = FL(1.0);
    for(i = ((irlength / 2) + 1); i < irlength; i++)
      win[i] = FL(0.0);

  p->mdtl = (int)(0.00095f * sr);
  p->mdtr = (int)(0.00095f * sr);
  p->delayfloat = 0.;

  p->ptl = 0;
  p->ptr = 0;

  /* setup values used to check if src has moved, illegal values to start with to ensure
     first read */
  p->anglev = -1;
  p->elevv = -41;

  return OK;
}


static int hrtfmove_process(CSOUND *csound, hrtfmove *p)
{
  /* local pointers to p */
  MYFLT *in = p->in;
  MYFLT *outsigl  = p->outsigl;
  MYFLT *outsigr = p->outsigr;

  /* common buffers and variables */
  MYFLT *insig = (MYFLT *)p->insig.auxp;
  MYFLT *outl = (MYFLT *)p->outl.auxp;
  MYFLT *outr = (MYFLT *)p->outr.auxp;

  MYFLT *hrtflpad = (MYFLT *)p->hrtflpad.auxp;
  MYFLT *hrtfrpad = (MYFLT *)p->hrtfrpad.auxp;

  MYFLT *complexinsig = (MYFLT *)p->complexinsig.auxp;
  MYFLT *hrtflfloat = (MYFLT *)p->hrtflfloat.auxp;
  MYFLT *hrtfrfloat = (MYFLT *)p->hrtfrfloat.auxp;
  MYFLT *outspecl = (MYFLT *)p->outspecl.auxp;
  MYFLT *outspecr = (MYFLT *)p->outspecr.auxp;

  MYFLT *overlapl = (MYFLT *)p->overlapl.auxp;
  MYFLT *overlapr = (MYFLT *)p->overlapr.auxp;

  MYFLT elev = *p->kelev;
  MYFLT angle = *p->kangle;
```

```
int counter = p->counter;
int n;

/* pointers into HRTF files: floating point data (even in 64 bit csound) */
float *fpindexl;
float *fpindexr;

int i,j,elevindex, angleindex, skip = 0;

int minphase = p->minphase;
int phasetrunc = p->phasetrunc;

MYFLT sr = p->sr;

int irlength = p->irlength;
int irlengthpad = p->irlengthpad;
int overlapsize = p->overlapsize;

/* local variables, mainly used for simplification */
MYFLT elevindexstore;
MYFLT angleindexlowstore;
MYFLT angleindexhighstore;

/* interpolation values */
MYFLT *lowl1 = (MYFLT *)p->lowl1.auxp;
MYFLT *lowr1 = (MYFLT *)p->lowr1.auxp;
MYFLT *lowl2 = (MYFLT *)p->lowl2.auxp;
MYFLT *lowr2 = (MYFLT *)p->lowr2.auxp;
MYFLT *highl1 = (MYFLT *)p->highl1.auxp;
MYFLT *highr1 = (MYFLT *)p->highr1.auxp;
MYFLT *highl2 = (MYFLT *)p->highl2.auxp;
MYFLT *highr2 = (MYFLT *)p->highr2.auxp;
MYFLT *currentphasel = (MYFLT *)p->currentphasel.auxp;
MYFLT *currentphaser = (MYFLT *)p->currentphaser.auxp;

/* local interpolation values */
MYFLT elevindexhighper, angleindex2per, angleindex4per;
int elevindexlow, elevindexhigh, angleindex1, angleindex2, angleindex3, angleindex4;
MYFLT magl,magr,phasel,phaser, magllow, magrlow, maglhigh, magrhigh;

/* phase truncation buffers and variables */
MYFLT *oldhrtflpad = (MYFLT *)p->oldhrtflpad.auxp;
MYFLT *oldhrtfrpad = (MYFLT *)p->oldhrtfrpad.auxp;
MYFLT *outlold = (MYFLT *)p->outlold.auxp;
MYFLT *outrold = (MYFLT *)p->outrold.auxp;
MYFLT *outspecoldl = (MYFLT *)p->outspecoldl.auxp;
MYFLT *outspecoldr = (MYFLT *)p->outspecoldr.auxp;
MYFLT *overlapoldl = (MYFLT *)p->overlapoldl.auxp;
MYFLT *overlapoldr = (MYFLT *)p->overlapoldr.auxp;

int oldelevindex = p ->oldelevindex;
int oldangleindex = p ->oldangleindex;

int cross = p ->cross;
int l = p->l;
int initialfade = p->initialfade;

int crossfade;
int crossout;

int fade = p->fade;
int fadebuffer = p->fadebuffer;

/* minimum phase buffers */
MYFLT *logmagl = (MYFLT *)p->logmagl.auxp;
MYFLT *logmagr = (MYFLT *)p->logmagr.auxp;
MYFLT *xhatwinl = (MYFLT *)p->xhatwinl.auxp;
MYFLT *xhatwinr = (MYFLT *)p->xhatwinr.auxp;
MYFLT *expxhatwinl = (MYFLT *)p->expxhatwinl.auxp;
MYFLT *expxhatwinr = (MYFLT *)p->expxhatwinr.auxp;
```

```
/* min phase window */
MYFLT *win = (MYFLT *)p->win.auxp;

/* min phase delay variables */
MYFLT *delmeml = (MYFLT *)p->delmeml.auxp;
MYFLT *delmemr = (MYFLT *)p->delmemr.auxp;
MYFLT delaylow1, delaylow2, delayhigh1, delayhigh2, delaylow, delayhigh;
MYFLT delayfloat = p->delayfloat;
int ptl = p->ptl;
int ptr = p->ptr;
int mdtl = p->mdtl;
int mdtr = p->mdtr;
int posl, posr;
MYFLT outvdl, outvdr, vdtl, vdtr, fracl, fracr, rpl, rpr;

/* start indices at correct value (start of file)/ zero indices. */
fpindexl = (float *) p->fpbeginl;
fpindexr = (float *) p->fpbeginr;

n = csound->ksmps;

for(j = 0; j < n; j++)
{
  /* ins and outs */
  insig[counter] = in[j];

  outsigl[j] = outl[counter];
  outsigr[j] = outr[counter];

  counter++;

  if(phasetrunc)
  {
    /* used to ensure fade does not happen on first run */
    if(initialfade < (irlength + 2))
      initialfade++;
  }

  if(counter == irlength)
  {
    /* process a block */
    crossfade = 0;
    crossout = 0;

    if(elev > FL(90.0))
      elev = FL(90.0);
    if(elev < FL(-40.0))
      elev = FL(-40.0);

    while(angle < FL(0.0))
      angle += FL(360.0);
    while(angle >= FL(360.0))
      angle -= FL(360.0);

    /* only update if location changes! */
    if(angle != p->anglev || elev != p->elevv)
    {
      /* two nearest elev indices to avoid recalculating */
      elevindexstore = (elev - minelev) / elevincrement;
      elevindexlow = (int)elevindexstore;

      if(elevindexlow < 13)
        elevindexhigh = elevindexlow + 1;
      /* highest index reached */
      else
        elevindexhigh = elevindexlow;

      /* get percentage value for interpolation */
      elevindexhighper = elevindexstore – elevindexlow;
```

```
/* read using an index system based on number of points measured per elevation
   at mit */
/* lookup indices, used to check for crossfade */
elevindex = (int)(elevindexstore + FL(0.5));
angleindex = (int)(angle / (FL(360.0) / elevationarray[elevindex]) + FL(0.5));
angleindex = angleindex % elevationarray[elevindex];

/* avoid recalculation */
angleindexlowstore = angle / (FL(360.) / elevationarray[elevindexlow]);
angleindexhighstore = angle / (FL(360.) / elevationarray[elevindexhigh]);

/* 4 closest indices, 2 low and 2 high */
angleindex1 = (int)angleindexlowstore;

angleindex2 = angleindex1 + 1;
angleindex2 = angleindex2 % elevationarray[elevindexlow];

angleindex3 = (int)angleindexhighstore;

angleindex4 = angleindex3 + 1;
angleindex4 = angleindex4 % elevationarray[elevindexhigh];

/* angle percentages for interp */
angleindex2per = angleindexlowstore – angleindex1;
angleindex4per = angleindexhighstore – angleindex3;

if(phasetrunc)
{
  if(angleindex!=oldangleindex || elevindex!=oldelevindex)
 {
    /* store last point and turn crossfade on, provided that initialfade value
       indicates first block processed! */
    /* (otherwise,there will be a fade in at the start). */
    if(initialfade>irlength)
   {
      /* post warning if fades ovelap */
      if(cross)
      {
        csound->Message(csound,Str("\nWARNING: fades are overlapping: this could
                lead to noise: reduce fade size or change trajectory\n\n"));
         cross = 0;
      }
      /* reset l, use as index to fade */
      l = 0;
      crossfade = 1;
      /* store old data */
      for(i = 0; i < irlengthpad; i++)
      {
        oldhrtflpad[i] = hrtflpad[i];
        oldhrtfrpad[i] = hrtfrpad[i];
      }
   }

   /* store point for current phase as trajectory comes closer to a new index */
    skip = 0;
    /* store current phase */
    if(angleindex > elevationarray[elevindex] / 2)
   {
      for(i = 0; i < elevindex; i++)
        skip +=((int)(elevationarray[i]/ 2) + 1) * irlength;
       for (i = 0; i < (elevationarray[elevindex] - angleindex); i++)
         skip += irlength;
      for(i = 0; i < irlength; i++)
      {
        currentphasel[i] = fpindexr[skip + i];
        currentphaser[i] = fpindexl[skip + i];
      }
   }
    else
   {
      for(i = 0; i < elevindex; i++)
```

```
        skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
      for (i = 0; i < angleindex; i++)
        skip += irlength;
      for(i = 0; i < irlength; i++)
      {
        currentphasel[i] = fpindexl[skip+i];
         currentphaser[i] = fpindexr[skip+i];
       }
    }
  }
}

/* for next check */
p->oldelevindex = elevindex;
p->oldangleindex = angleindex;

/* read 4 nearest HRTFs */
skip = 0;
/* switch l and r */
if(angleindex1>elevationarray[elevindexlow]/2)
{
  for(i = 0; i < elevindexlow; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i=0;i<(elevationarray[elevindexlow] – angleindex1); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
     lowl1[i] = fpindexr[skip+i];
     lowr1[i] = fpindexl[skip+i];
  }
}
else
{
  for(i = 0; i < elevindexlow; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex1; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = fpindexl[skip+i];
    lowr1[i] = fpindexr[skip+i];
  }
}

skip = 0;
if(angleindex2 > elevationarray[elevindexlow] / 2)
{
  for(i = 0; i < elevindexlow; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for(I = 0;I < (elevationarray[elevindexlow] - angleindex2); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = fpindexr[skip+i];
    lowr2[i] = fpindexl[skip+i];
  }
}
else
{
  for(i = 0; i < elevindexlow; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for(i = 0; i < angleindex2; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = fpindexl[skip+i];
    lowr2[i] = fpindexr[skip+i];
  }
}

skip = 0;
```

```
if(angleindex3>elevationarray[elevindexhigh]/2)
{
  for(i = 0; i < elevindexhigh; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexhigh] – angleindex3); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl1[i] = fpindexr[skip+i];
    highr1[i] = fpindexl[skip+i];
  }
}
else
{
  for(i = 0; i < elevindexhigh; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex3; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl1[i] = fpindexl[skip+i];
    highr1[i] = fpindexr[skip+i];
  }
}

skip = 0;
if(angleindex4>elevationarray[elevindexhigh]/2)
{
  for(i = 0; i < elevindexhigh; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexhigh] – angleindex4); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl2[i] = fpindexr[skip+i];
    highr2[i] = fpindexl[skip+i];
  }
}
else
{
  for(i = 0; i < elevindexhigh; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex4; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl2[i] = fpindexl[skip+i];
    highr2[i] = fpindexr[skip+i];
  }
}

/* interpolation */
/* 0 Hz and Nyq...absoulute values for mag */
magllow = FL(fabs(lowl1[0])) + (FL(fabs(lowl2[0])) - FL(fabs(lowl1[0]))) *
          angleindex2per;
maglhigh = FL(fabs(highl1[0])) + (FL(fabs(highl2[0])) – FL(fabs(highl1[0]))) *
           angleindex4per;
magl = magllow + (maglhigh - magllow) * elevindexhighper;
if(minphase)
{
  logmagl[0] = LOG((magl == FL(0.0) ? FL(0.00000001) : magl));
}
/* this is where real values of 0hz and nyq needed: if neg real, 180 degree phase
*/
/* if data was complex, mag interp would use fabs() inherently, phase would be
   0/pi */
/* if pi, real is negative! */
else
{
  if(currentphasel[0] < FL(0.))
    hrtflfloat[0] = -magl;
  else
   hrtflfloat[0] = magl;
}

magllow = FL(fabs(lowl1[1])) + (FL(fabs(lowl2[1])) - FL(fabs(lowl1[1]))) *
          angleindex2per;
maglhigh = FL(fabs(highl1[1])) + (FL(fabs(highl2[1])) - FL(fabs(highl1[1]))) *
           angleindex4per;
magl = magllow + (maglhigh-magllow) * elevindexhighper;
if(minphase)
{
  logmagl[1] = LOG(magl == FL(0.0) ? FL(0.00000001) : magl);
}
else
{
   if(currentphasel[1] < FL(0.))
     hrtflfloat[1] = -magl;
   else
     hrtflfloat[1] = magl;
}

magrlow = FL(fabs(lowr1[0])) + (FL(fabs(lowr2[0])) - FL(fabs(lowr1[0]))) *
          angleindex2per;
magrhigh = FL(fabs(highr1[0])) + (FL(fabs(highr2[0])) – FL(fabs(highr1[0]))) *
           angleindex4per;
magr = magrlow + (magrhigh - magrlow) * elevindexhighper;
if(minphase)
{
  logmagr[0] = LOG(magr == FL(0.0) ? FL(0.00000001) : magr);
}
else
{
  if(currentphaser[0] < FL(0.))
    hrtfrfloat[0] = -magr;
  else
    hrtfrfloat[0] = magr;
}

magrlow = FL(fabs(lowr1[1])) + (FL(fabs(lowr2[1])) - FL(fabs(lowr1[1])))  *
           angleindex2per;
magrhigh = FL(fabs(highr1[1])) + (FL(fabs(highr2[1])) - FL(fabs(highr1[1]))) *
            angleindex4per;
magr = magrlow + (magrhigh - magrlow) * elevindexhighper;
if(minphase)
{
  logmagr[1] = LOG(magr == FL(0.0) ? FL(0.00000001) : magr);
}
else
{
  if(currentphaser[1] < FL(0.))
    hrtfrfloat[1] = -magr;
  else
    hrtfrfloat[1] = magr;
}

/* remaining values */
for(i = 2; i < irlength; i += 2)
{
  /* interpolate high and low mags */
  magllow = lowl1[i] + (lowl2[i] – lowl1[i]) * angleindex2per;
  maglhigh = highl1[i] + (highl2[i] – highl1[i]) * angleindex4per;

  magrlow = lowr1[i] + (lowr2[i] – lowr1[i]) * angleindex2per;
  magrhigh = highr1[i] + (highr2[i] – highr1[i]) * angleindex4per;

  /* interpolate high and low results */
  magl = magllow + (maglhigh - magllow) * elevindexhighper;
  magr = magrlow + (magrhigh - magrlow) * elevindexhighper;

  if(phasetrunc)
  {
      /* use current phase, back to rectangular */
      phasel = currentphasel[i + 1];
      phaser = currentphaser[i + 1];

      /* polar to rectangular */
      hrtflfloat[i] = magl * COS(phasel);
      hrtflfloat[i+1] = magl * SIN(phasel);

      hrtfrfloat[i] = magr * COS(phaser);
      hrtfrfloat[i+1] = magr * SIN(phaser);
    }

     if(minphase)
    {
      /* store log magnitudes, 0 phases for ifft, do not allow log(0.0) */
      logmagl[i] = LOG(magl == FL(0.0) ? FL(0.00000001) : magl);
      logmagr[i] = LOG(magr == FL(0.0) ? FL(0.00000001) : magr);

      logmagl[i + 1] = FL(0.0);
      logmagr[i + 1] = FL(0.0);
    }
  }

  if(minphase)
  {
    /* ifft!...see Oppehneim and Schafer for min phase process...based on real
         cepstrum method */
    csound->InverseRealFFT(csound, logmagl, irlength);
    csound->InverseRealFFT(csound, logmagr, irlength);

    /* window, note no need to scale on csound iffts... */
    for(i = 0; i < irlength; i++)
    {
      xhatwinl[i] = logmagl[i] * win[i];
      xhatwinr[i] = logmagr[i] * win[i];
    }

    /* fft */
    csound->RealFFT(csound, xhatwinl, irlength);
    csound->RealFFT(csound, xhatwinr, irlength);

   /* exponential of result */
   /* 0 hz and nyq purely real... */
   expxhatwinl[0] = EXP(xhatwinl[0]);
   expxhatwinl[1] = EXP(xhatwinl[1]);
   expxhatwinr[0] = EXP(xhatwinr[0]);
   expxhatwinr[1] = EXP(xhatwinr[1]);

   /* exponential of real, cos/sin of imag */
   for(i = 2; i < irlength; i += 2)
   {
     expxhatwinl[i] = EXP(xhatwinl[i]) * COS(xhatwinl[i + 1]);
     expxhatwinl[i+1] = EXP(xhatwinl[i]) * SIN(xhatwinl[i + 1]);
     expxhatwinr[i] = EXP(xhatwinr[i]) * COS(xhatwinr[i + 1]);
     expxhatwinr[i+1] = EXP(xhatwinr[i]) * SIN(xhatwinr[i + 1]);
   }

   /* ifft for output buffers */
   csound->InverseRealFFT(csound, expxhatwinl, irlength);
   csound->InverseRealFFT(csound, expxhatwinr, irlength);

   /* output */
   for(i= 0; i < irlength; i++)
   {
     hrtflpad[i] = expxhatwinl[i];
     hrtfrpad[i] = expxhatwinr[i];
   }
 }

 /* use current phase and interped mag directly */
 if(phasetrunc)
```

```
{
  /* ifft */
  csound->InverseRealFFT(csound, hrtflfloat, irlength);
  csound->InverseRealFFT(csound, hrtfrfloat, irlength);

 for (i = 0; i < irlength; i++)
 {
    /* scale and pad buffers with zeros to fftbuff */
    hrtflpad[i] = hrtflfloat[i];
    hrtfrpad[i] = hrtfrfloat[i];
  }
}

/* zero pad impulse */
for(i = irlength; i < irlengthpad; i++)
{
  hrtflpad[i] = FL(0.0);
  hrtfrpad[i] = FL(0.0);
}

/* back to freq domain */
csound->RealFFT(csound, hrtflpad, irlengthpad);
csound->RealFFT(csound, hrtfrpad, irlengthpad);

if(minphase)
{
   /* read delay data: 4 nearest points, as above */
   /* point 1 */
   skip = 0;
   if(angleindex1 > elevationarray[elevindexlow] / 2)
   {
     for (i = 0; i < elevindexlow; i++)
       skip += ((int)(elevationarray[i] / 2) + 1);
    for(i = 0; i < (elevationarray[elevindexlow] - angleindex1); i++)
       skip++;
     delaylow1 = minphasedels[skip];
   }
   else
   {
     for (i = 0; i < elevindexlow; i++)
      skip += ((int)(elevationarray[i] / 2) + 1);
     for(i = 0; i < angleindex1; i++)
       skip++;
     delaylow1 = minphasedels[skip];
   }

   /* point 2 */
   skip = 0;
  if(angleindex2 > elevationarray[elevindexlow] / 2)
  {
     for (i = 0; i < elevindexlow; i++)
       skip += ((int)(elevationarray[i] / 2) + 1);
     for(i = 0; i < (elevationarray[elevindexlow] - angleindex2); i++)
       skip++;
     delaylow2 = minphasedels[skip];
   }
   else
   {
     for (i = 0; i < elevindexlow; i++)
       skip += ((int)(elevationarray[i] / 2) + 1);
     for (i = 0; i < angleindex2; i++)
       skip++;
     delaylow2 = minphasedels[skip];
   }

   /* point 3 */
   skip = 0;
   if(angleindex3 > elevationarray[elevindexhigh] / 2)
   {
     for (i = 0; i < elevindexhigh; i++)
       skip += ((int)(elevationarray[i] / 2) + 1);
      for(i = 0; i < (elevationarray[elevindexhigh] - angleindex3); i++)
        skip++;
      delayhigh1  =minphasedels[skip];
    }
    else
    {
      for (i = 0; i < elevindexhigh; i++)
        skip += ((int)(elevationarray[i] / 2) + 1);
      for (i = 0; i < angleindex3; i++)
        skip++;
      delayhigh1 = minphasedels[skip];
    }

    /* point 4 */
    skip = 0;
    if(angleindex4 > elevationarray[elevindexhigh] / 2)
    {
      for (i = 0; i < elevindexhigh; i++)
        skip += ((int)(elevationarray[i] / 2) + 1);
      for(i = 0; i < (elevationarray[elevindexhigh] – angleindex4); i++)
        skip++;
      delayhigh2 = minphasedels[skip];
    }
    else
    {
      for (i = 0; i < elevindexhigh; i++)
        skip += ((int)(elevationarray[i] / 2) + 1);
      for (i = 0; i < angleindex4; i++)
        skip++;
      delayhigh2 = minphasedels[skip];
    }

    /* delay interp */
    delaylow = delaylow1 + ((delaylow2 – delaylow1) * angleindex2per);
    delayhigh = delayhigh1 + ((delayhigh2 – delayhigh1) * angleindex4per);
    delayfloat = delaylow + ((delayhigh – delaylow) * elevindexhighper);
    p->delayfloat = delayfloat;
  }
   /* end of angle/elev change process */
  p->elevv = elev;
  p->anglev = angle;
}

/* look after overlap add */
for(i = 0; i < overlapsize ; i++)
{
  overlapl[i] = outl[i + irlength];
  overlapr[i] = outr[i + irlength];
  /* look after fade */
  if(phasetrunc)
  {
    if(crossfade)
    {
      overlapoldl[i] = outl[i + irlength];
      overlapoldr[i] = outr[i + irlength];
    }
    /* overlap will be previous fading out signal */
    if(cross)
    {
      overlapoldl[i] = outlold[i + irlength];
      overlapoldr[i] = outrold[i + irlength];
    }
  }
}

/* insert insig */
for (i = 0; i < irlength; i++)
  complexinsig[i] = insig[i];

for (i = irlength; i < irlengthpad; i++)
  complexinsig[i] = FL(0.0);
```

```
csound->RealFFT(csound, complexinsig, irlengthpad);

/* complex mult function... */
csound->RealFFTMult(csound, outspecl, hrtflpad, complexinsig, irlengthpad,
                    FL(1.0));
csound->RealFFTMult(csound, outspecr, hrtfrpad, complexinsig, irlengthpad,
                      FL(1.0));

/* convolution is the inverse FFT of above result */
csound->InverseRealFFT(csound, outspecl, irlengthpad);
csound->InverseRealFFT(csound, outspecr, irlengthpad);

/* real values, scaled (by a little more than usual to ensure no clipping) sr
     related */
for(i = 0; i < irlengthpad; i++)
{
  outl[i] = outspecl[i] / (sr / FL(38000.0));
  outr[i] = outspecr[i] / (sr / FL(38000.0));
}

if(phasetrunc)
{
  /* setup for fades */
  if(crossfade || cross)
  {
    crossout = 1;

    csound->RealFFTMult(csound, outspecoldl, oldhrtflpad, complexinsig,
                        irlengthpad, FL(1.0));
    csound->RealFFTMult(csound, outspecoldr, oldhrtfrpad, complexinsig,
                        irlengthpad, FL(1.0));

   csound->InverseRealFFT(csound, outspecoldl, irlengthpad);
    csound->InverseRealFFT(csound, outspecoldr, irlengthpad);

    /* scaled */
    for(i = 0; i < irlengthpad; i++)
    {
      outlold[i] = outspecoldl[i] / (sr / FL(38000.0));
      outrold[i] = outspecoldr[i] / (sr / FL(38000.0));
    }

    cross++;
    /* number of processing buffers in a fade */
    cross = cross % fade;
  }

  if(crossout)
  {
    /* do fade */
    for(i = 0; i < irlength; i++)
    {
      outl[i] = ((outlold[i] + (i<overlapsize ? overlapoldl[i] : 0)) * (FL(1.0)
                 - FL(l) / fadebuffer)) + (outl[i] + (i < overlapsize ?
                 overlapl[i] : 0)) * FL(l)/fadebuffer);
      outr[i] = ((outrold[i] + (i<overlapsize ? overlapoldr[i] : 0)) * (FL(1.0)
                 - FL(l) / fadebuffer)) + ((outr[i] + (i < overlapsize ?
                 overlapr[i] : 0)) * FL(l)/fadebuffer);
      l++;
    }
  }
  else
  for(i = 0; i < irlength; i++)
  {
    outl[i] = outl[i] + (i < overlapsize ? overlapl[i] : FL(0.0));
    outr[i] = outr[i] + (i < overlapsize ? overlapr[i] : FL(0.0));
  }
}

if(minphase)
```

```
    {
    /* use output direcly and add delay in time domain */
      for(i = 0; i < irlength; i++)
      {
        outl[i] = outl[i] + (i < overlapsize ? overlapl[i] : FL(0.0));
        outr[i] = outr[i] + (i < overlapsize ? overlapr[i] : FL(0.0));
      }

      if(angle > FL(180.0))
      {
        vdtr =  delayfloat * sr;
        vdtl = FL(0.0);
      }
      else
      {
        vdtr = FL(0.0);
        vdtl = delayfloat * sr;
      }

      /* delay right */
      if(vdtr > mdtr)
        vdtr = FL(mdtr);
      for(i = 0; i < irlength; i++)
      {
        rpr = ptr - vdtr;
        rpr = (rpr >= 0 ? (rpr < mdtr ? rpr : rpr - mdtr) : rpr + mdtr);
        posr = (int) rpr;
        fracr = rpr - posr;
         delmemr[ptr] = outr[i];
         outvdr = delmemr[posr] + fracr * (delmemr[(posr + 1 < mdtr ? posr + 1 : 0)]
                 - delmemr[posr]);
         outr[i] = outvdr;
         ptr = (ptr != mdtr - 1 ? ptr + 1 : 0);
      }

      /* delay left */
      if(vdtl > mdtl)
        vdtl = FL(mdtl);
      for(i = 0; i < irlength; i++)
      {
         rpl = ptl - vdtl;
         rpl = (rpl >= 0 ? (rpl < mdtl ? rpl : rpl - mdtl) : rpl + mdtl);
         posl = (int) rpl;
         fracl = rpl - (int) posl;
         delmeml[ptl] = outl[i];
         outvdl =  delmeml[posl] + fracl * (delmeml[(posl + 1 < mdtl ? posl + 1 : 0)]
                   - delmeml[posl]);
         outl[i] = outvdl;
         ptl = (ptl != mdtl - 1 ? ptl + 1 : 0);
      }

      p->ptl = ptl;
      p->ptr = ptr;
    }

    /* reset counter */
    counter = 0;
    if(phasetrunc)
    {
      /* update */
      p->cross = cross;
      p->l = l;
    }

  }   /* end of irlength == counter */

}   /* end of ksmps audio loop */

/* update */
p->counter = counter;
if(phasetrunc)
```

```
     p->initialfade = initialfade;

   return OK;
}

/* Csound hrtf magnitude interpolation, woodworth phase, static source: */
/* overlap add convolution */

/* aleft, aright hrtfstat ain, iang, iel, ifilel, ifiler [,iradius = 8.8, isr =
44100]...options of 48 and 96k sr */

/* see definitions above */

typedef struct
{
  OPDS  h;
  /* outputs and inputs */
  MYFLT *outsigl, *outsigr;
  MYFLT *in, *iangle, *ielev, *ifilel, *ifiler, *oradius, *osr;

  /*see definitions in INIT*/
  int irlength, irlengthpad, overlapsize;
  MYFLT sroverN;

  int counter;
  MYFLT sr;

  /* hrtf data padded */
  AUXCH hrtflpad,hrtfrpad;
  /* in and output buffers */
  AUXCH insig, outl, outr;

  /* memory local to perform method */
  /* insig fft */
  AUXCH complexinsig;
  /* hrtf buffers (rectangular complex form) */
  AUXCH hrtflfloat, hrtfrfloat;
  /* spectral data */
  AUXCH outspecl, outspecr;

  /* overlap data */
  AUXCH overlapl, overlapr;

  /* interpolation buffers */
  AUXCH lowl1, lowr1, lowl2, lowr2, highl1, highr1, highl2, highr2;

  /* buffers for impulse shift */
  AUXCH leftshiftbuffer, rightshiftbuffer;
}
hrtfstat;

static int hrtfstat_init(CSOUND *csound, hrtfstat *p)
{
  /* left and right data files: spectral mag, phase format. */
  MEMFIL *fpl = NULL, *fpr = NULL;
  char filel[MAXNAME], filer[MAXNAME];

  /* interpolation values */
  MYFLT *lowl1;
  MYFLT *lowr1;
  MYFLT *lowl2;
  MYFLT *lowr2;
  MYFLT *highl1;
  MYFLT *highr1;
  MYFLT *highl2;
  MYFLT *highr2;

  MYFLT *hrtflfloat;
  MYFLT *hrtfrfloat;

  MYFLT *hrtflpad;
  MYFLT *hrtfrpad;

  MYFLT elev = *p->ielev;
  MYFLT angle = *p->iangle;
  MYFLT r = *p->oradius;
  MYFLT sr = *p->osr;

  /* pointers into HRTF files */
  float *fpindexl=NULL;
  float *fpindexr=NULL;

  /* time domain impulse length, padded, overlap add */
  int irlength, irlengthpad, overlapsize;

  int i, skip = 0;

  /* local interpolation values */
  MYFLT elevindexhighper, angleindex2per, angleindex4per;
  int elevindexlow, elevindexhigh, angleindex1, angleindex2, angleindex3, angleindex4;
  MYFLT magl, magr, phasel, phaser, magllow, magrlow, maglhigh, magrhigh;

  /* local variables, mainly used for simplification */
  MYFLT elevindexstore;
  MYFLT angleindexlowstore;
  MYFLT angleindexhighstore;

  /* woodworth values */
  MYFLT radianangle, radianelev, itd, itdww, freq;

  /* shift */
  int shift;
  MYFLT *leftshiftbuffer;
  MYFLT *rightshiftbuffer;

  /* sr */
  if(sr != FL(44100.0) && sr != FL(48000.0) && sr != FL(96000.0))
     sr = FL(44100.0);
  p->sr = sr;

  if (UNLIKELY(csound->esr != sr))
      csound->Message(csound, Str("\n\nWARNING!!:\nOrchestra SR not compatible with HRTF
                                   processing SR of: %.0f\n\n"), sr);

  /* setup as per sr */
  if(sr == 44100 || sr == 48000)
  {
    irlength = 128;
    irlengthpad = 256;
    overlapsize = (irlength - 1);
  }
  else if(sr == 96000)
  {
    irlength = 256;
    irlengthpad = 512;
    overlapsize = (irlength - 1);
  }

  /* copy in string name... */
  strcpy(filel, (char*) p->ifilel);
  strcpy(filer, (char*) p->ifiler);

  /* reading files, with byte swap */
  if (UNLIKELY((fpl = csound->ldmemfile2withCB(csound, filel, CSFTYPE_FLOATS_BINARY,
      swap4bytes)) == NULL))
  return
    csound->InitError(csound, Str("\n\n\nCannot load left data file, exiting\n\n"));

  if (UNLIKELY((fpr = csound->ldmemfile2withCB(csound, filer, CSFTYPE_FLOATS_BINARY,
      swap4bytes)) == NULL))
  return
    csound->InitError(csound, Str("\n\n\nCannot load right data file, exiting\n\n"));
```

```
p->irlength = irlength;
p->irlengthpad = irlengthpad;
p->overlapsize = overlapsize;

p->sroverN = sr/irlength;

/* start indices at correct value (start of file)/ zero indices. (don't need to store
   here, as only accessing in INIT) */
fpindexl = (float *) fpl->beginp;
fpindexr = (float *) fpr->beginp;

/* buffers */
if (!p->insig.auxp || p->insig.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->insig);
if (!p->outl.auxp || p->outl.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outl);
if (!p->outr.auxp || p->outr.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outr);
if (!p->hrtflpad.auxp || p->hrtflpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->hrtflpad);
if (!p->hrtfrpad.auxp || p->hrtfrpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->hrtfrpad);
if (!p->complexinsig.auxp || p->complexinsig.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p-> complexinsig);
if (!p->hrtflfloat.auxp || p->hrtflfloat.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->hrtflfloat);
if (!p->hrtfrfloat.auxp || p->hrtfrfloat.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->hrtfrfloat);
if (!p->outspecl.auxp || p->outspecl.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outspecl);
if (!p->outspecr.auxp || p->outspecr.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad*sizeof(MYFLT), &p->outspecr);
if (!p->overlapl.auxp || p->overlapl.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize*sizeof(MYFLT), &p->overlapl);
if (!p->overlapr.auxp || p->overlapr.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize*sizeof(MYFLT), &p->overlapr);

memset(p->insig.auxp, 0, irlength * sizeof(MYFLT));
memset(p->outl.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outr.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtflpad.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtfrpad.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->complexinsig.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtflfloat.auxp, 0, irlength * sizeof(MYFLT));
memset(p->hrtfrfloat.auxp, 0, irlength * sizeof(MYFLT));
memset(p->outspecl.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outspecr.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->overlapl.auxp, 0, overlapsize * sizeof(MYFLT));
memset(p->overlapr.auxp, 0, overlapsize * sizeof(MYFLT));

  /* interpolation values */
if (!p->lowl1.auxp || p->lowl1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowl1);
if (!p->lowr1.auxp || p->lowr1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowr1);
if (!p->lowl2.auxp || p->lowl2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowl2);
if (!p->lowr2.auxp || p->lowr2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowr2);
if (!p->highl1.auxp || p->highl1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highl1);
if (!p->highr1.auxp || p->highr1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highr1);
if (!p->highl2.auxp || p->highl2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highl2);
if (!p->highr2.auxp || p->highr2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highr2);

/* best to zero, for future changes (filled in init) */
memset(p->lowl1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowr1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowl2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowr2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highl1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highl2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highr1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highr2.auxp, 0, irlength * sizeof(MYFLT));

/* shift buffers */
if (!p->leftshiftbuffer.auxp || p->leftshiftbuffer.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->leftshiftbuffer);
if (!p->rightshiftbuffer.auxp || p->rightshiftbuffer.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength*sizeof(MYFLT), &p->rightshiftbuffer);

memset(p->leftshiftbuffer.auxp, 0, irlength * sizeof(MYFLT));
memset(p->rightshiftbuffer.auxp, 0, irlength * sizeof(MYFLT));

lowl1 = (MYFLT *)p->lowl1.auxp;
lowr1 = (MYFLT *)p->lowr1.auxp;
lowl2 = (MYFLT *)p->lowl2.auxp;
lowr2 = (MYFLT *)p->lowr2.auxp;
highl1 = (MYFLT *)p->highl1.auxp;
highr1 = (MYFLT *)p->highr1.auxp;
highl2 = (MYFLT *)p->highl2.auxp;
highr2 = (MYFLT *)p->highr2.auxp;

leftshiftbuffer = (MYFLT *)p->leftshiftbuffer.auxp;
rightshiftbuffer = (MYFLT *)p->rightshiftbuffer.auxp;

hrtflfloat = (MYFLT *)p->hrtflfloat.auxp;
hrtfrfloat = (MYFLT *)p->hrtfrfloat.auxp;

hrtflpad = (MYFLT *)p->hrtflpad.auxp;
hrtfrpad = (MYFLT *)p->hrtfrpad.auxp;

if(r <= 0 || r > 15)
  r = FL(8.8);

if(elev > FL(90.0))
  elev = FL(90.0);
if(elev < FL(-40.0))
  elev = FL(-40.0);

while(angle < FL(0.0))
  angle += FL(360.0);
while(angle >= FL(360.0))
  angle -= FL(360.0);

/* two nearest elev indices to avoid recalculating */
elevindexstore = (elev - minelev) / elevincrement;
elevindexlow = (int)elevindexstore;

if(elevindexlow < 13)
  elevindexhigh = elevindexlow + 1;
/* highest index reached */
else
  elevindexhigh = elevindexlow;

/* get percentage value for interpolation */
elevindexhighper = elevindexstore - elevindexlow;

/* avoid recalculation */
angleindexlowstore = angle / (FL(360.) / elevationarray[elevindexlow]);
angleindexhighstore = angle / (FL(360.) / elevationarray[elevindexhigh]);

/* 4 closest indices, 2 low and 2 high */
angleindex1 = (int)angleindexlowstore;

angleindex2 = angleindex1 + 1;
angleindex2 = angleindex2 % elevationarray[elevindexlow];
```

```
angleindex3 = (int)angleindexhighstore;

angleindex4 = angleindex3 + 1;
angleindex4 = angleindex4 % elevationarray[elevindexhigh];

/* angle percentages for interp */
angleindex2per = angleindexlowstore - angleindex1;
angleindex4per = angleindexhighstore - angleindex3;

/* read 4 nearest HRTFs */
skip = 0;
/* switch l and r */
if(angleindex1 > elevationarray[elevindexlow] / 2)
{
  for(i = 0; i < elevindexlow; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexlow] – angleindex1); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = fpindexr[skip + i];
    lowr1[i] = fpindexl[skip + i];
  }
}
else
{
  for(i = 0; i < elevindexlow; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex1; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = fpindexl[skip + i];
    lowr1[i] = fpindexr[skip + i];
  }
}

skip = 0;
if(angleindex2 > elevationarray[elevindexlow] / 2)
{
  for(i = 0; i < elevindexlow; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexlow] – angleindex2); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = fpindexr[skip + i];
    lowr2[i] = fpindexl[skip + i];
  }
}
else
{
  for(i = 0; i < elevindexlow; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex2; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = fpindexl[skip + i];
    lowr2[i] = fpindexr[skip + i];
  }
}

skip = 0;
if(angleindex3 > elevationarray[elevindexhigh] / 2)
{
  for(i = 0; i < elevindexhigh; i++)
    skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexhigh] – angleindex3); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl1[i] = fpindexr[skip + i];
    highr1[i] = fpindexl[skip + i];
  }
}
else
{
  for(i = 0; i < elevindexhigh; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex3; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl1[i] = fpindexl[skip + i];
    highr1[i] = fpindexr[skip + i];
  }
}

skip = 0;
if(angleindex4 > elevationarray[elevindexhigh] / 2)
{
  for(i = 0; i < elevindexhigh; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexhigh] – angleindex4); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl2[i] = fpindexr[skip + i];
    highr2[i] = fpindexl[skip + i];
  }
}
else
{
  for(i = 0; i < elevindexhigh; i++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex4; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl2[i] = fpindexl[skip + i];
    highr2[i] = fpindexr[skip + i];
  }
}

/* woodworth process */
/* ITD formula, check which ear is relevant to calculate angle from */
if(angle > FL(180.))
  radianangle = (angle - FL(180.)) * FL(PI / 180.);
else
  radianangle = angle * FL(PI / 180.);
/* degrees to radians */
  radianelev = elev * FL(PI / 180.);

/* get in correct range for formula */
if(radianangle > FL(PI / 2.0))
  radianangle = FL(PI) - radianangle;

/* woodworth formula for itd */
itdww = (radianangle + sinf(radianangle)) * r * cosf(radianelev) / FL(c);

/* 0 Hz and Nyq... */
/* these are real values...may be neg (implying phase of pi: in phase truncation),
   so need fabs... */
magllow = FL(fabs(lowl1[0])) + (FL(fabs(lowl2[0])) – FL(fabs(lowl1[0]))) *
          angleindex2per;
maglhigh = FL(fabs(highl1[0])) + (FL(fabs(highl2[0])) – FL(fabs(highl1[0]))) *
           angleindex4per;
hrtflfloat[0] = magllow + (maglhigh - magllow) * elevindexhighper;

magllow = FL(fabs(lowl1[1])) + (FL(fabs(lowl2[1])) – FL(fabs(lowl1[1]))) *
          angleindex2per;
```

```
maglhigh = FL(fabs(highl1[1])) + (FL(fabs(highl2[1])) – FL(fabs(highl1[1]))) *
           angleindex4per;
hrtflfloat[1] = magllow + (maglhigh - magllow) * elevindexhighper;

magrlow = FL(fabs(lowr1[0])) + (FL(fabs(lowr2[0])) – FL(fabs(lowr1[0]))) *
          angleindex2per;
magrhigh = FL(fabs(highr1[0])) + (FL(fabs(highr2[0])) – FL(fabs(highr1[0]))) *
           angleindex4per;
hrtfrfloat[0] = magrlow + (magrhigh - magrlow) * elevindexhighper;

magrlow = FL(fabs(lowr1[1])) + (FL(fabs(lowr2[1])) – FL(fabs(lowr1[1]))) *
          angleindex2per;
magrhigh = FL(fabs(highr1[1])) + (FL(fabs(highr2[1])) – FL(fabs(highr1[1]))) *
           angleindex4per;
hrtfrfloat[1] = magrlow + (magrhigh - magrlow) * elevindexhighper;

/* magnitude interpolation */
for(i = 2; i < irlength; i+=2)
{
  /* interpolate high and low mags */
  magllow = lowl1[i] + (lowl2[i] - lowl1[i]) * angleindex2per;
  maglhigh = highl1[i]+(highl2[i] - highl1[i]) * angleindex4per;

  magrlow = lowr1[i] + (lowr2[i] - lowr1[i]) * angleindex2per;
  magrhigh = highr1[i] + (highr2[i] - highr1[i]) * angleindex4per;

  /* interpolate high and low results */
  magl = magllow + (maglhigh - magllow) * elevindexhighper;
  magr = magrlow + (magrhigh - magrlow) * elevindexhighper;

  freq = (i / 2) * p->sroverN;

  /* non linear itd...last value in array = 1.0, so back to itdww */
  if(p->sr == 96000)
  {
    if ((i / 2) < 6)
      itd = itdww * nonlinitd96k[(i / 2) - 1];
  }
  if(p->sr == 48000)
  {
    if ((i / 2) < 6)
      itd = itdww * nonlinitd48k[(i / 2) - 1];
  }
  if(p->sr == 44100)
  {
    if((i / 2) < 6)
      itd = itdww * nonlinitd[(i / 2) - 1];
  }

  if(angle > FL(180.))
  {
    phasel = TWOPI_F * freq * (itd / 2);
    phaser = TWOPI_F * freq * -(itd / 2);
  }
  else
  {
    phasel = TWOPI_F * freq * -(itd / 2);
    phaser = TWOPI_F * freq * (itd / 2);
  }

  /* polar to rectangular */
  hrtflfloat[i] = magl * COS(phasel);
  hrtflfloat[i+1] = magl * SIN(phasel);

  hrtfrfloat[i] = magr * COS(phaser);
  hrtfrfloat[i+1] = magr * SIN(phaser);
}

/* ifft */
csound->InverseRealFFT(csound, hrtflfloat, irlength);
csound->InverseRealFFT(csound, hrtfrfloat, irlength);
```

```
  for (i = 0; i < irlength; i++)
  {
    /* scale and pad buffers with zeros to fftbuff */
    leftshiftbuffer[i] = hrtflfloat[i];
    rightshiftbuffer[i] = hrtfrfloat[i];
  }

  /* shift for causality...impulse as is is centred around zero time lag...then phase
     added. */
  /* this step centres impulse around centre tap of filter (then phase moves it for
     correct itd...) */
  shift = irlength / 2;

  for(i = 0; i < irlength; i++)
  {
    hrtflpad[i] = leftshiftbuffer[shift];
    hrtfrpad[i] = rightshiftbuffer[shift];

    shift++;
    shift = shift % irlength;
  }

  /* zero pad impulse */
  for(i = irlength; i < irlengthpad; i++)
  {
    hrtflpad[i] = FL(0.0);
    hrtfrpad[i] = FL(0.0);
  }

  /* back to freq domain */
  csound->RealFFT(csound, hrtflpad, irlengthpad);
  csound->RealFFT(csound, hrtfrpad, irlengthpad);

  /* initialize counter */
  p->counter = 0;

  return OK;
}


static int hrtfstat_process(CSOUND *csound, hrtfstat *p)
{
  /* local pointers to p */
  MYFLT *in = p->in;
  MYFLT *outsigl  = p->outsigl;
  MYFLT *outsigr = p->outsigr;

  /* common buffers and variables */
  MYFLT *insig = (MYFLT *)p->insig.auxp;
  MYFLT *outl = (MYFLT *)p->outl.auxp;
  MYFLT *outr = (MYFLT *)p->outr.auxp;

  MYFLT *hrtflpad = (MYFLT *)p->hrtflpad.auxp;
  MYFLT *hrtfrpad = (MYFLT *)p->hrtfrpad.auxp;

  MYFLT *complexinsig = (MYFLT *)p->complexinsig.auxp;
  MYFLT *outspecl = (MYFLT *)p->outspecl.auxp;
  MYFLT *outspecr = (MYFLT *)p->outspecr.auxp;

  MYFLT *overlapl = (MYFLT *)p->overlapl.auxp;
  MYFLT *overlapr = (MYFLT *)p->overlapr.auxp;

  int counter = p->counter;
  int n, j, i;

  int irlength = p->irlength;
  int irlengthpad = p->irlengthpad;
  int overlapsize = p->overlapsize;

  MYFLT sr = p->sr;
  n = csound->ksmps;

  for(j = 0; j < n; j++)
  {
    /* ins and outs */
    insig[counter] = in[j];

    outsigl[j] = outl[counter];
    outsigr[j] = outr[counter];

    counter++;

    if(counter == irlength)
    {
      /* process a block */
      /* look after overlap add stuff */
      for(i = 0; i < overlapsize ; i++)
      {
        overlapl[i] = outl[i+irlength];
        overlapr[i] = outr[i+irlength];
      }

      /* insert insig for complex real,im fft, zero pad */
      for (i = 0; i <  irlength; i++)
        complexinsig[i] = insig[i];

      for (i = irlength; i <  irlengthpad; i++)
        complexinsig[i] = FL(0.0);

      csound->RealFFT(csound, complexinsig, irlengthpad);

      /* complex multiplication */
      csound->RealFFTMult(csound, outspecl, hrtflpad, complexinsig, irlengthpad,
                          FL(1.0));
      csound->RealFFTMult(csound, outspecr, hrtfrpad, complexinsig, irlengthpad,
                          FL(1.0));

      /* convolution is the inverse FFT of above result */
      csound->InverseRealFFT(csound, outspecl, irlengthpad);
      csound->InverseRealFFT(csound, outspecr, irlengthpad);

      /* scaled by a factor related to sr...? */
      for(i = 0; i < irlengthpad; i++)
      {
        outl[i] = outspecl[i] / (sr / FL(38000.0));
        outr[i] = outspecr[i] / (sr / FL(38000.0));
      }

      for(i = 0; i < irlength; i++)
      {
        outl[i] = outl[i] + (i < overlapsize ? overlapl[i] : FL(0.0));
        outr[i] = outr[i] + (i < overlapsize ? overlapr[i] : FL(0.0));
      }

      /* reset counter */
      counter = 0;

    }     /* end of irlength == counter */

  }   /* end of ksmps audio loop */

  /* update */
  p->counter = counter;

  return OK;
}

/* Csound hrtf magnitude interpolation, dynamic woodworth trajectory */
/* stft from fft.cpp in sndobj... */
/* stft based on sndobj implementation...some notes: */
/* using an overlapskip (same as m_counter) for in and out to control seperately for
   clarity... */

/* aleft, aright hrtfmove2 ain, kang, kel, ifilel, ifiler [, ioverlap = 4, iradius =
   8.8, isr = 44100] */
/* ioverlap is stft overlap, iradius is head radius, sr can also be 48000 and 96000 */

typedef struct
{
  OPDS  h;
  /* outputs and inputs */
  MYFLT *outsigl, *outsigr;
  MYFLT *in, *kangle, *kelev, *ifilel, *ifiler, *ooverlap, *oradius, *osr;

  /* check if relative source has changed! */
  MYFLT anglev, elevv;

  /* see definitions in INIT */
  int irlength;
  MYFLT sroverN;
  MYFLT sr;

  /* test inputs in init, get accepted value/default, and store in variables below. */
  int overlap;
  MYFLT radius;

  int hopsize;

  float *fpbeginl,*fpbeginr;

  /* to keep track of process */
  int counter, t;

  /* in and output buffers */
  AUXCH inbuf;
  AUXCH outbufl, outbufr;

  /* memory local to perform method */
  /* insig fft */
  AUXCH complexinsig;
  /* hrtf buffers (rectangular complex form) */
  AUXCH hrtflfloat, hrtfrfloat;
  /* spectral data */
  AUXCH outspecl, outspecr;

  /* interpolation buffers */
  AUXCH lowl1,lowr1,lowl2,lowr2,highl1,highr1,highl2,highr2;

  /* stft window */
  AUXCH win;
  /* used for skipping into next stft array on way in and out */
  AUXCH overlapskipin, overlapskipout;

}
hrtfmove2;

static int hrtfmove2_init(CSOUND *csound, hrtfmove2 *p)
{
  /* left and right data files: spectral mag, phase format. */
  MEMFIL *fpl = NULL, *fpr = NULL;

  char filel[MAXNAME], filer[MAXNAME];

  /* time domain impulse length */
  int irlength;

  /* stft window */
  MYFLT *win;
  /* overlap skip buffers */
  int *overlapskipin, *overlapskipout;
```

```
MYFLT *inbuf;
MYFLT *outbufl, *outbufr;

int overlap = (int)*p->ooverlap;
MYFLT r = *p->oradius;
MYFLT sr = *p->osr;

int i = 0;

if(sr != 44100 && sr != 48000 && sr != 96000)
sr = 44100;
p->sr = sr;

if (UNLIKELY(csound->esr != sr))
csound->Message(csound, Str("\n\nWARNING!!:\nOrchestra SR not compatible with HRTF
                processing SR of: %.0f\n\n"), sr);

/* setup as per sr */
if(sr == 44100 || sr == 48000)
  irlength = 128;
else if(sr == 96000)
irlength = 256;

/* copy in string name... */
strcpy(filel, (char*) p->ifilel);
strcpy(filer, (char*) p->ifiler);

/* reading files, with byte swap */
if (UNLIKELY((fpl = csound->ldmemfile2withCB(csound, filel, CSFTYPE_FLOATS_BINARY,
    swap4bytes)) == NULL))
return
  csound->InitError(csound, Str("\n\n\nCannot load left data file, exiting\n\n"));

if (UNLIKELY((fpr = csound->ldmemfile2withCB(csound, filer, CSFTYPE_FLOATS_BINARY,
    swap4bytes)) == NULL))
return
  csound->InitError(csound, Str("\n\n\nCannot load right data file, exiting\n\n"));

p->irlength = irlength;
p->sroverN = sr / irlength;

/* file handles */
p->fpbeginl = (float *) fpl->beginp;
p->fpbeginr = (float *) fpr->beginp;

if(overlap != 2 && overlap != 4 && overlap != 8 && overlap != 16)
  overlap = 4;
p->overlap = overlap;

if(r <= 0 || r > 15)
  r = FL(8.8);
p->radius = r;

p->hopsize = (int)(irlength / overlap);

/* buffers */
if (!p->inbuf.auxp || p->inbuf.size < (overlap * irlength) * sizeof(MYFLT))
  csound->AuxAlloc(csound, (overlap * irlength) * sizeof(MYFLT), &p->inbuf);
/* 2d arrays in 1d! */
if (!p->outbufl.auxp || p->outbufl.size < (overlap * irlength) * sizeof(MYFLT))
  csound->AuxAlloc(csound, (overlap * irlength) * sizeof(MYFLT), &p->outbufl);
if (!p->outbufr.auxp || p->outbufr.size < (overlap * irlength) * sizeof(MYFLT))
  csound->AuxAlloc(csound, (overlap * irlength) * sizeof(MYFLT), &p->outbufr);
  if (!p->complexinsig.auxp || p->complexinsig.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p-> complexinsig);
if (!p->hrtflfloat.auxp || p->hrtflfloat.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->hrtflfloat);
if (!p->hrtfrfloat.auxp || p->hrtfrfloat.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->hrtfrfloat);
if (!p->outspecl.auxp || p->outspecl.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->outspecl);
```

```
if (!p->outspecr.auxp || p->outspecr.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->outspecr);

memset(p->inbuf.auxp, 0, (overlap*irlength) * sizeof(MYFLT));
memset(p->outbufl.auxp, 0, (overlap*irlength) * sizeof(MYFLT));
memset(p->outbufr.auxp, 0, (overlap*irlength) * sizeof(MYFLT));
memset(p->complexinsig.auxp, 0, irlength * sizeof(MYFLT));
memset(p->hrtflfloat.auxp, 0, irlength * sizeof(MYFLT));
memset(p->hrtfrfloat.auxp, 0, irlength * sizeof(MYFLT));
memset(p->outspecl.auxp, 0, irlength * sizeof(MYFLT));
memset(p->outspecr.auxp, 0, irlength * sizeof(MYFLT));

/* interpolation values */
if (!p->lowl1.auxp || p->lowl1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowl1);
if (!p->lowr1.auxp || p->lowr1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowr1);
if (!p->lowl2.auxp || p->lowl2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowl2);
if (!p->lowr2.auxp || p->lowr2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowr2);
if (!p->highl1.auxp || p->highl1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highl1);
if (!p->highr1.auxp || p->highr1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highr1);
if (!p->highl2.auxp || p->highl2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highl2);
if (!p->highr2.auxp || p->highr2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highr2);

memset(p->lowl1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowr1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowl2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowr2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highl1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highl2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highr1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highr2.auxp, 0, irlength * sizeof(MYFLT));

if (!p->win.auxp || p->win.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->win);
if (!p->overlapskipin.auxp || p->overlapskipin.size < overlap * sizeof(int))
  csound->AuxAlloc(csound, overlap * sizeof(int), &p->overlapskipin);
if (!p->overlapskipout.auxp || p->overlapskipout.size < overlap * sizeof(int))
  csound->AuxAlloc(csound, overlap * sizeof(int), &p->overlapskipout);

memset(p->win.auxp, 0, irlength * sizeof(MYFLT));
memset(p->overlapskipin.auxp, 0, overlap * sizeof(int));
memset(p->overlapskipout.auxp, 0, overlap * sizeof(int));

win = (MYFLT *)p->win.auxp;
overlapskipin = (int *)p->overlapskipin.auxp;
overlapskipout = (int *)p->overlapskipout.auxp;
inbuf = (MYFLT *)p->inbuf.auxp;
outbufl = (MYFLT *)p->outbufl.auxp;
outbufr = (MYFLT *)p->outbufr.auxp;

/* window is hanning */
for(i = 0; i < irlength; i++)
  win[i] = FL(0.5) - FL((0.5 * cos(i * TWOPI / FL(irlength – 1))));

for(i = 0; i < overlap; i++)
{
  /* so, for example in overlap 4: will be 0, 32, 64, 96 if ir = 128 */
  overlapskipin[i] = p->hopsize * i;
  overlapskipout[i] = p->hopsize * i;
}

/* initialise */
p->counter = 0;
p->t = 0;
```

```
  /* setup values used to check if src has moved, illegal values to start with to ensure
     first read */
  p->anglev = -1;
  p->elevv = -41;

  return OK;
}

static int hrtfmove2_process(CSOUND *csound, hrtfmove2 *p)
{
  /* local pointers to p */
  MYFLT *in = p->in;
  MYFLT *outsigl  = p->outsigl;
  MYFLT *outsigr = p->outsigr;

  /* common buffers and variables */
  MYFLT *inbuf = (MYFLT *)p->inbuf.auxp;

  MYFLT *outbufl = (MYFLT *)p->outbufl.auxp;
  MYFLT *outbufr = (MYFLT *)p->outbufr.auxp;

  MYFLT outsuml = FL(0.0), outsumr = FL(0.0);

  MYFLT *complexinsig = (MYFLT *)p->complexinsig.auxp;
  MYFLT *hrtflfloat = (MYFLT *)p->hrtflfloat.auxp;
  MYFLT *hrtfrfloat = (MYFLT *)p->hrtfrfloat.auxp;
  MYFLT *outspecl = (MYFLT *)p->outspecl.auxp;
  MYFLT *outspecr = (MYFLT *)p->outspecr.auxp;

  MYFLT elev = *p->kelev;
  MYFLT angle = *p->kangle;
  int overlap = p->overlap;
  MYFLT r = p->radius;

  MYFLT sr = p->sr;
  MYFLT sroverN = p->sroverN;

  int hopsize = p->hopsize;

  MYFLT *win = (MYFLT *)p->win.auxp;
  int *overlapskipin = (int *)p->overlapskipin.auxp;
  int *overlapskipout = (int *)p->overlapskipout.auxp;

  int counter = p ->counter;
  int t = p ->t;
  int n;

  /* pointers into HRTF files */
  float *fpindexl;
  float *fpindexr;

  int i, j, skip = 0;

  /* interpolation values */
  MYFLT *lowl1 = (MYFLT *)p->lowl1.auxp;
  MYFLT *lowr1 = (MYFLT *)p->lowr1.auxp;
  MYFLT *lowl2 = (MYFLT *)p->lowl2.auxp;
  MYFLT *lowr2 = (MYFLT *)p->lowr2.auxp;
  MYFLT *highl1 = (MYFLT *)p->highl1.auxp;
  MYFLT *highr1 = (MYFLT *)p->highr1.auxp;
  MYFLT *highl2 = (MYFLT *)p->highl2.auxp;
  MYFLT *highr2 = (MYFLT *)p->highr2.auxp;

  /* local interpolation values */
  MYFLT elevindexhighper, angleindex2per, angleindex4per;
  int elevindexlow, elevindexhigh, angleindex1, angleindex2, angleindex3, angleindex4;
  MYFLT magl, magr, phasel, phaser, magllow, magrlow, maglhigh, magrhigh;

  /* woodworth values */
  MYFLT radianangle, radianelev, itd, itdww, freq;
```

```
int irlength = p->irlength;

/* local variables, mainly used for simplification */
MYFLT elevindexstore;
MYFLT angleindexlowstore;
MYFLT angleindexhighstore;


/* start indices at correct value (start of file)/ zero indices.
      */
fpindexl = (float *) p->fpbeginl;
fpindexr = (float *) p->fpbeginr;

n = csound->ksmps;

/* ksmps loop */
for(j = 0; j < n; j++)
{
  /* distribute the signal and apply the window */
  /* according to a time pointer (kept by overlapskip[n]) */
  for(i = 0; i < overlap; i++)
  {
    inbuf[(i * irlength) + overlapskipin[i]] = in[j] * win[overlapskipin[i]];
    overlapskipin[i]++;
  }

  counter++;

  if(counter == hopsize)
  {
    /* process a block */
    if(elev > FL(90.0))
      elev = FL(90.0);
    if(elev < FL(-40.0))
      elev = FL(-40.0);

    while(angle < FL(0.0))
      angle += FL(360.0);
    while(angle >= FL(360.0))
      angle -= FL(360.0);

    if(angle != p->anglev || elev != p->elevv)
    {
      /* two nearest elev indices to avoid recalculating */
      elevindexstore = (elev - minelev) / elevincrement;
      elevindexlow = (int)elevindexstore;

      if(elevindexlow < 13)
        elevindexhigh = elevindexlow + 1;
      /* highest index reached */
      else
        elevindexhigh = elevindexlow;

      /* get percentage value for interpolation */
      elevindexhighper = elevindexstore – elevindexlow;

      /* avoid recalculation */
      angleindexlowstore = angle / (FL(360.) / elevationarray[elevindexlow]);
      angleindexhighstore = angle / (FL(360.) / elevationarray[elevindexhigh]);

      /* 4 closest indices, 2 low and 2 high */
      angleindex1 = (int)angleindexlowstore;

      angleindex2 = angleindex1 + 1;
      angleindex2 = angleindex2 % elevationarray[elevindexlow];

      angleindex3 = (int)angleindexhighstore;

      angleindex4 = angleindex3 + 1;
      angleindex4 = angleindex4 % elevationarray[elevindexhigh];
```

```
/* angle percentages for interp */
angleindex2per = angleindexlowstore – angleindex1;
angleindex4per = angleindexhighstore – angleindex3;

/* read 4 nearest HRTFs */
skip = 0;
/* switch l and r */
if(angleindex1>elevationarray[elevindexlow]/2)
{
  for(i = 0; i < elevindexlow; i++)
    skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexlow] – angleindex1); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = fpindexr[skip + i];
    lowr1[i] = fpindexl[skip + i];
  }
}
else
{
  for(i = 0; i < elevindexlow; i++)
    skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex1; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = fpindexl[skip + i];
    lowr1[i] = fpindexr[skip + i];
  }
}

skip = 0;
if(angleindex2>elevationarray[elevindexlow]/2)
{
  for(i = 0; i < elevindexlow; i++)
    skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexlow] – angleindex2); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = fpindexr[skip + i];
    lowr2[i] = fpindexl[skip + i];
  }
}
else
{
  for(i = 0; i < elevindexlow; i++)
   skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex2; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = fpindexl[skip + i];
    lowr2[i] = fpindexr[skip + i];
  }
}

skip = 0;
if(angleindex3>elevationarray[elevindexhigh]/2)
{
  for(i = 0; i < elevindexhigh; i++)
    skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexhigh] - angleindex3); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl1[i] = fpindexr[skip + i];
    highr1[i] = fpindexl[skip + i];
  }
}
else
{
  for(i = 0; i < elevindexhigh; i++)
    skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex3; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl1[i] = fpindexl[skip + i];
    highr1[i] = fpindexr[skip + i];
  }
}

skip = 0;
if(angleindex4>elevationarray[elevindexhigh]/2)
{
  for(i = 0; i < elevindexhigh; i++)
    skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexhigh] - angleindex4); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl2[i] = fpindexr[skip + i];
    highr2[i] = fpindexl[skip + i];
  }
}
else
{
  for(i = 0; i < elevindexhigh; i++)
    skip += ((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex4; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl2[i] = fpindexl[skip + i];
    highr2[i] = fpindexr[skip + i];
  }
}

/* woodworth process */
/* ITD formula, check which ear is relevant to  calculate angle from */
/* degrees to radians */
if(angle > FL(180.))
  radianangle = (angle - FL(180.0)) * FL(PI / 180.0);
else
  radianangle = angle * FL(PI / 180.0);
radianelev = elev * FL(PI / 180.0);

/* get in correct range for formula */
if(radianangle > (PI / 2.0))
  radianangle = FL(PI) - radianangle;

/* woodworth formula for itd */
itdww = (radianangle + sinf(radianangle)) * r * cosf(radianelev) / FL(c);

/* 0 Hz and Nyq... */
/* need fabs() here, to get mag, as 0hz and nyq stored as a real value, to allow
   for possible negative, implying phase of pi (in phase truncation) */
magllow = FL(fabs(lowl1[0])) + (FL(fabs(lowl2[0])) - FL(fabs(lowl1[0]))) *
          angleindex2per;
maglhigh = FL(fabs(highl1[0])) + (FL(fabs(highl2[0])) – FL(fabs(highl1[0]))) *
           angleindex4per;
hrtflfloat[0] = magllow + (maglhigh - magllow) * elevindexhighper;

magllow = FL(fabs(lowl1[1])) + (FL(fabs(lowl2[1])) - FL(fabs(lowl1[1]))) *
          angleindex2per;
maglhigh = FL(fabs(highl1[1])) + (FL(fabs(highl2[1])) - FL(fabs(highl1[1]))) *
           angleindex4per;
hrtflfloat[1] = magllow + (maglhigh - magllow) * elevindexhighper;
```

```
  magrlow = FL(fabs(lowr1[0])) + (FL(fabs(lowr2[0])) - FL(fabs(lowr1[0]))) *
            angleindex2per;
  magrhigh = FL(fabs(highr1[0])) + (FL(fabs(highr2[0])) – FL(fabs(highr1[0]))) *
             angleindex4per;
  hrtfrfloat[0] = magrlow + (magrhigh - magrlow) * elevindexhighper;

  magrlow = FL(fabs(lowr1[1])) + (FL(fabs(lowr2[1])) - FL(fabs(lowr1[1]))) *
            angleindex2per;
  magrhigh = FL(fabs(highr1[1])) + (FL(fabs(highr2[1])) – FL(fabs(highr1[1]))) *
             angleindex4per;
  hrtfrfloat[1] = magrlow + (magrhigh - magrlow) * elevindexhighper;

  /* magnitude interpolation */
  for(i = 2; i < irlength; i += 2)
  {
    /* interpolate high and low mags */
    magllow = lowl1[i] + (lowl2[i] – lowl1[i]) * angleindex2per;
    maglhigh = highl1[i] + (highl2[i] – highl1[i]) * angleindex4per;

    magrlow = lowr1[i] + (lowr2[i] – lowr1[i]) * angleindex2per;
    magrhigh = highr1[i] + (highr2[i] – highr1[i]) * angleindex4per;

    /* interpolate high and low results */
    magl = magllow + (maglhigh - magllow) * elevindexhighper;
    magr = magrlow + (magrhigh - magrlow) * elevindexhighper;

    freq = (i / 2) * sroverN;

    /* non linear itd...last value in array = 1.0, so back to itdww */
    if(p->sr == 96000)
    {
      if ((i / 2) < 6)
        itd = itdww * nonlinitd96k[(i / 2) - 1];
    }
    if(p->sr == 48000)
    {
      if ((i / 2) < 6)
        itd = itdww * nonlinitd48k[(i / 2) - 1];
    }
    if(p->sr == 44100)
    {
      if((i / 2) < 6)
        itd = itdww * nonlinitd[(i / 2) - 1];
    }

    if(angle > FL(180.))
    {
      phasel=TWOPI_F * freq * (itd/2);
      phaser=TWOPI_F * freq * -(itd/2);}
    else
    {
      phasel=TWOPI_F * freq * -(itd/2);
      phaser=TWOPI_F * freq * (itd/2);
    }

    /* polar to rectangular */
    hrtflfloat[i] = magl * COS(phasel);
    hrtflfloat[i+1] = magl * SIN(phasel);

    hrtfrfloat[i] = magr * COS(phaser);
    hrtfrfloat[i+1] = magr * SIN(phaser);
  }

  p->elevv = elev;
  p->anglev = angle;
}

/* t used to read inbuf...*/
t--;
if(t < 0)
  t = overlap - 1;
```

```
      /* insert insig for complex real, im fft */
      for(i = 0; i < irlength; i++)
        complexinsig[i] = inbuf[(t * irlength) + i];

      /* zero the current input sigframe time pointer */
      overlapskipin[t] = 0;

      csound->RealFFT(csound, complexinsig, irlength);

      csound->RealFFTMult(csound, outspecl, hrtflfloat, complexinsig, irlength,FL(1.0));
      csound->RealFFTMult(csound, outspecr, hrtfrfloat, complexinsig, irlength,FL(1.0));

      /* convolution is the inverse FFT of above result */
      csound->InverseRealFFT(csound, outspecl, irlength);
      csound->InverseRealFFT(csound, outspecr, irlength);

      /* need scaling based on overlap (more overlaps -> louder) and sr... */
      for(i = 0; i < irlength; i++)
      {
        outbufl[(t * irlength) + i] = outspecl[i] / (overlap * FL(0.5) * (sr /
          FL(44100.0)));
        outbufr[(t * irlength) + i] =  outspecr[i] / (overlap * FL(0.5) * (sr /
          FL(44100.0)));
      }

    }       /* end of !counter % hopsize */

    /* output = sum of all relevant outputs: eg if overlap = 4 and counter = 0, */
    /* outsigl[j] = outbufl[0] + outbufl[128 + 96] + outbufl[256 + 64] + outbufl[384 +
       32]; */
    /*         * * * * [ ]         + */
    /*           * * * [*]         + */
    /*             * * [*] *       + */
    /*               * [*] * *     = */
    /* stft! */

    outsuml = outsumr = FL(0.0);

    for(i = 0; i < (int)overlap; i++)
    {
      outsuml += outbufl[(i * irlength) + overlapskipout[i]] * win[overlapskipout[i]];
      outsumr += outbufr[(i * irlength) + overlapskipout[i]] * win[overlapskipout[i]];
      overlapskipout[i]++;
    }

    if(counter == hopsize)
    {
      /* zero output incrementation... */
      /* last buffer will have gone from 96 to 127...then 2nd last will have gone from
         64 to 127... */
      overlapskipout[t] = 0;
      counter = 0;
    }

    outsigl[j] = outsuml;
    outsigr[j] = outsumr;

  }   /* end of ksmps audio loop */

  /* update */
  p->t = t;
  p->counter = counter;

  return OK;
}

/* see csound manual (extending csound) for details of below */
static OENTRY localops[] =
{
  { "hrtfmove", sizeof(hrtfmove),5, "aa", "akkSSooo",
    (SUBR)hrtfmove_init, NULL, (SUBR)hrtfmove_process },
  { "hrtfstat", sizeof(hrtfstat),5, "aa", "aiiSSoo",
    (SUBR)hrtfstat_init, NULL, (SUBR)hrtfstat_process },
  { "hrtfmove2",  sizeof(hrtfmove2),5, "aa", "akkSSooo",
    (SUBR)hrtfmove2_init, NULL, (SUBR)hrtfmove2_process }
};

LINKAGE
```

Appendix 3: Binaural Reverb Opcodes

3.1 hrtfearly

```
/*
Brian Carty
PhD Code August 2010
binaural reverb: early reflections
*/

#include "csdl.h"
#define SQUARE(X) (X)*(X)

/* definitions, from mit */
#define minelev -40
#define elevincrement 10

static const int elevationarray[14] = {56, 60, 72, 72, 72, 72, 72, 60, 56, 45, 36, 24,
                                        12, 1 };

/* for ppc byte switch */
#ifdef WORDS_BIGENDIAN
static int swap4bytes(CSOUND* csound, MEMFIL* mfp)
{
  char c1, c2, c3, c4;
  char *p = mfp->beginp;
  int  size = mfp->length;

  while (size >= 4)
  {
    c1 = p[0]; c2 = p[1]; c3 = p[2]; c4 = p[3];
    p[0] = c4; p[1] = c3; p[2] = c2; p[3] = c1;
    size -= 4; p +=4;
  }

  return OK;
}
#else
static int (*swap4bytes)(CSOUND*, MEMFIL*) = NULL;
#endif

/* low pass filter for overall surface shape */
MYFLT filter(MYFLT* sig, MYFLT highcoeff, MYFLT lowcoeff, MYFLT *del, int vecsize, MYFLT
             sr)
{
  MYFLT costh, coef;
  int i;

  /* setup filter */
  MYFLT T = FL(1.0) / sr;
  MYFLT twopioversr = FL(2.0 * PI * T);
  MYFLT freq;
  MYFLT check;
  MYFLT scale, nyqresponse, irttwo, highresponse, lowresponse, cosw, a, b, c, x, y;

  irttwo = FL(1.0 / sqrt(2.0));

  /* simple filter deals with difference in low and high */
  highresponse = FL(1.0) - highcoeff;
  lowresponse = FL(1.0) - lowcoeff;
  /* scale factor: walls assumed to be low pass */
  scale = lowresponse;
  nyqresponse = highresponse + lowcoeff;
  /* should always be lowpass! */
  if(nyqresponse > irttwo)
```

```
    nyqresponse = irttwo;

  /* calculate cutoff, according to nyqresponse */
  /* w = twopioversr * f (= sr / (MYFLT)2.0) (w = pi in the case of nyq...2pi/sr * sr/2)
  */
  /* cosw = (MYFLT)cos(w);... = -1 in case of nyq */
  cosw = FL(-1.0);

  a = c = FL(SQUARE(nyqresponse) - FL(1.0));
  b = (FL(2.0) * cosw * FL(SQUARE(nyqresponse))) - FL(2.0);
  /* '+' and '-' sqrt in quadratic equation give equal results in this scenario:
     working backwards to find cutoff freq of simple tone filter! */
  x = (-b + FL(sqrt(SQUARE(b) - FL(4.0) * a * c))) / (FL(2.0) * a);
  y = (-FL(SQUARE(x)) - FL(1.0)) / (FL(2.0) * x);

  check = FL(2.0) - y;
  /* check for legal acos arg */
  if(check < FL(-1.0))
    check = FL(-1.0);
  freq = FL(acos(check));
  freq /= twopioversr;

  /* filter */
  costh = FL(2.0) - FL(cos(freq * twopioversr));
  coef = FL((sqrt(costh * costh - 1.0) - costh));

  for(i = 0; i < vecsize; i++)
  {
    /* filter */
    sig[i] = (sig[i] * (1 + coef) - *del * coef);
    /* scale */
    sig[i] *= scale;
    /* store */
    *del = sig[i];
    }

  return *sig;
}

/* band pass for surface detail, from csound eqfil */
MYFLT band(MYFLT* sig, MYFLT cfreq, MYFLT bw, MYFLT g, MYFLT *del, int vecsize, MYFLT
           sr)
{
  MYFLT T = FL(1.0) / sr;
  MYFLT pioversr = FL(PI) * T;
  MYFLT a = FL(cos(cfreq * pioversr * 2.0));
  MYFLT b = FL(tan(bw * pioversr));
  MYFLT c = (FL(1.0) - b) / (FL(1.0) + b);
  MYFLT w, y;
  int i;

  for(i = 0; i < vecsize; i++)
  {
    w = sig[i] + a * (FL(1.0) + c) * del[0] - c * del[1];
    y = w * c - a * (FL(1.0) + c) * del[0] + del[1];
    sig[i] = FL(0.5) * (y + sig[i] + g * (sig[i] - y));
    del[1] = del[0];
    del[0] = w;
  }

  return *sig;
}

typedef struct
{
  OPDS  h;
  /* out l/r, low rt60, high rt60, amp, delay for latediffuse */
  MYFLT *outsigl, *outsigr, *irt60low, *irt60high, *imfp;
  /* input, source, listener, hrtf files, default room, [fadelength, sr, order, threed,
     headrot, roomsize, wall high and low absorb coeffs, gain for 3 band pass, same for
     floor and ceiling] */
```

```
MYFLT *in, *srcx, *srcy, *srcz, *lstnrx, *lstnry, *lstnrz, *ifilel, *ifiler,
      *idefroom, *ofade, *osr, *porder, *othreed, *Oheadrot, *ormx, *ormy, *ormz,
      *owlh, *owll, *owlg1, *owlg2, *owlg3, *oflh, *ofll, *oflg1, *oflg2,
      *oflg3, *oclh, *ocll, *oclg1, *oclg2, *oclg3;

/* check if relative source has changed, to avoid recalculations */
MYFLT srcxv, srcyv, srczv, lstnrxv, lstnryv, lstnrzv;
MYFLT srcxk, srcyk, srczk, lstnrxk, lstnryk, lstnrzk;
MYFLT rotatev;

/* processing buffer sizes, depends on sr */
int irlength, irlengthpad, overlapsize;
MYFLT sr;
int counter;

/* crossfade preparation and checks */
int fade, fadebuffer;
int initialfade;

/* interpolation buffer declaration */
AUXCH lowl1, lowr1, lowl2, lowr2;
AUXCH highl1, highr1, highl2, highr2;
AUXCH hrtflinterp, hrtfrinterp, hrtflpad, hrtfrpad;
AUXCH hrtflpadold, hrtfrpadold;

/* convolution and in/output buffers */
AUXCH inbuf,inbufpad;
AUXCH outlspec, outrspec;
AUXCH outlspecold, outrspecold;
AUXCH overlapl, overlapr;
AUXCH overlaplold, overlaprold;

/* no of impulses based on order */
int impulses, order;
int M;
/* 3d check*/
int threed;

/* speed of sound*/
MYFLT c;

/* Image Model*/
MYFLT rmx, rmy, rmz;
int maxdelsamps;

/* for each reflection*/
AUXCH hrtflpadspec, hrtfrpadspec, hrtflpadspecold, hrtfrpadspecold;
AUXCH outl, outr, outlold, outrold;
AUXCH currentphasel, currentphaser;
AUXCH dell, delr;
AUXCH tempsrcx, tempsrcy, tempsrcz;
AUXCH dist;
AUXCH dtime;
AUXCH amp;

/* temp o/p buffers */
AUXCH predell, predelr;

/* processing values that need to be kept for each reflection*/
/* dynamic values based on no. fo impulses*/
AUXCH oldelevindex, oldangleindex;
AUXCH cross, l, delp, skipdel;
AUXCH vdt;

/* wall details */
MYFLT wallcoeflow, wallcoefhigh, wallg1, wallg2, wallg3;
MYFLT floorcoeflow, floorcoefhigh, floorg1, floorg2, floorg3;
MYFLT ceilingcoeflow, ceilingcoefhigh, ceilingg1, ceilingg2, ceilingg3;
/* wall filter q*/
MYFLT q;
```

```
  /* file pointers*/
  float *fpbeginl, *fpbeginr;

} early;

static int early_init(CSOUND *csound, early *p)
{
  /* iterator */
  int i;

  /* left and right data files: spectral mag, phase format.*/
  MEMFIL *fpl=NULL, *fpr=NULL;
  char filel[MAXNAME],filer[MAXNAME];

  /* processing sizes*/
  int irlength, irlengthpad, overlapsize;

  /* walls: surface area*/
  MYFLT wallS1, wallS2, cfS;

  /* rt60 */
  MYFLT Salphalow, Salphahigh;
  MYFLT vol;

  /* room */
  MYFLT rmx, rmy, rmz;
  /* default room */
  int defroom;

  /* dynamic values, based on number of impulses...*/
  int *oldelevindex;
  int *oldangleindex;
  int *skipdel;

  /* order calculation */
  int impulses = 1;
  int temp = 2;

  /* defs for delay lines */
  MYFLT maxdist = FL(0.0);
  MYFLT maxdtime;
  int  maxdelsamps;

  /* late tail */
  MYFLT meanfreepath;
  MYFLT surfacearea = FL(0.0);

  /* setup defaults for optional parameters */
  int fade = (int)*p->ofade;
  MYFLT sr = *p->osr;
  int threed = (int)*p->othreed;
  int order = (int)*p->porder;

  /* fade length: default 8, max 24, min 1 (fade is a local variable)*/
  if(fade < 1 || fade > 24)
    fade = 8;
  p->fade = fade;

  /* threed defaults to 2d! */
  if(threed < 0 || threed > 1)
    threed = 0;
  p->threed = threed;

  /* order: max 4, default 1 */
  if(order < 0 || order > 4)
    order = 1;
  p->order = order;

  /* sr, defualt 44100 */
  if(sr != 44100 && sr != 48000 && sr != 96000)
    sr = 44100;
```

```
p->sr = sr;

if (UNLIKELY(csound->esr != sr))
  csound->Message(csound, Str("\n\nWARNING!!:\nOrchestra SR not compatible with HRTF
                  processing SR of: %.0f\n\n"), sr);

/* setup as per sr */
if(sr == 44100 || sr == 48000)
{
  irlength = 128;
  irlengthpad = 256;
  overlapsize = (irlength - 1);
}
else if(sr == 96000)
{
  irlength = 256;
  irlengthpad = 512;
  overlapsize = (irlength - 1);
}

/* copy in string name...*/
strcpy(filel, (char*) p->ifilel);
strcpy(filer, (char*) p->ifiler);

/* reading files, with byte swap */
if (UNLIKELY((fpl = csound->ldmemfile2withCB(csound, filel, CSFTYPE_FLOATS_BINARY,
    swap4bytes)) == NULL))
return
  csound->InitError(csound, Str("\n\n\nCannot load left data file, exiting\n\n"));

if (UNLIKELY((fpr = csound->ldmemfile2withCB(csound, filer, CSFTYPE_FLOATS_BINARY,
    swap4bytes)) == NULL))
return
  csound->InitError(csound, Str("\n\n\nCannot load right data file, exiting\n\n"));

/* file handles */
p->fpbeginl = (float *) fpl->beginp;
p->fpbeginr = (float *) fpr->beginp;

/* setup structure values */
p->irlength = irlength;
p->irlengthpad = irlengthpad;
p->overlapsize = overlapsize;
p->c = 344.0;

/* zero structure values */
p->counter = 0;
p->initialfade = 0;
p->M = 0;

/* the amount of buffers to fade over */
p->fadebuffer = (int)fade * irlength;

defroom = (int)*p->idefroom;
/* 3 default rooms allowed*/
if(defroom > 3)
  defroom = 1;

/* setup wall coeffs: walls: plasterboard, ceiling: painted plaster, floor: carpet if
   any default room is chosen, default parameters for walls/ceiling/floor */
if(defroom)
{
  p->wallcoefhigh = FL(.3);
  p->wallcoeflow = FL(.1);
  p->wallg1 = FL(.75);
  p->wallg2 = FL(.95);
  p->wallg3 = FL(.9);
  p->floorcoefhigh = FL(.6);
  p->floorcoeflow = FL(.1);
  p->floorg1 = FL(.95);
  p->floorg2 = FL(.6);
  p->floorg3 = FL(.35);
  p->ceilingcoefhigh = FL(.2);
  p->ceilingcoeflow = FL(.1);
  p->ceilingg1 = FL(1.0);
  p->ceilingg2 = FL(1.0);
  p->ceilingg3 = FL(1.0);
}
/* otherwise use values, if valid */
else
{
  p->wallcoefhigh = (*p->owlh > FL(0.0) && *p->owlh < FL(1.0)) ? *p->owlh : FL(.3);
  p->wallcoeflow = (*p->owll > FL(0.0) && *p->owll < FL(1.0)) ? *p->owll : FL(.1);
  p->wallg1 = (*p->owlg1 > FL(0.0) && *p->owlg1 < FL(10.0)) ? *p->owlg1 : FL(.75);
  p->wallg2 = (*p->owlg2 > FL(0.0) && *p->owlg2 < FL(10.0)) ? *p->owlg2 : FL(.95);
  p->wallg3 = (*p->owlg3 > FL(0.0) && *p->owlg3 < FL(10.0)) ? *p->owlg3 : FL(.9);
  p->floorcoefhigh = (*p->oflh > FL(0.0) && *p->oflh < FL(1.0)) ? *p->oflh : FL(.6);
  p->floorcoeflow = (*p->ofll > FL(0.0) && *p->ofll < FL(1.0)) ? *p->ofll : FL(.1);
  p->floorg1 = (*p->oflg1 > FL(0.0) && *p->oflg1 < FL(10.0)) ? *p->oflg1 : FL(.95);
  p->floorg2 = (*p->oflg2 > FL(0.0) && *p->oflg2 < FL(10.0)) ? *p->oflg2 : FL(.6);
  p->floorg3 = (*p->oflg3 > FL(0.0) && *p->oflg3 < FL(10.0)) ? *p->oflg3 : FL(.35);
  p->ceilingcoefhigh = (*p->oclh > FL(0.0) && *p->oclh < FL(1.0)) ? *p->oclh : FL(.2);
  p->ceilingcoeflow = (*p->ocll > FL(0.0) && *p->ocll < FL(1.0)) ? *p->ocll : FL(.1);
  p->ceilingg1 = (*p->oclg1 > FL(0.0) && *p->oclg1 < FL(10.0)) ? *p->oclg1 : FL(1.);
  p->ceilingg2 = (*p->oclg2 > FL(0.0) && *p->oclg2 <  FL(10.0)) ? *p->oclg2 : FL(1.);
  p->ceilingg3 = (*p->oclg3 > FL(0.0) && *p->oclg3 < FL(10.0)) ? *p->oclg3 : FL(1.);
}

/* medium room */
if(defroom == 1)
{
  rmx = 10;
  rmy = 10;
  rmz = 3;
}
/* small */
else if(defroom == 2)
{
  rmx = 4;
  rmy = 4;
  rmz = 3;
}
/* large */
else if(defroom == 3)
{
  rmx = 20;
  rmy = 25;
  rmz = 7;
}

/* read values if they exist, use medium if not valid (must be at least a 2 * 2 * 2
   room! */
else
{
  rmx = *p->ormx >= FL(2.0) ? *p->ormx : 10;
  rmy = *p->ormy >= FL(2.0) ? *p->ormy : 10;
  rmz = *p->ormz >= FL(2.0) ? *p->ormz : 3;
}

/* store */
p->rmx = rmx;
p->rmy = rmy;
p->rmz = rmz;

/* how many sources? */
if(threed)
{
  for(i = 1; i <= order; i++)
  {
    impulses += (4 * i);
    if(i <= (order - 1))
    /* sources = 2d impulses for order, plus 2 * each preceding no of impulses eg
```

```
       order 2: 2d = 1 + 4 + 8 = 13, 3d + 2*5 + 2 = 25*/
      temp += 2*impulses;
    else
      impulses = impulses + temp;
  }
}
else
{
  for(i = 1; i <= order; i++)
    /* there will be 4 * order additional impulses for each order*/
    impulses += (4*i);
}
p->impulses = impulses;

/* allocate memory, reuse if possible: interpolation buffers */
if (!p->lowl1.auxp || p->lowl1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowl1);
if (!p->lowr1.auxp || p->lowr1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowr1);
if (!p->lowl2.auxp || p->lowl2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowl2);
if (!p->lowr2.auxp || p->lowr2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->lowr2);
if (!p->highl1.auxp || p->highl1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highl1);
if (!p->highr1.auxp || p->highr1.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highr1);
if (!p->highl2.auxp || p->highl2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highl2);
if (!p->highr2.auxp || p->highr2.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->highr2);
if (!p->hrtflinterp.auxp || p->hrtflinterp.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->hrtflinterp);
if (!p->hrtfrinterp.auxp || p->hrtfrinterp.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->hrtfrinterp);

memset(p->lowl1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowr1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowl2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->lowr2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highl1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highl2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highr1.auxp, 0, irlength * sizeof(MYFLT));
memset(p->highr2.auxp, 0, irlength * sizeof(MYFLT));
memset(p->hrtflinterp.auxp, 0, irlength * sizeof(MYFLT));
memset(p->hrtfrinterp.auxp, 0, irlength * sizeof(MYFLT));

/* hrtf processing buffers */
if (!p->hrtflpad.auxp || p->hrtflpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->hrtflpad);
if (!p->hrtfrpad.auxp || p->hrtfrpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->hrtfrpad);
if (!p->hrtflpadold.auxp || p->hrtflpadold.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->hrtflpadold);
if (!p->hrtfrpadold.auxp || p->hrtfrpadold.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->hrtfrpadold);

memset(p->hrtflpad.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtfrpad.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtflpadold.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtfrpadold.auxp, 0, irlengthpad * sizeof(MYFLT));

/* convolution & processing */
if (!p->inbuf.auxp || p->inbuf.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->inbuf);
if (!p->inbufpad.auxp || p->inbufpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p-> inbufpad);

memset(p->inbuf.auxp, 0, irlength * sizeof(MYFLT));
memset(p->inbufpad.auxp, 0, irlengthpad * sizeof(MYFLT));
```

```
if (!p->outlspec.auxp || p->outlspec.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->outlspec);
if (!p->outrspec.auxp || p->outrspec.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->outrspec);

memset(p->outlspec.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outrspec.auxp, 0, irlengthpad * sizeof(MYFLT));

if (!p->outlspecold.auxp || p->outlspecold.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->outlspecold);
if (!p->outrspecold.auxp || p->outrspecold.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->outrspecold);

memset(p->outlspecold.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->outrspecold.auxp, 0, irlengthpad * sizeof(MYFLT));

if (!p->overlapl.auxp || p->overlapl.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize * sizeof(MYFLT), &p->overlapl);
if (!p->overlapr.auxp || p->overlapr.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize * sizeof(MYFLT), &p->overlapr);

memset(p->overlapl.auxp, 0, overlapsize * sizeof(MYFLT));
memset(p->overlapr.auxp, 0, overlapsize * sizeof(MYFLT));

if (!p->overlaplold.auxp || p->overlaplold.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize * sizeof(MYFLT), &p->overlaplold);
if (!p->overlaprold.auxp || p->overlaprold.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize * sizeof(MYFLT), &p->overlaprold);

memset(p->overlaplold.auxp, 0, overlapsize * sizeof(MYFLT));
memset(p->overlaprold.auxp, 0, overlapsize * sizeof(MYFLT));

/* dynamic values, based on no. of impulses*/
if (!p->predell.auxp || p->predell.size < irlength * impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * impulses * sizeof(MYFLT), &p->predell);
if (!p->predelr.auxp || p->predelr.size < irlength * impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * impulses * sizeof(MYFLT), &p->predelr);

memset(p->predell.auxp, 0, irlength * impulses * sizeof(MYFLT));
memset(p->predelr.auxp, 0, irlength * impulses * sizeof(MYFLT));

if (!p->cross.auxp || p->cross.size < impulses * sizeof(int))
  csound->AuxAlloc(csound, impulses * sizeof(int), &p->cross);
if (!p->l.auxp || p->l.size < impulses * sizeof(int))
  csound->AuxAlloc(csound, impulses * sizeof(int), &p->l);
if (!p->delp.auxp || p->delp.size < impulses * sizeof(int))
  csound->AuxAlloc(csound, impulses * sizeof(int), &p->delp);
if (!p->skipdel.auxp || p->skipdel.size < impulses * sizeof(int))
  csound->AuxAlloc(csound, impulses * sizeof(int), &p->skipdel);
if (!p->vdt.auxp || p->vdt.size < impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, impulses * sizeof (MYFLT), &p->vdt);

memset(p->cross.auxp, 0, impulses * sizeof(int));
memset(p->l.auxp, 0, impulses * sizeof(int));
memset(p->delp.auxp, 0, impulses * sizeof(int));
memset(p->vdt.auxp, 0, impulses * sizeof(MYFLT));
/* skipdel looked after below */

/* values distinct to each reflection*/
if (!p->tempsrcx.auxp || p->tempsrcx.size < impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, impulses * sizeof(MYFLT), &p->tempsrcx);
if (!p->tempsrcy.auxp || p->tempsrcy.size < impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, impulses * sizeof(MYFLT), &p->tempsrcy);
if (!p->tempsrcz.auxp || p->tempsrcz.size < impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, impulses * sizeof(MYFLT), &p->tempsrcz);
if (!p->dist.auxp || p->dist.size < impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, impulses * sizeof(MYFLT), &p->dist);
if (!p->dtime.auxp || p->dtime.size < impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, impulses * sizeof(MYFLT), &p->dtime);
if (!p->amp.auxp || p->amp.size < impulses * sizeof(MYFLT))
csound->AuxAlloc(csound, impulses * sizeof(MYFLT), &p->amp);
```

```
memset(p->tempsrcx.auxp, 0, impulses * sizeof(MYFLT));
memset(p->tempsrcy.auxp, 0, impulses * sizeof(MYFLT));
memset(p->tempsrcz.auxp, 0, impulses * sizeof(MYFLT));
memset(p->dist.auxp, 0, impulses * sizeof(MYFLT));
memset(p->dtime.auxp, 0, impulses * sizeof(MYFLT));
memset(p->amp.auxp, 0, impulses * sizeof(MYFLT));

if (!p->oldelevindex.auxp || p->oldelevindex.size < impulses * sizeof(int))
  csound->AuxAlloc(csound, impulses * sizeof(int), &p->oldelevindex);
if (!p->oldangleindex.auxp || p->oldangleindex.size < impulses * sizeof(int))
  csound->AuxAlloc(csound, impulses * sizeof(int), &p->oldangleindex);

/* no need to zero above, as filled below...*/

/* -1 for first check */
oldelevindex = (int *)p->oldelevindex.auxp;
oldangleindex = (int *)p->oldangleindex.auxp;

for(i = 0; i < impulses; i++)
  oldelevindex[i] = oldangleindex[i] = -1;

/* more values distinct to each reflection */
if(!p->hrtflpadspec.auxp || p->hrtflpadspec.size < irlengthpad * impulses *
   sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * impulses * sizeof(MYFLT), &p->hrtflpadspec);
if(!p->hrtfrpadspec.auxp || p->hrtfrpadspec.size < irlengthpad * impulses *
   sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * impulses * sizeof(MYFLT), &p->hrtfrpadspec);

memset(p->hrtflpadspec.auxp, 0, irlengthpad * impulses * sizeof(MYFLT));
memset(p->hrtfrpadspec.auxp, 0, irlengthpad * impulses * sizeof(MYFLT));

if (!p->hrtflpadspecold.auxp || p->hrtflpadspecold.size < irlengthpad * impulses *
    sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * impulses * sizeof(MYFLT),
                   &p->hrtflpadspecold);
if (!p->hrtfrpadspecold.auxp || p->hrtfrpadspecold.size < irlengthpad * impulses *
    sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * impulses * sizeof(MYFLT),
                   &p->hrtfrpadspecold);

memset(p->hrtflpadspecold.auxp, 0, irlengthpad * impulses * sizeof(MYFLT));
memset(p->hrtfrpadspecold.auxp, 0, irlengthpad * impulses * sizeof(MYFLT));

if (!p->outl.auxp || p->outl.size < irlengthpad * impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * impulses * sizeof(MYFLT), &p->outl);
if (!p->outr.auxp || p->outr.size < irlengthpad * impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * impulses * sizeof(MYFLT), &p->outr);

memset(p->outl.auxp, 0, irlengthpad * impulses * sizeof(MYFLT));
memset(p->outr.auxp, 0, irlengthpad * impulses * sizeof(MYFLT));

if (!p->outlold.auxp || p->outlold.size < irlengthpad * impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * impulses * sizeof(MYFLT), &p->outlold);
if (!p->outrold.auxp || p->outrold.size < irlengthpad * impulses * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * impulses * sizeof(MYFLT), &p->outrold);

memset(p->outlold.auxp, 0, irlengthpad * impulses * sizeof(MYFLT));
memset(p->outrold.auxp, 0, irlengthpad * impulses * sizeof(MYFLT));

if (!p->currentphasel.auxp || p->currentphasel.size < irlength * impulses *
    sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * impulses * sizeof(MYFLT), &p->currentphasel);
if (!p->currentphaser.auxp || p->currentphaser.size < irlength * impulses *
    sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * impulses * sizeof(MYFLT), &p->currentphaser);

memset(p->currentphasel.auxp, 0, irlength * impulses * sizeof(MYFLT));
memset(p->currentphaser.auxp, 0, irlength * impulses * sizeof(MYFLT));
```

```
/* setup rt60 calcs...*/
/* rectangular room: surface area of opposite walls, and floor/ceiling */
wallS1 = rmy * rmz;
wallS2 = rmx * rmz;
cfS = rmx * rmy;

/* volume and surface areas, for rt60 calc, for high and low frequencies */
vol = rmx * rmy * rmz;

/* add all surfaces, 2 of each wall in shoebox geometry */
Salphalow = wallS1 * FL(log(1.0 - p->wallcoeflow)) * FL(2.0);
Salphalow += wallS2 * FL(log(1.0 - p->wallcoeflow)) * FL(2.0);
Salphalow += cfS * FL(log(1.0 - p->floorcoeflow));
Salphalow += cfS * FL(log(1.0 - p->ceilingcoeflow));
Salphahigh = wallS1 * FL(log(1.0 - p->wallcoefhigh)) * FL(2.0);
Salphahigh += wallS2 * FL(log(1.0 - p->wallcoefhigh)) * FL(2.0);
Salphahigh += cfS * FL(log(1.0 - p->floorcoefhigh));
Salphahigh += cfS * FL(log(1.0 - p->ceilingcoefhigh));

/* wall filter quality factor (4 octaves for required spread!: .2666667) (2 octaves =
   .6667 implies 125 - 500, cf 250, 500 - 2k, cf 1k, 2000 - 8000, cf 4k) (4 octaves =
   .2666667 implies 62.5 - 1000, cf 250, 250 - 4000, cf 1k, 1000 - 16k, cf 4k */
p->q = FL(.2666667);

*p->irt60low = (FL(-0.161) * vol)/Salphalow;
*p->irt60high = (FL(-0.161) * vol)/Salphahigh;

/* calculate max delay according to max dist from order */
/* use hypotenuse rule to get max dist */
/* could calculate per order, but designed for low order use */
maxdist = FL(sqrt(SQUARE(rmx) + SQUARE(rmy)));
if(threed)
  maxdist = FL(sqrt(SQUARE(maxdist)+ SQUARE(rmz)));
maxdist = maxdist * (order + 1);

maxdtime = maxdist / p->c;
maxdelsamps = (int)(maxdtime * sr);
p->maxdelsamps = maxdelsamps;

surfacearea = FL(2.0) * wallS1 + FL(2.0) * wallS2 + FL(2.0) * cfS;

meanfreepath = FL(4.0) * vol / (surfacearea * p->c);

/* set output...*/
*p->imfp = meanfreepath;

/* allocate delay memory for each impulse */
if (!p->dell.auxp || p->dell.size < maxdelsamps * sizeof(MYFLT) * impulses)
  csound->AuxAlloc(csound, maxdelsamps * sizeof(MYFLT) * impulses, &p->dell);
if (!p->delr.auxp || p->delr.size < maxdelsamps * sizeof(MYFLT) * impulses)
  csound->AuxAlloc(csound, maxdelsamps * sizeof(MYFLT) * impulses, &p->delr);

memset(p->dell.auxp, 0, maxdelsamps * impulses * sizeof(MYFLT));
memset(p->delr.auxp, 0, maxdelsamps * impulses * sizeof(MYFLT));

/* amount to skip in to each del line */
skipdel = (int *)p->skipdel.auxp;

for(i = 0; i < impulses; i++)
  skipdel[i] = i * maxdelsamps;

/* setup values used to check if relative source position has changed, start at
   illegal value to ensure first process read */
p->srcxv = FL(-1.0);
p->srcyv = FL(-1.0);
p->srczv = FL(-1.0);
p->lstnrxv = FL(-1.0);
p->lstnryv = FL(-1.0);
p->lstnrzv = FL(-1.0);
p->srcxk = FL(-1.0);
p->srcyk = FL(-1.0);
```

```
  p->srczk = FL(-1.0);
  p->lstnrxk = FL(-1.0);
  p->lstnryk = FL(-1.0);
  p->lstnrzk = FL(-1.0);

  p->rotatev = FL(0.0);

  return OK;
}

static int early_process(CSOUND *csound, early *p)
{
  /* iterators */
  int n, i, j;

  /* local pointers to p */
  MYFLT *in = p->in;
  MYFLT *outsigl  = p->outsigl;
  MYFLT *outsigr = p->outsigr;

  int irlength = p->irlength;
  int irlengthpad = p->irlengthpad;
  int overlapsize = p->overlapsize;

  int counter = p->counter;

  /* convolution buffers */
  MYFLT *lowl1 = (MYFLT *)p->lowl1.auxp;
  MYFLT *lowr1 = (MYFLT *)p->lowr1.auxp;
  MYFLT *lowl2 = (MYFLT *)p->lowl2.auxp;
  MYFLT *lowr2 = (MYFLT *)p->lowr2.auxp;
  MYFLT *highl1 = (MYFLT *)p->highl1.auxp;
  MYFLT *highr1 = (MYFLT *)p->highr1.auxp;
  MYFLT *highl2 = (MYFLT *)p->highl2.auxp;
  MYFLT *highr2 = (MYFLT *)p->highr2.auxp;
  MYFLT *hrtflinterp = (MYFLT *)p->hrtflinterp.auxp;
  MYFLT *hrtfrinterp = (MYFLT *)p->hrtfrinterp.auxp;

  /* hrtf processing buffers */
  MYFLT *hrtflpad = (MYFLT *)p->hrtflpad.auxp;
  MYFLT *hrtfrpad = (MYFLT *)p->hrtfrpad.auxp;
  MYFLT *hrtflpadold = (MYFLT *)p->hrtflpadold.auxp;
  MYFLT *hrtfrpadold = (MYFLT *)p->hrtfrpadold.auxp;

  /* pointers into HRTF files: floating point data(even in 64 bit csound)*/
  float *fpindexl = (float *)p->fpbeginl;
  float *fpindexr = (float *)p->fpbeginr;

  /* local copies */
  MYFLT srcx = *p->srcx;
  MYFLT srcy = *p->srcy;
  MYFLT srcz = *p->srcz;
  MYFLT lstnrx = *p->lstnrx;
  MYFLT lstnry = *p->lstnry;
  MYFLT lstnrz = *p->lstnrz;
  MYFLT rotate = *p->Oheadrot;

  MYFLT sr = p->sr;

  /* local variables, mainly used for simplification */
  MYFLT elevindexstore;
  MYFLT angleindexlowstore;
  MYFLT angleindexhighstore;

  /* for reading */
  MYFLT angle, elev;
  int elevindex;
  int angleindex;
  int skip = 0;

  /* crossfade preparation and checks */
```

```
int fade = p->fade;
int fadebuffer = p->fadebuffer;
int initialfade = p->initialfade;
int crossfade;
int crossout;

/* interpolation variable declaration: local */
int elevindexlow, elevindexhigh, angleindex1, angleindex2, angleindex3, angleindex4;
MYFLT elevindexhighper, angleindex2per, angleindex4per;
MYFLT magllow, magrlow, maglhigh, magrhigh, magl, magr, phasel, phaser;

/* convolution and in/output buffers */
MYFLT *inbuf = (MYFLT *)p->inbuf.auxp;
MYFLT *inbufpad = (MYFLT *)p->inbufpad.auxp;
MYFLT *outlspec = (MYFLT *)p->outlspec.auxp;
MYFLT *outrspec = (MYFLT *)p->outrspec.auxp;
MYFLT *outlspecold = (MYFLT *)p->outlspecold.auxp;
MYFLT *outrspecold = (MYFLT *)p->outrspecold.auxp;
MYFLT *overlapl = (MYFLT *)p->overlapl.auxp;
MYFLT *overlapr = (MYFLT *)p->overlapr.auxp;
MYFLT *overlaplold = (MYFLT *)p->overlaplold.auxp;
MYFLT *overlaprold = (MYFLT *)p->overlaprold.auxp;
MYFLT *predell = (MYFLT *)p->predell.auxp;
MYFLT *predelr = (MYFLT *)p->predelr.auxp;

MYFLT outltot, outrtot;

/* distinct to each reflection */
MYFLT *hrtflpadspec = (MYFLT *)p->hrtflpadspec.auxp;
MYFLT *hrtfrpadspec = (MYFLT *)p->hrtfrpadspec.auxp;
MYFLT *hrtflpadspecold = (MYFLT *)p->hrtflpadspecold.auxp;
MYFLT *hrtfrpadspecold = (MYFLT *)p->hrtfrpadspecold.auxp;
MYFLT *outl = (MYFLT *)p->outl.auxp;
MYFLT *outr = (MYFLT *)p->outr.auxp;
MYFLT *outlold = (MYFLT *)p->outlold.auxp;
MYFLT *outrold = (MYFLT *)p->outrold.auxp;
MYFLT *currentphasel = (MYFLT *)p->currentphasel.auxp;
MYFLT *currentphaser = (MYFLT *)p->currentphaser.auxp;
MYFLT *dell = (MYFLT *)p->dell.auxp;
MYFLT *delr = (MYFLT *)p->delr.auxp;

/* as above */
int *oldelevindex = (int *)p->oldelevindex.auxp;
int *oldangleindex = (int *)p->oldangleindex.auxp;
int *cross = (int *)p->cross.auxp;
int *l = (int *)p->l.auxp;
int *delp = (int *)p->delp.auxp;
int *skipdel = (int *)p->skipdel.auxp;
MYFLT *vdt = (MYFLT *)p->vdt.auxp;
MYFLT *dist = (MYFLT *)p->dist.auxp;
MYFLT *dtime = (MYFLT *)p->dtime.auxp;
MYFLT *amp = (MYFLT *)p->amp.auxp;
MYFLT *tempsrcx = (MYFLT *)p->tempsrcx.auxp;
MYFLT *tempsrcy = (MYFLT *)p->tempsrcy.auxp;
MYFLT *tempsrcz = (MYFLT *)p->tempsrcz.auxp;
MYFLT tempdist;

/* from structure */
int impulses = p->impulses;
int order = p->order;
int M = p->M;
int threed = p->threed;

/* used in vdel */
int maxdelsamps = p->maxdelsamps;
MYFLT c = p->c;
int pos;
MYFLT rp, frac;

/* room size */
MYFLT rmx = p->rmx;
MYFLT rmy = p->rmy;
MYFLT rmz = p->rmz;

/* xc = x coordinate, etc...*/
int xc, yc, zc, lowz, highz;

/* to simplify formulae, local */
MYFLT formx, formy, formz;
int formxpow, formypow, formzpow;

int wallreflections, floorreflections, ceilingreflections;
MYFLT delsinglel, delsingler;
MYFLT deldoublel[2], deldoubler[2];

/* temp variables, for efficiency */
MYFLT tempx, tempy;

/* angle / elev calc of source location */
MYFLT newpntx, newpnty, newpntz;
MYFLT ab,ac,bc;
MYFLT coselev;

/* processing size! */
n = csound->ksmps;

/* check for legal src/lstnr locations */
/* restricted to being inside room! */
if(srcx > (rmx - FL(.1)))
  srcx = rmx - FL(.1);
if(srcx < FL(.1))
  srcx = FL(.1);
if(srcy > (rmy - FL(.1)))
  srcy = rmy - FL(.1);
if(srcy < FL(.1))
  srcy = FL(.1);
if(srcz > (rmz - FL(.1)))
  srcz = rmz - FL(.1);
if(srcz < FL(.1))
  srcz = FL(.1);
if(lstnrx > (rmx - FL(.1)))
  lstnrx = rmx - FL(.1);
if(lstnrx < FL(.1))
  lstnrx = FL(.1);
if(lstnry > (rmy - FL(.1)))
  lstnry = rmy - FL(.1);
if(lstnry < FL(.1))
  lstnry = FL(.1);
if(lstnrz > (rmz - FL(.1)))
  lstnrz = rmz - FL(.1);
if(lstnrz < FL(.1))
  lstnrz = FL(.1);

/* k rate computations: sources, distances, delays, amps for each image source. */
/* need minus values for formula... */

/* only update if relative source updates! improves speed in static sources by a
   factor of 2-3! */
if(srcx != p->srcxk || srcy != p->srcyk || srcz != p->srczk || lstnrx != p->lstnrxk ||
   lstnry != p->lstnryk || lstnrz != p->lstnrzk)
{
  p->srcxk = srcx;
  p->srcyk = srcy;
  p->srczk = srcz;
  p->lstnrxk = lstnrx;
  p->lstnryk = lstnry;
  p->lstnrzk = lstnrz;

  for(xc = -order; xc <= order; xc++)
  {
    for(yc = abs(xc) - order; yc <= order - abs(xc); yc++)
    {
      /* only scroll through z plane if 3d required...*/
      if(threed)
      {
        lowz = abs(yc) - (order - abs(xc));
        highz = (order - abs(xc)) - abs(yc);
      }
      else
      {
        lowz = 0;
        highz = 0;
      }
      for(zc = lowz; zc <= highz; zc++)
      {
        /* to avoid recalculation, especially at audio rate for delay, later on */
        formxpow = (int)pow(-1.0, xc);
        formypow = (int)pow(-1.0, yc);
        formzpow = (int)pow(-1.0, zc);
        formx = (xc + (1 - formxpow)/2) * rmx;
        formy = (yc + (1 - formypow)/2) * rmy;
        formz = (zc + (1 - formzpow)/2) * rmz;

        /* image */
        tempsrcx[M] = formxpow * srcx + formx;
        tempsrcy[M] = formypow * srcy + formy;
        tempsrcz[M] = formzpow * srcz + formz;

        /* Calculate delay here using source and listener location */
       dist[M] = FL(sqrt(SQUARE(tempsrcx[M] - lstnrx) + SQUARE(tempsrcy[M] - lstnry,)
                  + SQUARE(tempsrcz[M] - lstnrz)));

        /* in seconds... */
        dtime[M] = dist[M] / c;

        /* furthest allowable distance....max amp = 1. */
        tempdist = (dist[M] < FL(.45) ? FL(.45) : dist[M]);

        /* high amp value may cause clipping on early reflections...reduce overall amp
           if so...*/
        /* SPL inverse distance law */
        amp[M] = FL(.45) / tempdist;

        /* vdels for distance processing: */
        vdt[M] = dtime[M] * sr;
        if(vdt[M] > maxdelsamps)
          vdt[M] = FL(maxdelsamps);

        M++;
        M = M % impulses;
      }
    }
  }
}

/* a rate... */
for(j=0;j<n;j++)
{
  /* input */
  inbuf[counter] = in[j];

  /* output */
  outltot = 0.0;
  outrtot = 0.0;

  /* for each reflection */
  for(M = 0; M < impulses; M++)
  {
    /* a rate vdel: */
    rp = delp[M] - vdt[M];
    rp = (rp >= 0 ? (rp < maxdelsamps ? rp : rp - maxdelsamps) : rp + maxdelsamps);
    frac = rp - (int)rp;
    /* shift into correct part of buffer */
    pos = (int)rp + skipdel[M];
    /* write to l and r del lines */
    dell[delp[M] + skipdel[M]] = predell[counter + M * irlength] * amp[M];
    delr[delp[M] + skipdel[M]] = predelr[counter + M * irlength] * amp[M];
    /* read, at variable interpolated speed */
    outltot += dell[pos] + frac*(dell[(pos + 1 < (maxdelsamps + skipdel[M]) ? pos + 1
               : skipdel[M])] - dell[pos]);
    outrtot += delr[pos] + frac*(delr[(pos + 1 < (maxdelsamps + skipdel[M]) ? pos + 1
               : skipdel[M])] - delr[pos]);
    delp[M] = (delp[M] != maxdelsamps - 1 ? delp[M] + 1 : 0);

    outsigl[j] = outltot;
    outsigr[j] = outrtot;
  }
  counter++;

  /* used to ensure fade does not happen on first run */
  if(initialfade < (irlength + 2))
    initialfade++;

  /* 'hrtf buffer' rate */
  if(counter == irlength)
  {
    /* reset */
    M = 0;
    /* run according to formula */
    for(xc = -order; xc <= order; xc++)
    {
      for(yc = abs(xc) - order; yc <= order - abs(xc); yc++)
      {
        /* only scroll through z plane if 3d required... */
        if(threed)
        {
          lowz = abs(yc) - (order - abs(xc));
          highz = (order - abs(xc)) - abs(yc);
        }
        else
        {
          lowz = 0;
          highz = 0;
        }
        for(zc = lowz; zc <= highz; zc++)
        {
          /* zero */
          crossout = 0;
          crossfade = 0;

          /* avoid unnecessary processing if relative source location has not changed
          */
          if(srcx != p->srcxv || srcy != p->srcyv || srcz != p->srczv || lstnrx !=
             p->lstnrxv || lstnry != p->lstnryv || lstnrz != p->lstnrzv || rotate !=
             p->rotatev)
          {
            /* if first process complete (128 samps in) and source is too close to
               listener: warning, do not process */
            /* duda and martens range dependence jasa 98: 5 times radius: near
               field...hrtf changes! */
            if(dist[M] < FL(.45) && initialfade > irlength)
            ;  /* do not process... */
            else
            {
              /* to avoid case where atan2 is invalid */
              tempx = tempsrcx[M] - lstnrx;
              tempy = tempsrcy[M] - lstnry;
              if(tempx == 0 && tempy == 0)
                angle = 0;
              else
              {
                /* - to invert anticlockwise to clockwise */
                angle = FL(-(atan2(tempy, tempx)) * 180.0 / PI);
                /* add 90 to go from y axis (front) */
                angle = angle + 90;
              }

              /* xy point will be same as source, z same as listener: a workable
                 triangle */
              newpntx = tempsrcx[M];
              newpnty = tempsrcy[M];
              newpntz = lstnrz;

              /* ab: source to listener, ac: source to new point under/over source, bc
                 listener to new point */
              /* a = source, b = listener, c = new point */
              ab = FL(sqrt(SQUARE(tempsrcx[M] - lstnrx) + SQUARE(tempsrcy[M] –
                   lstnry) + SQUARE(tempsrcz[M] - lstnrz)));
              ac = FL(sqrt(SQUARE(tempsrcx[M] - newpntx) + SQUARE(tempsrcy[M] –
                   newpnty) + SQUARE(tempsrcz[M] - newpntz)));
              bc = FL(sqrt(SQUARE(lstnrx - newpntx) + SQUARE(lstnry - newpnty) +
                   SQUARE(lstnrz - newpntz)));

              /* elev: when bc == 0 -> source + listener at same x,y point (may happen
                 in first run, checked after that) angle = 0, elev = 0 if at same
                 point, or source may be directly above/below */
              if(bc == FL(0.0))
              {
                /* source at listener */
                if(ac == FL(0.0))
                  elev = FL(0.0);
                /* source above listener */
                else
                  elev = FL(90.0);
              }
              else
              {
                /* cosine rule */
                coselev = FL((SQUARE(bc) + SQUARE(ab) - SQUARE(ac)) / (2.0 * ab
                          * bc));
                elev = FL(acos(coselev)* 180.0 / PI);
              }

              /* if z coefficient of source < listener: source below listener...*/
              if(tempsrcz[M] < lstnrz)
                elev *= -1;

              if(elev > FL(90.0))
                elev = FL(90.0);
              if(elev < FL(-40.0))
                elev = FL(-40.0);

              /* two nearest elev indices to avoid recalculating */
              elevindexstore = (elev - minelev) / elevincrement;
              elevindexlow = (int)elevindexstore;

              if(elevindexlow < 13)
                elevindexhigh = elevindexlow + 1;
              /* highest index reached */
              else
                elevindexhigh = elevindexlow;

              /* get percentage value for interpolation */
              elevindexhighper = elevindexstore - elevindexlow;

              /* head rotation */
              angle -= rotate;

              while(angle < FL(0.0))
                angle += FL(360.0);
              while(angle >= FL(360.0))
                angle -= FL(360.0);

              /* as above,lookup index, used to check for crossfade */
              elevindex = (int)(elevindexstore + 0.5);
```

```
angleindex = (int)(angle / (360.0 / elevationarray[elevindex]) + 0.5);
angleindex = angleindex % elevationarray[elevindex];

/* avoid recalculation */
angleindexlowstore = angle / (FL(360.0) / elevationarray[elevindexlow]);
angleindexhighstore = angle / (FL(360.0) /
                      elevationarray[elevindexhigh]);

/* 4 closest indices, 2 low and 2 high */
angleindex1 = (int)angleindexlowstore;

angleindex2 = angleindex1 + 1;
angleindex2 = angleindex2 % elevationarray[elevindexlow];

angleindex3 = (int)angleindexhighstore;

angleindex4 = angleindex3 + 1;
angleindex4 = angleindex4 % elevationarray[elevindexhigh];

/* angle percentages for interp */
angleindex2per = angleindexlowstore - angleindex1;
angleindex4per = angleindexhighstore - angleindex3;

/* crossfade happens if index changes:nearest measurement changes, 1st
   step: store old values */
if (oldelevindex[M] != elevindex || oldangleindex[M] != angleindex)
{
  if(initialfade > irlength)
  {
    /* warning on overlapping fades */
    if(cross[M])
    {
      csound->Message(csound, "\nWARNING: fades are overlapping: this
                      could lead to noise: reduce fade size or
                      change trajectory\n\n");
      cross[M] = 0;
    }
    /* reset l */
    l[M] = 0;
    crossfade = 1;
    for(i = 0; i < irlengthpad; i++)
    {
      hrtflpadspecold[irlengthpad * M + i] = hrtflpadspec[irlengthpad *
                                             M + i];
      hrtfrpadspecold[irlengthpad * M + i] = hrtfrpadspec[irlengthpad *
                                             M + i];
    }
  }

  skip = 0;
  /* store current phase */
  if(angleindex > elevationarray[elevindex] / 2)
  {
    for(i = 0; i < elevindex; i ++)
      skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
    for (i = 0; i < (elevationarray[elevindex] - angleindex); i++)
      skip += irlength;
    for(i = 0; i < irlength; i++)
    {
      currentphasel[irlength * M + i] = fpindexr[skip + i];
      currentphaser[irlength * M + i] = fpindexl[skip + i];
    }
  }
  else
  {
    for(i = 0; i < elevindex; i ++)
      skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
    for (i = 0; i < angleindex; i++)
      skip += irlength;
    for(i = 0; i < irlength; i++)
    {
      currentphasel[irlength * M + i] = fpindexl[skip+i];
      currentphaser[irlength * M + i] = fpindexr[skip+i];
    }
  }
}

/* for next check */
oldelevindex[M] = elevindex;
oldangleindex[M] = angleindex;

/* read 4 nearest HRTFs */
/* switch l and r */
skip = 0;
if(angleindex1 > elevationarray[elevindexlow] / 2)
{
  for(i = 0; i < elevindexlow; i ++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexlow] - angleindex1); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = fpindexr[skip+i];
    lowr1[i] = fpindexl[skip+i];
  }
}
else
{
  for(i = 0; i < elevindexlow; i ++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex1; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl1[i] = fpindexl[skip+i];
    lowr1[i] = fpindexr[skip+i];
  }
}

skip = 0;
if(angleindex2 > elevationarray[elevindexlow] / 2)
{
  for(i = 0; i < elevindexlow; i ++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexlow] - angleindex2); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = fpindexr[skip+i];
    lowr2[i] = fpindexl[skip+i];
  }
}
else
{
  for(i = 0; i < elevindexlow; i ++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex2; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    lowl2[i] = fpindexl[skip+i];
    lowr2[i] = fpindexr[skip+i];
  }
}

skip = 0;
if(angleindex3 > elevationarray[elevindexhigh] / 2)
{
  for(i = 0; i < elevindexhigh; i ++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexhigh] - angleindex3); i++)
```

```
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl1[i] = fpindexr[skip+i];
    highr1[i] = fpindexl[skip+i];
  }
}
else
{
  for(i = 0; i < elevindexhigh; i ++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex3; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl1[i] = fpindexl[skip+i];
    highr1[i] = fpindexr[skip+i];
  }
}

skip = 0;
if(angleindex4 > elevationarray[elevindexhigh] / 2)
{
  for(i = 0; i < elevindexhigh; i ++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < (elevationarray[elevindexhigh] - angleindex4); i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl2[i] = fpindexr[skip+i];
    highr2[i] = fpindexl[skip+i];
  }
}
else
{
  for(i = 0; i < elevindexhigh; i ++)
    skip +=((int)(elevationarray[i] / 2) + 1) * irlength;
  for (i = 0; i < angleindex4; i++)
    skip += irlength;
  for(i = 0; i < irlength; i++)
  {
    highl2[i] = fpindexl[skip+i];
    highr2[i] = fpindexr[skip+i];
  }
}

/* magnitude interpolation */
/* 0hz and Nyq */
magllow = FL(fabs(lowl1[0])) + (FL(fabs(lowl2[0]) - fabs(lowl1[0]))) *
          angleindex2per;
maglhigh = FL(fabs(highl1[0])) + (FL(fabs(highl2[0]) - fabs(highl1[0])))
           * angleindex4per;
magrlow = FL(fabs(lowr1[0])) + (FL(fabs(lowr2[0]) - fabs(lowr1[0]))) *
          angleindex2per;
magrhigh = FL(fabs(highr1[0])) + (FL(fabs(highr2[0]) - fabs(highr1[0])))
           * angleindex4per;
magl = magllow + (maglhigh - magllow) * elevindexhighper;
magr = magrlow + (magrhigh - magrlow) * elevindexhighper;
if(currentphasel[M * irlength] < FL(0.0))
  hrtflinterp[0] = - magl;
else
  hrtflinterp[0] = magl;
if(currentphaser[M * irlength] < FL(0.0))
  hrtfrinterp[0] = - magr;
else
  hrtfrinterp[0] = magr;

magllow = FL(fabs(lowl1[1])) + (FL(fabs(lowl2[1]) - fabs(lowl1[1]))) *
          angleindex2per;
maglhigh = FL(fabs(highl1[1])) + (FL(fabs(highl2[1]) - fabs(highl1[1])))
           * angleindex4per;
```

```
magrlow = FL(fabs(lowr1[1])) + (FL(fabs(lowr2[1]) - fabs(lowr1[1]))) *
          angleindex2per;
magrhigh = FL(fabs(highr1[1])) + (FL(fabs(highr2[1]) - fabs(highr1[1])))
           * angleindex4per;
magl = magllow + (maglhigh - magllow) * elevindexhighper;
magr = magrlow + (magrhigh - magrlow) * elevindexhighper;
if(currentphasel[M * irlength + 1] < FL(0.))
  hrtflinterp[1] = - magl;
else
  hrtflinterp[1] = magl;
if(currentphaser[M * irlength + 1] < FL(0.))
  hrtfrinterp[1] = - magr;
else
  hrtfrinterp[1] = magr;

/* other values are complex, in fftw format */
for(i = 2; i < irlength; i += 2)
{
  /* interpolate high and low magnitudes */
  magllow = lowl1[i] + (lowl2[i] - lowl1[i]) * angleindex2per;
  maglhigh = highl1[i] + (highl2[i] - highl1[i]) * angleindex4per;

  magrlow = lowr1[i] + (lowr2[i] - lowr1[i]) * angleindex2per;
  magrhigh = highr1[i] + (highr2[i] - highr1[i]) * angleindex4per;

  /* interpolate high and low results,use current phase */
  magl = magllow +  (maglhigh - magllow) * elevindexhighper;
  phasel = currentphasel[M * irlength + i + 1];

  /* polar to rectangular */
  hrtflinterp[i] = magl * FL(cos(phasel));
  hrtflinterp[i + 1] = magl * FL(sin(phasel));

  magr = magrlow + (magrhigh - magrlow) * elevindexhighper;
  phaser = currentphaser[M * irlength + i + 1];

  hrtfrinterp[i] = magr * FL(cos(phaser));
  hrtfrinterp[i + 1] = magr * FL(sin(phaser));
}

csound->InverseRealFFT(csound, hrtflinterp, irlength);
csound->InverseRealFFT(csound, hrtfrinterp, irlength);

/* wall filters... */
/* all 4 walls are the same! (trivial to make them different...) */
/* x axis, wall1 (left) */
wallreflections = (int)abs((int)(xc * .5 - .25 + (.25 * pow(-1.0,
                            xc))));
/* wall2, x (right) */
wallreflections += (int)abs((int)(xc * .5 + .25 - (.25 * pow(-1.0,
                             xc))));
/* yaxis, wall3 (bottom) */
wallreflections += (int)abs((int)(yc * .5 - .25 + (.25 * pow(-1.0,
                             yc))));
/* yaxis, wall 4 (top) */
wallreflections += (int)abs((int)(yc * .5 + .25 - (.25 * pow(-1.0,
                             yc))));
if(threed)
{
  /* floor (negative z) */
  floorreflections = (int)abs((int)(zc * .5 - .25 + (.25 * pow(-1.0,
                     zc))));
  /* ceiling (positive z) */
  ceilingreflections = (int)abs((int)(zc * .5 + .25 - (.25 * pow(-1.0,
                       zc))));
}

/* fixed parameters on bands etc (to limit no of inputs), but these
   could trivially be variable */
/* note: delay values can be reused: zeroed every time as only used in
   processing hrtf, once every irlength, so not used continuously...*/
```

```
/* if processing was continuous, would need separate mem for each
   filter, store for next run etc...*/
for(i = 0; i < wallreflections; i++)
{
  delsinglel = delsingler = FL(0.0);
  filter(hrtflinterp, p->wallcoefhigh, p->wallcoeflow, &delsinglel,
         irlength, sr);
  filter(hrtfrinterp, p->wallcoefhigh, p->wallcoeflow, &delsingler,
         irlength, sr);
  deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
  band(hrtflinterp, FL(250.0), FL(250.0) / p->q, p->wallg1, deldoublel,
       irlength, sr);
  band(hrtfrinterp, FL(250.0), FL(250.0) / p->q, p->wallg1, deldoubler,
       irlength, sr);
  deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
  band(hrtflinterp, FL(1000.0), FL(1000.0) / p->q, p->wallg2,
       deldoublel, irlength, sr);
  band(hrtfrinterp, FL(1000.0), FL(1000.0) / p->q, p->wallg2,
       deldoubler, irlength, sr);
  deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
  band(hrtflinterp, FL(4000.0), FL(4000.0) / p->q, p->wallg3,
       deldoublel, irlength, sr);
  band(hrtfrinterp, FL(4000.0), FL(4000.0) / p->q, p->wallg3,
       deldoubler, irlength, sr);
}
if(threed)
{
  for(i = 0; i < floorreflections; i++)
  {
    delsinglel = delsingler = FL(0.0);
    filter(hrtflinterp, p->floorcoefhigh, p->floorcoeflow, &delsinglel,
           irlength, sr);
    filter(hrtfrinterp, p->floorcoefhigh, p->floorcoeflow, &delsingler,
           irlength, sr);
    deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
    band(hrtflinterp, FL(250.0), FL(250.0) / p->q, p->floorg1,
         deldoublel, irlength, sr);
    band(hrtfrinterp, FL(250.0), FL(250.0) / p->q, p->floorg1,
         deldoubler, irlength, sr);
    deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
    band(hrtflinterp, FL(1000.q), FL(1000.0) / p->q, p->floorg2,
         deldoublel, irlength, sr);
    band(hrtfrinterp, FL(1000.0), FL(1000.0) / p->q, p->floorg2,
         deldoubler, irlength, sr);
    deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
    band(hrtflinterp, FL(4000.0), FL(4000.0) / p->q, p->floorg3,
         deldoublel, irlength, sr);
    band(hrtfrinterp, FL(4000.0), FL(4000.0) / p->q, p->floorg3,
         deldoubler, irlength, sr);
  }
  for(i = 0; i < ceilingreflections; i++)
  {
    delsinglel = delsingler = FL(0.0);
    filter(hrtflinterp, p->ceilingcoefhigh, p->ceilingcoeflow,
           &delsinglel, irlength, sr);
    filter(hrtfrinterp, p->ceilingcoefhigh, p->ceilingcoeflow,
           &delsingler, irlength, sr);
    deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
    band(hrtflinterp, FL(250.0), FL(250.0) / p->q, p->ceilingg1,
         deldoublel, irlength, sr);
    band(hrtfrinterp, FL(250.0), FL(250.0) / p->q, p->ceilingg1,
         deldoubler, irlength, sr);
    deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
    band(hrtflinterp, FL(1000.0), FL(1000.0) / p->q, p->ceilingg2,
         deldoublel, irlength, sr);
    band(hrtfrinterp, FL(1000.0), FL(1000.0) / p->q, p->ceilingg2,
         deldoubler, irlength, sr);
    deldoublel[0] = deldoublel[1] = deldoubler[0] = deldoubler[1] = 0.0;
    band(hrtflinterp, FL(4000.0), FL(4000.0) / p->q, p->ceilingg3,
         deldoublel, irlength, sr);
    band(hrtfrinterp, FL(4000.0), FL(4000.0) / p->q, p->ceilingg3,
              deldoubler, irlength, sr);
          }
        }

        for(i = 0; i < irlength; i++)
        {
          hrtflpad[i] = hrtflinterp[i];
          hrtfrpad[i] = hrtfrinterp[i];
        }

        for(i = irlength; i < irlengthpad; i++)
        {
          hrtflpad[i] = FL(0.0);
          hrtfrpad[i] = FL(0.0);
        }

        /* back to freq domain */
        csound->RealFFT(csound, hrtflpad, irlengthpad);
        csound->RealFFT(csound, hrtfrpad, irlengthpad);

        /* store */
        for(i = 0; i < irlengthpad; i++)
        {
          hrtflpadspec[M * irlengthpad + i] = hrtflpad[i];
          hrtfrpadspec[M * irlengthpad + i] = hrtfrpad[i];
        }
      }
    }  /* end of source / listener relative change process */

    /* look after overlap add */
    for(i = 0; i < overlapsize ; i++)
    {
      overlapl[i] = outl[M * irlengthpad + i + irlength];
      overlapr[i] = outr[M * irlengthpad + i + irlength];
      if(crossfade)
      {
        overlaplold[i] = outl[M * irlengthpad + i + irlength];
        overlaprold[i] = outr[M * irlengthpad + i + irlength];
      }
      /* overlap will be previous fading out signal */
      if(cross[M])
      {
        overlaplold[i] = outlold[M * irlengthpad + i + irlength];
        overlaprold[i] = outrold[M * irlengthpad + i + irlength];
      }
    }

    /* insert insig */
    for (i = 0; i <  irlength; i++)
      inbufpad[i] = inbuf[i];

    for (i = irlength; i <  irlengthpad; i++)
      inbufpad[i] = FL(0.0);

    csound->RealFFT(csound, inbufpad, irlengthpad);

    for(i = 0; i < irlengthpad; i ++)
    {
      hrtflpad[i] = hrtflpadspec[M * irlengthpad + i];
      hrtfrpad[i] = hrtfrpadspec[M * irlengthpad + i];
    }

    /* convolution: spectral multiplication */
    csound->RealFFTMult(csound, outlspec, hrtflpad, inbufpad, irlengthpad,
                        FL(1.0));
    csound->RealFFTMult(csound, outrspec, hrtfrpad, inbufpad, irlengthpad,
                        FL(1.0));

    csound->InverseRealFFT(csound, outlspec, irlengthpad);
    csound->InverseRealFFT(csound, outrspec, irlengthpad);
```

```
/* scale */
for(i = 0; i < irlengthpad; i++)
{
  outlspec[i] = outlspec[i]/(sr/FL(38000.0));
  outrspec[i] = outrspec[i]/(sr/FL(38000.0));
}

/* store */
for(i = 0; i < irlengthpad; i++)
{
  outl[M * irlengthpad + i] = outlspec[i];
  outr[M * irlengthpad + i] = outrspec[i];
}

/* setup for fades */
if(crossfade || cross[M])
{
  crossout = 1;

  /* need to put these values into a buffer for processing */
  for(i = 0; i < irlengthpad; i++)
  {
    hrtflpadold[i] = hrtflpadspecold[M * irlengthpad + i];
    hrtfrpadold[i] = hrtfrpadspecold[M * irlengthpad + i];
  }

  /* convolution */
  csound->RealFFTMult(csound, outlspecold, hrtflpadold, inbufpad,
                      irlengthpad, FL(1.0));
  csound->RealFFTMult(csound, outrspecold, hrtfrpadold, inbufpad,
                       irlengthpad, FL(1.0));

  /* ifft, back to time domain */
  csound->InverseRealFFT(csound, outlspecold, irlengthpad);
  csound->InverseRealFFT(csound, outrspecold, irlengthpad);

  /* scale */
  for(i = 0; i < irlengthpad; i++)
  {
    outlspecold[i] = outlspecold[i]/(sr/FL(38000.0));
    outrspecold[i] = outrspecold[i]/(sr/FL(38000.0));
  }

  /* o/p real values */
  for(i = 0; i < irlengthpad; i++)
  {
    outlold[M * irlengthpad + i] = outlspecold[i];
    outrold[M * irlengthpad + i] = outrspecold[i];
  }

  cross[M]++;
  cross[M] = cross[M] % fade;
}

if(crossout)
  for(i = 0; i < irlength; i++)
  {
    predell[i + M * irlength] = (outlspecold[i] + (i < overlapsize ?
      overlaplold[i] : FL(0.0))) * FL(1. - (FL(l[M]) / fadebuffer)) +
      (outlspec[i] + (i < overlapsize ? overlapl[i] : FL(0.0))) * (FL(l[M])
      / fadebuffer);
    predelr[i + M * irlength] = (outrspecold[i] + (i < overlapsize ?
      overlaprold[i] : FL(0.0))) * FL(1. - (FL(l[M]) / fadebuffer)) +
      (outrspec[i] + (i < overlapsize ? overlapr[i] : FL(0.0))) * (FL(l[M])
      / fadebuffer);
    l[M]++;
  }
else
  for(i = 0; i < irlength; i++)
  {
    predell[i + M * irlength] = outlspec[i] + (i < overlapsize ? overlapl[i]
                                            : FL(0.0));
                predelr[i + M * irlength] = outrspec[i] + (i < overlapsize ? overlapr[i]
                                            : FL(0.0));
              }

            M++;
            M = M % impulses;

          } /* z */
        } /* y */
      } /* x */

      counter = 0;
      /* need to store these values here, as storing them after check would not allow
         each impulse to be processed! */
      p->srcxv = srcx;
      p->srcyv = srcy;
      p->srczv = srcz;
      p->lstnrxv = lstnrx;
      p->lstnryv = lstnry;
      p->lstnrzv = lstnrz;
      p->rotatev = rotate;

    }  /* end of counter == irlength */

    /* update */
    p->counter = counter;
    p->initialfade = initialfade;

  } /* end of ksmps loop */

  return OK;
}

static OENTRY localops[] =
{
  {
    "hrtfearly",   sizeof(early), 5, "aaiii", "axxxxxxSSioopoOoooooooooooooooooo",
    (SUBR)early_init, NULL, (SUBR)early_process
  }
};

LINKAGE
```

3.2 hrtfreverb

```c
/*
Brian Carty
PhD Code August 2010
binaural reverb: diffuse field
*/

#include "csdl.h"
#define SQUARE(X) (X)*(X)

/* endian issues: swap bytes for ppc */
#ifdef WORDS_BIGENDIAN
static int swap4bytes(CSOUND* csound, MEMFIL* mfp)
{
  char c1, c2, c3, c4;
  char *p = mfp->beginp;
  int  size = mfp->length;

  while (size >= 4)
  {
    c1 = p[0]; c2 = p[1]; c3 = p[2]; c4 = p[3];
    p[0] = c4; p[1] = c3; p[2] = c2; p[3] = c1;
    size -= 4; p +=4;
  }

  return OK;
}
#else
static int (*swap4bytes)(CSOUND*, MEMFIL*) = NULL;
#endif

/* matrices for feedback delay network (fdn) */
#define mthird -1.f / 3
#define tthird 2.f / 3
#define msix -1.f / 6
#define fsix 5.f / 6
#define mtw -1.f / 12
#define etw 11.f / 12

static const MYFLT matrix6[36] =
  {tthird,mthird,mthird,mthird,mthird,mthird,
   mthird,tthird,mthird,mthird,mthird,mthird,
   mthird,mthird,tthird,mthird,mthird,mthird,
   mthird,mthird,mthird,tthird,mthird,mthird,
   mthird,mthird,mthird,mthird,tthird,mthird,
   mthird,mthird,mthird,mthird,mthird,tthird};

static const MYFLT matrix12[144] =
  {fsix,msix,msix,msix,msix,msix,msix,msix,msix,msix,msix,msix,
   msix,fsix,msix,msix,msix,msix,msix,msix,msix,msix,msix,msix,
   msix,msix,fsix,msix,msix,msix,msix,msix,msix,msix,msix,msix,
   msix,msix,msix,fsix,msix,msix,msix,msix,msix,msix,msix,msix,
   msix,msix,msix,msix,fsix,msix,msix,msix,msix,msix,msix,msix,
   msix,msix,msix,msix,msix,fsix,msix,msix,msix,msix,msix,msix,
   msix,msix,msix,msix,msix,msix,fsix,msix,msix,msix,msix,msix,
   msix,msix,msix,msix,msix,msix,msix,fsix,msix,msix,msix,msix,
   msix,msix,msix,msix,msix,msix,msix,msix,fsix,msix,msix,msix,
   msix,msix,msix,msix,msix,msix,msix,msix,msix,fsix,msix,msix,
   msix,msix,msix,msix,msix,msix,msix,msix,msix,msix,fsix,msix,
   msix,msix,msix,msix,msix,msix,msix,msix,msix,msix,msix,fsix};

static const MYFLT matrix24[576] =

{etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
```

```
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,etw,mtw,mtw
,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,etw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw,mtw
,mtw,mtw,mtw,etw};

/* for delay line lengths */
static const int primes[229] =
  {
    17,     23,     59,     71,     113,    127,    163,    191,    211,    229,
    271,    283,    313,    337,    359,    373,    409,    461,    541,    587,
    631,    691,    709,    773,    829,    863,    919,    971,    1039,   1069,
    1123,   1171,   1217,   1259,   1303,   1373,   1423,   1483,   1511,   1597,
    1627,   1669,   1733,   1787,   1847,   1867,   1913,   1951,   2027,   2081,
    2131,   2179,   2213,   2269,   2333,   2383,   2423,   2467,   2531,   2579,
    2617,   2671,   2729,   2789,   2837,   2861,   2917,   2999,   3011,   3083,
    3121,   3169,   3209,   3259,   3331,   3389,   3449,   3469,   3533,   3571,
    3613,   3671,   3727,   3779,   3821,   3889,   3917,   3989,   4001,   4051,
    4111,   4177,   4231,   4271,   4337,   4391,   4447,   4483,   4517,   4567,
    4621,   4691,   4733,   4787,   4817,   4861,   4919,   4967,   5023,   5077,
    5113,   5167,   5233,   5297,   5309,   5351,   5441,   5483,   5507,   5563,
    5641,   5683,   5711,   5783,   5821,   5857,   5927,   5981,   6011,   6067,
    6121,   6173,   6217,   6271,   6317,   6361,   6421,   6473,   6529,   6581,
    6607,   6661,   6733,   6793,   6841,   6883,   6911,   6961,   7027,   7057,
    7109,   7177,   7211,   7297,   7349,   7393,   7417,   7481,   7523,   7561,
    7607,   7673,   7717,   7789,   7841,   7879,   7919,   7963,   8017,   8081,
    8111,   8167,   8209,   8287,   8317,   8377,   8443,   8467,   8521,   8563,
    8623,   8677,   8713,   8761,   8831,   8867,   8923,   8963,   9013,   9059,
    9109,   9187,   9221,   9257,   9323,   9371,   9413,   9461,   9511,   9587,
    9631,   9679,   9721,   9781,   9803,   9859,   9949,   9973,   10039,  10079,
    10111,  10177,  10211,  10259,  10333,  10391,  10429,  10459,  10513,  10589,
    10607,  10663,  10711,  10799,  10831,  10859,  10909,  10979,  11003
  };

typedef struct
{
  OPDS h;
  /* in / out */
  /* outputs l/r and delay required for late del...*/
  MYFLT *outsigl, *outsigr, *idel;
  /* mean free path and order are optional, meanfp defaults to medium room, opcode
     can be used as stand alone binaural reverb, or spatially accurate taking meanfp and
     order from earlies opcode */
  MYFLT *insig, *ilowrt60, *ihighrt60, *ifilel, *ifiler, *osr, *omeanfp, *porder;

  /* internal data / class variables */
  MYFLT delaytime;
  int delaytimeint, basedelay;

  /* number of delay lines */
  int M;

  /* delay line iterators */
  int u, v, w, x, y, z;
  int ut, vt, wt, xt, yt, zt;
  int utf1, vtf1, wtf1, xtf1, ytf1, ztf1;
  int utf2, vtf2, wtf2, xtf2, ytf2, ztf2;

  /* buffer lengths, change for different sr */
  int irlength;
  int irlengthpad;
  int overlapsize;

  /* memory buffers: delays */
  AUXCH delays;
  /* filter coeffs */
  AUXCH gi, ai;
  /* matrix manipulations */
  AUXCH inmat, inmatlp, dellp, outmat;
  /* delays */
  AUXCH del1, del2, del3, del4, del5, del6;
  AUXCH del1t, del2t, del3t, del4t, del5t, del6t;
  AUXCH del1tf, del2tf, del3tf, del4tf, del5tf, del6tf, del7tf, del8tf, del9tf, del10tf,
        del11tf, del12tf;
  /* filter variables, spectral manipulations */
  AUXCH power, HRTFave, num, denom, cohermags, coheru, coherv;
  AUXCH filtout, filtuout, filtvout, filtpad, filtupad, filtvpad;

  /* output of matrix cycle, with IIRs in combs and FIR tone, then l and r o/p processed
     with u and v coherence filters */
  /* with overlap buffers for overlap add convolution */
  AUXCH matrixlu, matrixrv;
  AUXCH olmatrixlu, olmatrixrv;
  /* above processed with hrtf l and r filters */
  /* with overlap buffers for overlap add convolution */
  AUXCH hrtfl, hrtfr;
  AUXCH olhrtfl, olhrtfr;
  /* filter coeff */
  MYFLT b;
  /* 1st order FIR mem */
  MYFLT inoldl, inoldr;
  /* for storing hrtf data used to create filters */
  AUXCH buffl, buffr;

  /* counter */
  int counter;

  MYFLT sr;

}hrtfreverb;

int hrtfreverb_init(CSOUND *csound, hrtfreverb *p)
{
  /* left and right data files: spectral mag, phase format */
  MEMFIL *fpl = NULL, *fpr = NULL;
  char filel[MAXNAME],filer[MAXNAME];
  /* files contain floats */
  float *fpindexl, *fpindexr;

  /* processing sizes */
  int irlength, irlengthpad, overlapsize;

  /* pointers used to fill buffers in data structure */
  int *delaysp;
  MYFLT *gip, *aip;
  MYFLT *powerp, *HRTFavep, *nump, *denomp, *cohermagsp, *coherup, *cohervp;
  MYFLT *filtoutp, *filtuoutp, *filtvoutp, *filtpadp, *filtupadp, *filtvpadp;
  MYFLT *bufflp, *buffrp;

  /* iterators, file skip */
  int i, j;
  int skip = 0;
  int skipdouble = 0;
```

```c
/* used in choice of delay line lengths */
int basedelay;

/* local filter variables for spectral manipulations */
MYFLT rel, rer, retemp, iml, imr, imtemp;

/* setup filters */
MYFLT T, alpha, aconst, exp;
int clipcheck = 0;

MYFLT sr = (MYFLT)*p->osr;
MYFLT meanfp = (MYFLT)*p->omeanfp;
int order = (int)*p->porder;

/* delay line variables */
MYFLT delaytime, meanfporder;
int delaytimeint;
int Msix, Mtwelve, Mtwentyfour;
int meanfpsamps, meanfpordersamps;
int test;

MYFLT rt60low = (MYFLT)*p->ilowrt60;
MYFLT rt60high = (MYFLT)*p->ihighrt60;

int M;

/* sr, defualt 44100 */
if(sr != 44100 && sr != 48000 && sr != 96000)
  sr = 44100;
p->sr = sr;

if (UNLIKELY(csound->esr != sr))
  csound->Message(csound, Str("\n\nWARNING!!:\nOrchestra SR not compatible with HRTF
                  processing SR of: %.0f\n\n"), sr);

/* meanfp: defaults to room size 10 * 10 * 3 (max of 1: v. large room, min according
   to min room dimensions in early: 2 * 2 * 2) */
if(meanfp <= 0.003876 || meanfp > 1)
  meanfp = FL(0.0109);

/* order: defaults to 1 (4 is max for earlies) */
if(order < 0 || order > 4)
  order = 1;

/* rt60 values must be positive and non zero */
if(rt60low <= 0)
  rt60low = FL(0.01);

if(rt60high <= 0)
  rt60high = FL(0.01);

/* setup as per sr */
if(sr == 44100 || sr == 48000)
{
  irlength = 128;
  irlengthpad = 256;
  overlapsize = (irlength - 1);
}
else if(sr == 96000)
{
  irlength = 256;
  irlengthpad = 512;
  overlapsize = (irlength - 1);
}

/* copy in string name... */
strcpy(filel, (char*) p->ifilel);
strcpy(filer, (char*) p->ifiler);

/* reading files, with byte swap */
if (UNLIKELY((fpl = csound->ldmemfile2withCB(csound, filel, CSFTYPE_FLOATS_BINARY,
    swap4bytes)) == NULL))
return
  csound->InitError(csound, Str("\n\n\nCannot load left data file, exiting\n\n"));

if (UNLIKELY((fpr = csound->ldmemfile2withCB(csound, filer, CSFTYPE_FLOATS_BINARY,
    swap4bytes)) == NULL))
return
  csound->InitError(csound, Str("\n\n\nCannot load right data file, exiting\n\n"));

/* do not need to be in p, as only used in init */
fpindexl = (float *)fpl->beginp;
fpindexr = (float *)fpr->beginp;

/* setup structure values */
p->irlength = irlength;
p->irlengthpad = irlengthpad;
p->overlapsize = overlapsize;

/* allocate memory */
if (!p->power.auxp || p->power.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->power);
if (!p->HRTFave.auxp || p->HRTFave.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->HRTFave);
if (!p->num.auxp || p->num.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->num);
if (!p->denom.auxp || p->denom.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->denom);
if (!p->cohermags.auxp || p->cohermags.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->cohermags);
if (!p->coheru.auxp || p->coheru.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->coheru);
if (!p->coherv.auxp || p->coherv.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->coherv);

if (!p->filtout.auxp || p->filtout.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->filtout);
if (!p->filtuout.auxp || p->filtuout.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->filtuout);
if (!p->filtvout.auxp || p->filtvout.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->filtvout);
if (!p->filtpad.auxp || p->filtpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->filtpad);
if (!p->filtupad.auxp || p->filtupad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->filtupad);
if (!p->filtvpad.auxp || p->filtvpad.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->filtvpad);

/* zero numerator and power buffer, as they accumulate */
memset(p->power.auxp, 0, irlength * sizeof(MYFLT));
memset(p->num.auxp, 0, irlength * sizeof(MYFLT));
/* no need to zero other above mem, as it will be filled in init */

if (!p->matrixlu.auxp || p->matrixlu.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->matrixlu);
if (!p->matrixrv.auxp || p->matrixrv.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->matrixrv);
if (!p->olmatrixlu.auxp || p->olmatrixlu.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize * sizeof(MYFLT), &p->olmatrixlu);
if (!p->olmatrixrv.auxp || p->olmatrixrv.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize * sizeof(MYFLT), &p->olmatrixrv);
if (!p->hrtfl.auxp || p->hrtfl.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->hrtfl);
if (!p->hrtfr.auxp || p->hrtfr.size < irlengthpad * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlengthpad * sizeof(MYFLT), &p->hrtfr);
if (!p->olhrtfl.auxp || p->olhrtfl.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize * sizeof(MYFLT), &p->olhrtfl);
if (!p->olhrtfr.auxp || p->olhrtfr.size < overlapsize * sizeof(MYFLT))
  csound->AuxAlloc(csound, overlapsize * sizeof(MYFLT), &p->olhrtfr);

memset(p->matrixlu.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->matrixrv.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->olmatrixlu.auxp, 0, overlapsize * sizeof(MYFLT));
memset(p->olmatrixrv.auxp, 0, overlapsize * sizeof(MYFLT));
memset(p->hrtfl.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->hrtfr.auxp, 0, irlengthpad * sizeof(MYFLT));
memset(p->olhrtfl.auxp, 0, overlapsize * sizeof(MYFLT));
memset(p->olhrtfr.auxp, 0, overlapsize * sizeof(MYFLT));

if (!p->buffl.auxp || p->buffl.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->buffl);
if (!p->buffr.auxp || p->buffr.size < irlength * sizeof(MYFLT))
  csound->AuxAlloc(csound, irlength * sizeof(MYFLT), &p->buffr);

memset(p->buffl.auxp, 0, irlength * sizeof(MYFLT));
memset(p->buffr.auxp, 0, irlength * sizeof(MYFLT));

/* buffers to store hrtf data */
bufflp = (MYFLT *)p->buffl.auxp;
buffrp = (MYFLT *)p->buffr.auxp;

/* 0 delay iterators */
p->u = p->v = p->w = p->x = p->y = p->z = 0;
p->ut = p->vt = p->wt = p->xt = p->yt = p->zt = 0;
p->utf1 = p->vtf1 = p->wtf1 = p->xtf1 = p->ytf1 = p->ztf1 = 0;
p->utf2 = p->vtf2 = p->wtf2 = p->xtf2 = p->ytf2 = p->ztf2 = 0;

/* calculate delayline lengths */
meanfporder = meanfp * (order + 1);
meanfpsamps = (int)(meanfp * sr);
meanfpordersamps = (int)(meanfporder * sr);

/* setup reverb time */
delaytime = rt60low > rt60high ? rt60low : rt60high;

/* in samples */
delaytime *= sr;
/* schroeder suggests 0.15 modes per Hz, so M should be > 0.15 t60 */
delaytime /= 7;

/* which no. of delay lines implies ave delay nearest to mfp(which is an appropriate
   ave)? */
Msix = abs((int)(delaytime / 6) - meanfpsamps);
Mtwelve = abs((int)(delaytime / 12) - meanfpsamps);
Mtwentyfour = abs((int)(delaytime / 24) - meanfpsamps);
M = Mtwelve < Mtwentyfour ? (Msix < Mtwelve ? 6 : 12) : 24;

delaytime /= M;
delaytimeint = (int)delaytime;

if(delaytimeint < meanfpsamps)
  delaytimeint = meanfpsamps;

/* maximum value, according to primes array and delay line allocation */
if(delaytimeint > 10112)
  delaytimeint = 10112;

/* minimum values, according to primes array and delay line allocation */
if(M==6)
{
  if(delaytimeint < 164)
    delaytimeint = 164;
}
if(M==12)
{
  if(delaytimeint < 374)
    delaytimeint = 374;
}
if(M==24)
{
  if(delaytimeint < 410)
    delaytimeint = 410;
}
```

```
/* allocate memory based on M: number of delays */
if (!p->delays.auxp || p->delays.size < M * sizeof(int))
  csound->AuxAlloc(csound, M * sizeof(int), &p->delays);
if (!p->gi.auxp || p->gi.size < M * sizeof(MYFLT))
  csound->AuxAlloc(csound, M * sizeof(MYFLT), &p->gi);
if (!p->ai.auxp || p->ai.size < M * sizeof(MYFLT))
  csound->AuxAlloc(csound, M * sizeof(MYFLT), &p->ai);
if (!p->inmat.auxp || p->inmat.size < M * sizeof(MYFLT))
  csound->AuxAlloc(csound, M * sizeof(MYFLT), &p->inmat);
if (!p->inmatlp.auxp || p->inmatlp.size < M * sizeof(MYFLT))
  csound->AuxAlloc(csound, M * sizeof(MYFLT), &p->inmatlp);
if (!p->dellp.auxp || p->dellp.size < M * sizeof(MYFLT))
  csound->AuxAlloc(csound, M * sizeof(MYFLT), &p->dellp);
if (!p->outmat.auxp || p->outmat.size < M * sizeof(MYFLT))
  csound->AuxAlloc(csound, M * sizeof(MYFLT), &p->outmat);

memset(p->delays.auxp, 0, M * sizeof(int));
memset(p->gi.auxp, 0, M * sizeof(MYFLT));
memset(p->ai.auxp, 0, M * sizeof(MYFLT));
memset(p->inmat.auxp, 0, M * sizeof(MYFLT));
memset(p->inmatlp.auxp, 0, M * sizeof(MYFLT));
memset(p->dellp.auxp, 0, M * sizeof(MYFLT));
memset(p->outmat.auxp, 0, M * sizeof(MYFLT));

/* choose appropriate base delay times */
for(i = 0; i < 212; i++)
{
  if(M == 6)
    test = (i > 6 ? i : 6) - 6;
  else if(M == 12)
    test = (i > 15 ? i : 15) - 15;
  else
    test = (i > 16 ? i : 16) - 16;

  if(primes[i] > delaytimeint || primes[test] > meanfpordersamps)
  {
    basedelay = i - 1;
    if(primes[test] > meanfpordersamps)
      csound->Message(csound, "\nfdn delay > earlies del..., fixed!");
    *p->idel = FL(meanfpordersamps - primes[test - 1]) / sr;
    break;
  }
}

delaysp = (int *)p->delays.auxp;

/* fill delay data, note this data can be filled locally */
delaysp[0] = primes[basedelay];
delaysp[1] = primes[basedelay + 3];
delaysp[2] = primes[basedelay - 3];
delaysp[3] = primes[basedelay + 6];
delaysp[4] = primes[basedelay - 6];
delaysp[5] = primes[basedelay + 9];
if(M ==12 || M==24)
{
  delaysp[6] = primes[basedelay - 9];
  delaysp[7] = primes[basedelay + 12];
  delaysp[8] = primes[basedelay - 12];
  delaysp[9] = primes[basedelay + 15];
  delaysp[10] = primes[basedelay - 15];
  delaysp[11] = primes[basedelay + 18];
}
if(M ==24)
{
  /* fill in gaps... */
  delaysp[12] = primes[basedelay + 1];
  delaysp[13] = primes[basedelay - 1];
  delaysp[14] = primes[basedelay + 4];
  delaysp[15] = primes[basedelay - 4];
  delaysp[16] = primes[basedelay + 7];
  delaysp[17] = primes[basedelay - 7];
  delaysp[18] = primes[basedelay + 10];
  delaysp[19] = primes[basedelay - 10];
  delaysp[20] = primes[basedelay + 13];
  delaysp[21] = primes[basedelay - 13];
  delaysp[22] = primes[basedelay + 16];
  delaysp[23] = primes[basedelay - 16];
}

/* setup and zero delay lines */
if (!p->del1.auxp || p->del1.size < delaysp[0] * sizeof(MYFLT))
  csound->AuxAlloc(csound, delaysp[0] * sizeof(MYFLT), &p->del1);
if (!p->del2.auxp || p->del2.size < delaysp[1] * sizeof(MYFLT))
  csound->AuxAlloc(csound, delaysp[1] * sizeof(MYFLT), &p->del2);
if (!p->del3.auxp || p->del3.size < delaysp[2] * sizeof(MYFLT))
  csound->AuxAlloc(csound, delaysp[2] * sizeof(MYFLT), &p->del3);
if (!p->del4.auxp || p->del4.size < delaysp[3] * sizeof(MYFLT))
  csound->AuxAlloc(csound, delaysp[3] * sizeof(MYFLT), &p->del4);
if (!p->del5.auxp || p->del5.size < delaysp[4] * sizeof(MYFLT))
  csound->AuxAlloc(csound, delaysp[4] * sizeof(MYFLT), &p->del5);
if (!p->del6.auxp || p->del6.size < delaysp[5] * sizeof(MYFLT))
  csound->AuxAlloc(csound, delaysp[5] * sizeof(MYFLT), &p->del6);

memset(p->del1.auxp, 0, delaysp[0] * sizeof(MYFLT));
memset(p->del2.auxp, 0, delaysp[1] * sizeof(MYFLT));
memset(p->del3.auxp, 0, delaysp[2] * sizeof(MYFLT));
memset(p->del4.auxp, 0, delaysp[3] * sizeof(MYFLT));
memset(p->del5.auxp, 0, delaysp[4] * sizeof(MYFLT));
memset(p->del6.auxp, 0, delaysp[5] * sizeof(MYFLT));

/* if 12 delay lines required */
if(M == 12 || M==24)
{
  if (!p->del1t.auxp || p->del1t.size < delaysp[6] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[6] * sizeof(MYFLT), &p->del1t);
  if (!p->del2t.auxp || p->del2t.size < delaysp[7] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[7] * sizeof(MYFLT), &p->del2t);
  if (!p->del3t.auxp || p->del3t.size < delaysp[8] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[8] * sizeof(MYFLT), &p->del3t);
  if (!p->del4t.auxp || p->del4t.size < delaysp[9] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[9] * sizeof(MYFLT), &p->del4t);
  if (!p->del5t.auxp || p->del5t.size < delaysp[10] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[10] * sizeof(MYFLT), &p->del5t);
  if (!p->del6t.auxp || p->del6t.size < delaysp[11] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[11] * sizeof(MYFLT), &p->del6t);

  memset(p->del1t.auxp, 0, delaysp[6] * sizeof(MYFLT));
  memset(p->del2t.auxp, 0, delaysp[7] * sizeof(MYFLT));
  memset(p->del3t.auxp, 0, delaysp[8] * sizeof(MYFLT));
  memset(p->del4t.auxp, 0, delaysp[9] * sizeof(MYFLT));
  memset(p->del5t.auxp, 0, delaysp[10] * sizeof(MYFLT));
  memset(p->del6t.auxp, 0, delaysp[11] * sizeof(MYFLT));
}
if(M==24)
{
  if (!p->del1tf.auxp || p->del1tf.size < delaysp[12] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[12] * sizeof(MYFLT), &p->del1tf);
  if (!p->del2tf.auxp || p->del2tf.size < delaysp[13] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[13] * sizeof(MYFLT), &p->del2tf);
  if (!p->del3tf.auxp || p->del3tf.size < delaysp[14] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[14] * sizeof(MYFLT), &p->del3tf);
  if (!p->del4tf.auxp || p->del4tf.size < delaysp[15] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[15] * sizeof(MYFLT), &p->del4tf);
  if (!p->del5tf.auxp || p->del5tf.size < delaysp[16] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[16] * sizeof(MYFLT), &p->del5tf);
  if (!p->del6tf.auxp || p->del6tf.size < delaysp[17] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[17] * sizeof(MYFLT), &p->del6tf);
  if (!p->del7tf.auxp || p->del7tf.size < delaysp[18] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[18] * sizeof(MYFLT), &p->del7tf);
  if (!p->del8tf.auxp || p->del8tf.size < delaysp[19] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[19] * sizeof(MYFLT), &p->del8tf);
  if (!p->del9tf.auxp || p->del9tf.size < delaysp[20] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[20] * sizeof(MYFLT), &p->del9tf);
  if (!p->del10tf.auxp || p->del10tf.size < delaysp[21] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[21] * sizeof(MYFLT), &p->del10tf);
  if (!p->del11tf.auxp || p->del11tf.size < delaysp[22] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[22] * sizeof(MYFLT), &p->del11tf);
  if (!p->del12tf.auxp || p->del12tf.size < delaysp[23] * sizeof(MYFLT))
    csound->AuxAlloc(csound, delaysp[23] * sizeof(MYFLT), &p->del12tf);

  memset(p->del1tf.auxp, 0, delaysp[12] * sizeof(MYFLT));
  memset(p->del2tf.auxp, 0, delaysp[13] * sizeof(MYFLT));
  memset(p->del3tf.auxp, 0, delaysp[14] * sizeof(MYFLT));
  memset(p->del4tf.auxp, 0, delaysp[15] * sizeof(MYFLT));
  memset(p->del5tf.auxp, 0, delaysp[16] * sizeof(MYFLT));
  memset(p->del6tf.auxp, 0, delaysp[17] * sizeof(MYFLT));
  memset(p->del7tf.auxp, 0, delaysp[18] * sizeof(MYFLT));
  memset(p->del8tf.auxp, 0, delaysp[19] * sizeof(MYFLT));
  memset(p->del9tf.auxp, 0, delaysp[20] * sizeof(MYFLT));
  memset(p->del10tf.auxp, 0, delaysp[21] * sizeof(MYFLT));
  memset(p->del11tf.auxp, 0, delaysp[22] * sizeof(MYFLT));
  memset(p->del12tf.auxp, 0, delaysp[23] * sizeof(MYFLT));
}

powerp = (MYFLT *)p->power.auxp;
HRTFavep = (MYFLT *)p->HRTFave.auxp;
nump = (MYFLT *)p->num.auxp;
denomp = (MYFLT *)p->denom.auxp;
cohermagsp = (MYFLT *)p->cohermags.auxp;
coherup = (MYFLT *)p->coheru.auxp;
cohervp = (MYFLT *)p->coherv.auxp;

/* usually, just go through all files; in this case, just doubled due to symmetry
   (with exceptions, as below) */
for(i = 0; i < 368; i ++)
{
  /* if at a measurement where no doubling for symmetry necessary... */
  if(i == 0 || i == 28 || i == 29 || i == 59 || i == 60 || i == 96  || i == 97 || i ==
     133 || i == 134 || i == 170 || i == 171 || i == 207 || i == 208 || i == 244  || i
     == 245 || i == 275 || i == 276 || i == 304  || i == 305 || i == 328 || i == 346
     || i == 347 || i == 359 || i == 360 || i == 366 || i == 367)
    skipdouble = 1;
  else
    skipdouble = 0;

  for(j = 0; j < irlength; j ++)
  {
    bufflp[j] = fpindexl[skip + j];
    buffrp[j] = fpindexr[skip + j];
  }

  /* deal with 0 hz and nyq: may be a negative real val, no need for fabs() as
     squaring anyway! */
  /* skipdouble: l = r */
  if(skipdouble)
  {
    powerp[0] = powerp[0] + FL(SQUARE(bufflp[0]));
    powerp[1] = powerp[1] + FL(SQUARE(bufflp[1]));
  }
  /* include both */
  else
  {
    powerp[0] = powerp[0] + FL(SQUARE(bufflp[0])) + FL(SQUARE(buffrp[0]));
    powerp[1] = powerp[1] + FL(SQUARE(bufflp[1])) + FL(SQUARE(buffrp[1]));
  }

  for(j = 2; j < irlength; j += 2)
  {
    if(skipdouble)
      powerp[j] = powerp[j] + (MYFLT)SQUARE(bufflp[j]);
    else
      powerp[j] = powerp[j] + (MYFLT)SQUARE(bufflp[j]) + (MYFLT)SQUARE(buffrp[j]);
    powerp[j + 1] = FL(0.0);
  }
  skip += irlength;
}

for(i = 0; i < irlength; i++)
  HRTFavep[i] = FL(sqrt(powerp[i] / 710));

fpindexl = (float *)fpl->beginp;
fpindexr = (float *)fpr->beginp;
skip = 0;

/* coherence values */
for(i = 0; i< 368; i++)
{
  /* if at a measurement where no doubling for symmetry necessary... */
  if(i == 0 || i == 28 || i == 29 || i == 59 || i == 60 || i == 96  || i == 97 || i =
     133 || i == 134 || i == 170 || i == 171 || i == 207 || i == 208 || i == 244  || i
     == 245 || i == 275 || i == 276 || i == 304  || i == 305 || i == 328 || i == 346
     || i == 347 || i == 359 || i == 360 || i == 366 || i == 367)
    skipdouble = 1;
  else
    skipdouble = 0;

  for(j = 0; j < irlength; j ++)
  {
    bufflp[j] = fpindexl[skip + j];
    buffrp[j] = fpindexr[skip + j];
  }

  /* back to rectangular to find numerator: need complex nos */
  /* 0Hz and Nyq ok as real */
  if(skipdouble)
  {
    nump[0] = nump[0] + (bufflp[0] * buffrp[0]);
    nump[1] = nump[1] + (bufflp[1] * buffrp[1]);
  }
  else
  {
    nump[0] = nump[0] + (bufflp[0] * buffrp[0]) + (buffrp[0] * bufflp[0]);
    nump[1] = nump[1] + (bufflp[1] * buffrp[1]) + (buffrp[1] * bufflp[1]);
  }

  /* complex multiplication */
  /* (a + i b)(c + i d) */
  /* = (a c - b d) + i(a d + b c) */
  /* conjugate: d becomes -d -> = (a c + b d) + i(- a d + b c) */
  /* doing l * conj r and r * conj l here, as dataset symmetrical...for non-
     symmetrical, just go through all and do l * conj r */
  for(j = 2; j < irlength; j += 2)
  {
    rel = bufflp[j] * (MYFLT)cos(bufflp[j + 1]);
    iml = bufflp[j] * (MYFLT)sin(bufflp[j + 1]);
    rer = buffrp[j] * (MYFLT)cos(buffrp[j + 1]);
    imr = buffrp[j] * (MYFLT)sin(buffrp[j + 1]);
    if(skipdouble)
    {
      nump[j] = nump[j] + ((rel * rer) + (iml * imr));
      nump[j + 1] = nump[j + 1] + ((rel * -imr) + (iml * rer));
    }
    else
    {
      nump[j] = nump[j] + ((rel * rer) + (iml * imr)) + ((rer * rel) + (imr * iml));
      nump[j + 1] = nump[j + 1] + ((rel * -imr) + (iml * rer)) + ((rer * -iml) + (imr
                    * rel));
    }
  }
  skip += irlength;
}

/* 0 & nyq = fabs() for mag... */
```

```
nump[0] = FL(fabs(nump[0]));
nump[1] = FL(fabs(nump[1]));

/* magnitudes of sum of conjugates */
for(i = 2; i < irlength; i += 2)
{
  retemp = nump[i];
  imtemp = nump[i + 1];
  nump[i] = FL(sqrt(SQUARE(retemp) + SQUARE(imtemp)));
  nump[i + 1] = FL(0.0);
}

/* sqrt (powl * powr) powl = powr in symmetric case, so just power[] needed */
for(i = 0; i < irlength; i++)
  denomp[i] = powerp[i];

/* coherence values */
cohermagsp[0] = nump[0] / denomp[0];
cohermagsp[1] = nump[1] / denomp[1];

for(i = 2; i < irlength; i += 2)
{
  cohermagsp[i] = nump[i] / denomp[i];
  cohermagsp[i+1] = FL(0.0);
}

/* coherence formula */
coherup[0] = FL(sqrt((1.0 + cohermagsp[0]) / 2.0));
coherup[1] = FL(sqrt((1.0 + cohermagsp[1]) / 2.0));
cohervp[0] = FL(sqrt((1.0 - cohermagsp[0]) / 2.0));
cohervp[1] = FL(sqrt((1.0 - cohermagsp[1]) / 2.0));

for(i = 2; i < irlength; i += 2)
{
  coherup[i] = FL(sqrt((1.0 + cohermagsp[i]) / 2.0));
  cohervp[i] = FL(sqrt((1.0 - cohermagsp[i]) / 2.0));
  coherup[i + 1] = FL(0.0);
  cohervp[i + 1] = FL(0.0);
}

/* no need to go back to rectangular for fft, as phase = 0, so same */
csound->InverseRealFFT(csound, HRTFavep, irlength);
csound->InverseRealFFT(csound, coherup, irlength);
csound->InverseRealFFT(csound, cohervp, irlength);

filtoutp = (MYFLT *)p->filtout.auxp;
filtuoutp = (MYFLT *)p->filtuout.auxp;
filtvoutp = (MYFLT *)p->filtvout.auxp;
filtpadp = (MYFLT *)p->filtpad.auxp;
filtupadp = (MYFLT *)p->filtupad.auxp;
filtvpadp = (MYFLT *)p->filtvpad.auxp;

/* shift */
for(i = 0; i < irlength; i++)
{
  filtoutp[i] = HRTFavep[(i + (irlength / 2)) % irlength];
  filtuoutp[i] = coherup[(i + (irlength / 2)) % irlength];
  filtvoutp[i] = cohervp[(i + (irlength / 2)) % irlength];
}

for(i = 0; i < irlength; i++)
{
  filtpadp[i] = filtoutp[i];
  filtupadp[i] = filtuoutp[i];
  filtvpadp[i] = filtvoutp[i];
}
for(i = irlength; i < irlengthpad; i++)
{
  filtpadp[i] = FL(0.0);
  filtupadp[i] = FL(0.0);
  filtvpadp[i] = FL(0.0);
  }

  csound->RealFFT(csound, filtpadp, irlengthpad);
  csound->RealFFT(csound, filtupadp, irlengthpad);
  csound->RealFFT(csound, filtvpadp, irlengthpad);

  T = FL(1.0 / sr);

  gip = (MYFLT *)p->gi.auxp;
  aip = (MYFLT *)p->ai.auxp;

  do
  {
    clipcheck = 0;
    alpha = rt60high / rt60low;
    p->b = FL((1.0 - alpha) / (1.0 + alpha));
    aconst = FL((log(10.0) / 4.0) * (1.0 - (1.0 / SQUARE(alpha))));
    for(i = 0; i < M; i++)
    {
      exp = FL((-3.0 * delaysp[i] * T) / rt60low);
      gip[i] = FL(pow(10.0, exp));
      aip[i] =  exp * aconst;

      if(aip[i] > .99 || aip[i] < -.99)
      {
        csound->Message("\nwarning, approaching instability, fixed with a flat late
                         reverb!");
        clipcheck = 1;
        if(aip[i] > .99)
          rt60high = rt60low;
        else
          rt60low = rt60high;
        break;
      }

    }
  }while(clipcheck);

  /* initialise counter and filter delays */
  p->counter = 0;
  p->inoldl = 0;
  p->inoldr = 0;
  p->M = M;

  return OK;
}

int hrtfreverb_process(CSOUND *csound, hrtfreverb *p)
{
  int i, j, k, n = csound->ksmps;

  /* signals in, out */
  MYFLT *in = p->insig;
  MYFLT *outl = p->outsigl;
  MYFLT *outr = p->outsigr;

  /* pointers to delay data */
  MYFLT *del1p, *del2p, *del3p, *del4p, *del5p, *del6p;
  MYFLT *del1tp, *del2tp, *del3tp, *del4tp, *del5tp, *del6tp;
  MYFLT *del1tfp, *del2tfp, *del3tfp, *del4tfp, *del5tfp, *del6tfp, *del7tfp, *del8tfp,
        *del9tfp, *del10tfp, *del11tfp, *del12tfp;
  int *delaysp;

  /* matrix manipulation */
  MYFLT *inmatp, *inmatlpp, *dellpp, *outmatp;

  /* delay line iterators */
  int u, v, w, x, y, z;
  int ut, vt, wt, xt, yt, zt;
  int utf1, vtf1, wtf1, xtf1, ytf1, ztf1;
  int utf2, vtf2, wtf2, xtf2, ytf2, ztf2;
```

```
/* number of delays */
int M = p->M;

/* FIR temp variables */
MYFLT tonall, tonalr;
MYFLT b = p->b;

/* IIR variables */
MYFLT *gip, *aip;

/* counter */
int counter = p->counter;

/* matrix/coher and hrtf filter buffers, with overlap add buffers */
MYFLT *matrixlup = (MYFLT *)p->matrixlu.auxp;
MYFLT *matrixrvp = (MYFLT *)p->matrixrv.auxp;
MYFLT *olmatrixlup = (MYFLT *)p->olmatrixlu.auxp;
MYFLT *olmatrixrvp = (MYFLT *)p->olmatrixrv.auxp;
MYFLT *hrtflp = (MYFLT *)p->hrtfl.auxp;
MYFLT *hrtfrp = (MYFLT *)p->hrtfr.auxp;
MYFLT *olhrtflp = (MYFLT *)p->olhrtfl.auxp;
MYFLT *olhrtfrp = (MYFLT *)p->olhrtfr.auxp;

/* processing lengths */
int irlength = p->irlength;
int irlengthpad = p->irlengthpad;
int overlapsize = p->overlapsize;

/* 1st order FIR mem */
MYFLT inoldl = p->inoldl;
MYFLT inoldr = p->inoldr;

/* filters, created in INIT */
MYFLT *filtpadp = (MYFLT *)p->filtpad.auxp;
MYFLT *filtupadp = (MYFLT *)p->filtupad.auxp;
MYFLT *filtvpadp = (MYFLT *)p->filtvpad.auxp;

MYFLT sr = p->sr;

del1p = (MYFLT *)p->del1.auxp;
del2p = (MYFLT *)p->del2.auxp;
del3p = (MYFLT *)p->del3.auxp;
del4p = (MYFLT *)p->del4.auxp;
del5p = (MYFLT *)p->del5.auxp;
del6p = (MYFLT *)p->del6.auxp;

if(M==12 || M==24)
{
  del1tp = (MYFLT *)p->del1t.auxp;
  del2tp = (MYFLT *)p->del2t.auxp;
  del3tp = (MYFLT *)p->del3t.auxp;
  del4tp = (MYFLT *)p->del4t.auxp;
  del5tp = (MYFLT *)p->del5t.auxp;
  del6tp = (MYFLT *)p->del6t.auxp;
}
if(M==24)
{
  del1tfp = (MYFLT *)p->del1tf.auxp;
  del2tfp = (MYFLT *)p->del2tf.auxp;
  del3tfp = (MYFLT *)p->del3tf.auxp;
  del4tfp = (MYFLT *)p->del4tf.auxp;
  del5tfp = (MYFLT *)p->del5tf.auxp;
  del6tfp = (MYFLT *)p->del6tf.auxp;
  del7tfp = (MYFLT *)p->del7tf.auxp;
  del8tfp = (MYFLT *)p->del8tf.auxp;
  del9tfp = (MYFLT *)p->del9tf.auxp;
  del10tfp = (MYFLT *)p->del10tf.auxp;
  del11tfp = (MYFLT *)p->del11tf.auxp;
  del12tfp = (MYFLT *)p->del12tf.auxp;
}
```

```
delaysp = (int *)p->delays.auxp;

inmatp = (MYFLT *)p->inmat.auxp;
inmatlpp = (MYFLT *)p->inmatlp.auxp;
dellpp = (MYFLT *)p->dellp.auxp;
outmatp = (MYFLT *)p->outmat.auxp;

gip = (MYFLT *)p->gi.auxp;
aip = (MYFLT *)p->ai.auxp;

/* point to structure */
u = p->u;
v = p->v;
w = p->w;
x = p->x;
y = p->y;
z = p->z;
if(M==12 || M==24)
{
  ut = p->ut;
  vt = p->vt;
  wt = p->wt;
  xt = p->xt;
  yt = p->yt;
  zt = p->zt;
}
if(M==24)
{
  utf1 = p->utf1;
  vtf1 = p->vtf1;
  wtf1 = p->wtf1;
  xtf1 = p->xtf1;
  ytf1 = p->ytf1;
  ztf1 = p->ztf1;
  utf2 = p->utf2;
  vtf2 = p->vtf2;
  wtf2 = p->wtf2;
  xtf2 = p->xtf2;
  ytf2 = p->ytf2;
  ztf2 = p->ztf2;
}

/* processing loop */
for(i=0; i < n; i++)
{
  /* tonal filter: 1 - b pow(z,-1) / 1 - b
  1/1-b in - b/1-b in(old) */
  /* dot product of l and r = 0 for uncorrelated */
  tonall = (del1p[u] - del2p[v] + del3p[w] - del4p[x] + del5p[y] - del6p[z]);
  if(M==12 || M==24)
    tonall += (del1tp[ut] - del2tp[vt] + del3tp[wt] - del4tp[xt] + del5tp[yt] –
               del6tp[zt]);
  if(M==24)
    tonall += (del1tfp[utf1] - del2tfp[vtf1] + del3tfp[wtf1] - del4tfp[xtf1] +
               del5tfp[ytf1] - del6tfp[ztf1] + del7tfp[utf2] - del8tfp[vtf2] +
               del9tfp[wtf2] - del10tfp[xtf2] + del11tfp[ytf2] - del12tfp[ztf2]);
  matrixlup[counter] = FL(((1.0 / (1.0 - b)) * tonall) - ((b / (1.0 - b)) * inoldl));
  matrixlup[counter] /= M;
  inoldl = tonall;

  tonalr = (del1p[u] + del2p[v] + del3p[w] + del4p[x] + del5p[y] + del6p[z]);
  if(M==12 || M==24)
    tonalr += (del1tp[ut] + del2tp[vt] + del3tp[wt] + del4tp[xt] + del5tp[yt] +
               del6tp[zt]);
  if(M==24)
    tonalr += (del1tfp[utf1] - del2tfp[vtf1] + del3tfp[wtf1] - del4tfp[xtf1] +
               del5tfp[ytf1] - del6tfp[ztf1] + del7tfp[utf2] - del8tfp[vtf2] +
               del9tfp[wtf2] - del10tfp[xtf2] + del11tfp[ytf2] - del12tfp[ztf2]);
  matrixrvp[counter] = FL(((1.0 / (1.0 - b)) * tonalr) - ((b / (1.0 - b)) * inoldr));
  matrixrvp[counter] /= M;
```

```
inoldr = tonalr;

/* inputs from del lines (need more for larger fdn) */
inmatp[0] = del1p[u];
inmatp[1] = del2p[v];
inmatp[2] = del3p[w];
inmatp[3] = del4p[x];
inmatp[4] = del5p[y];
inmatp[5] = del6p[z];

if(M==12 || M==24)
{
  inmatp[6] = del1tp[ut];
  inmatp[7] = del2tp[vt];
  inmatp[8] = del3tp[wt];
  inmatp[9] = del4tp[xt];
  inmatp[10] = del5tp[yt];
  inmatp[11] = del6tp[zt];
}
if(M==24)
{
  inmatp[12] = del1tfp[utf1];
  inmatp[13] = del2tfp[vtf1];
  inmatp[14] = del3tfp[wtf1];
  inmatp[15] = del4tfp[xtf1];
  inmatp[16] = del5tfp[ytf1];
  inmatp[17] = del6tfp[ztf1];
  inmatp[18] = del7tfp[utf2];
  inmatp[19] = del8tfp[vtf2];
  inmatp[20] = del9tfp[wtf2];
  inmatp[21] = del10tfp[xtf2];
  inmatp[22] = del11tfp[ytf2];
  inmatp[23] = del12tfp[ztf2];
}

/* low pass each
filter:
gi ( 1 - ai / 1 - ai pow(z,-1) )
 op = gi - gi ai x(n) + ai del
 del = op */

for(j = 0; j < M; j++)
{
  inmatlpp[j] = (gip[j] * (1 - aip[j]) * inmatp[j]) + (aip[j] * dellpp[j]);
  dellpp[j] = inmatlpp[j];
}

/* matrix mult: multiplying a vector by a matrix:
   embedded householders cause stability issues, as reported by Murphy...*/
for(j = 0; j < M; j++)
{
  outmatp[j] = FL(0.0);
  for(k = 0; k < M; k++)
  {
    if(M==24)
      outmatp[j] += (matrix24[j * M + k] * inmatlpp[k]);
    else if(M==12)
      outmatp[j] += (matrix12[j * M + k] * inmatlpp[k]);
    else
      outmatp[j] += (matrix6[j * M + k] * inmatlpp[k]);
  }
}

del1p[u] = outmatp[0] + in[i];
del2p[v] = outmatp[1] + in[i];
del3p[w] = outmatp[2] + in[i];
del4p[x] = outmatp[3] + in[i];
del5p[y] = outmatp[4] + in[i];
del6p[z] = outmatp[5] + in[i];
if(M == 12 || M == 24)
{
  del1tp[ut] = outmatp[6] + in[i];
  del2tp[vt] = outmatp[7] + in[i];
  del3tp[wt] = outmatp[8] + in[i];
  del4tp[xt] = outmatp[9] + in[i];
  del5tp[yt] = outmatp[10] + in[i];
  del6tp[zt] = outmatp[11] + in[i];
}
if(M == 24)
{
  del1tfp[utf1] = outmatp[12] + in[i];
  del2tfp[vtf1] = outmatp[13] + in[i];
  del3tfp[wtf1] = outmatp[14] + in[i];
  del4tfp[xtf1] = outmatp[15] + in[i];
  del5tfp[ytf1] = outmatp[16] + in[i];
  del6tfp[ztf1] = outmatp[17] + in[i];
  del7tfp[utf2] = outmatp[18] + in[i];
  del8tfp[vtf2] = outmatp[19] + in[i];
  del9tfp[wtf2] = outmatp[20] + in[i];
  del10tfp[xtf2] = outmatp[21] + in[i];
  del11tfp[ytf2] = outmatp[22] + in[i];
  del12tfp[ztf2] = outmatp[23] + in[i];
}

u = (u != delaysp[0] - 1 ? u + 1 : 0);
v = (v != delaysp[1] - 1 ? v + 1 : 0);
w = (w != delaysp[2] - 1 ? w + 1 : 0);
x = (x != delaysp[3] - 1 ? x + 1 : 0);
y = (y != delaysp[4] - 1 ? y + 1 : 0);
z = (z != delaysp[5] - 1 ? z + 1 : 0);

if(M == 12 || M == 24)
{
  ut = (ut != delaysp[6] - 1 ? ut + 1 : 0);
  vt = (vt != delaysp[7] - 1 ? vt + 1 : 0);
  wt = (wt != delaysp[8] - 1 ? wt + 1 : 0);
  xt = (xt != delaysp[9] - 1 ? xt + 1 : 0);
  yt = (yt != delaysp[10] - 1 ? yt + 1 : 0);
  zt = (zt != delaysp[11] - 1 ? zt + 1 : 0);
}
if(M == 24)
{
  utf1 = (utf1 != delaysp[12] - 1 ? utf1 + 1 : 0);
  vtf1 = (vtf1 != delaysp[13] - 1 ? vtf1 + 1 : 0);
  wtf1 = (wtf1 != delaysp[14] - 1 ? wtf1 + 1 : 0);
  xtf1 = (xtf1 != delaysp[15] - 1 ? xtf1 + 1 : 0);
  ytf1 = (ytf1 != delaysp[16] - 1 ? ytf1 + 1 : 0);
  ztf1 = (ztf1 != delaysp[17] - 1 ? ztf1 + 1 : 0);
  utf2 = (utf2 != delaysp[18] - 1 ? utf2 + 1 : 0);
  vtf2 = (vtf2 != delaysp[19] - 1 ? vtf2 + 1 : 0);
  wtf2 = (wtf2 != delaysp[20] - 1 ? wtf2 + 1 : 0);
  xtf2 = (xtf2 != delaysp[21] - 1 ? xtf2 + 1 : 0);
  ytf2 = (ytf2 != delaysp[22] - 1 ? ytf2 + 1 : 0);
  ztf2 = (ztf2 != delaysp[23] - 1 ? ztf2 + 1 : 0);
}

/* output, increment counter */
outl[i] = hrtflp[counter];
outr[i] = hrtfrp[counter];

counter++;

if(counter == irlength)
{
  for(j = irlength; j < irlengthpad; j++)
  {
    matrixlup[j] = FL(0.0);
    matrixrvp[j] = FL(0.0);
  }

  /* fft result from matrices */
  csound->RealFFT(csound, matrixlup, irlengthpad);
```

```
csound->RealFFT(csound, matrixrvp, irlengthpad);

/* convolution: spectral multiplication */
csound->RealFFTMult(csound, matrixlup, matrixlup, filtupadp, irlengthpad,
                    (MYFLT)1.0);
csound->RealFFTMult(csound, matrixrvp, matrixrvp, filtvpadp, irlengthpad,
                    (MYFLT)1.0);

/* ifft result */
csound->InverseRealFFT(csound, matrixlup, irlengthpad);
csound->InverseRealFFT(csound, matrixrvp, irlengthpad);

for(j = 0; j < irlength; j++)
{
  matrixlup[j] = matrixlup[j] + (j < overlapsize ? olmatrixlup[j] : FL(1.0));
  matrixrvp[j] = matrixrvp[j] + (j < overlapsize ? olmatrixrvp[j] : FL(1.0));
}

/* store overlap for next time */
for(j = 0; j < overlapsize; j++)
{
  olmatrixlup[j] = matrixlup[j + irlength];
  olmatrixrvp[j] = matrixrvp[j + irlength];
}

/* coherence formula */
for(j = 0; j < irlength; j++)
{
  hrtflp[j] = matrixlup[j] + matrixrvp[j];
  hrtfrp[j] = matrixlup[j] - matrixrvp[j];
}

for(j = irlength; j < irlengthpad; j++)
{
  hrtflp[j] = FL(0.0);
  hrtfrp[j] = FL(0.0);
}

/* fft result from matrices */
csound->RealFFT(csound, hrtflp, irlengthpad);
csound->RealFFT(csound, hrtfrp, irlengthpad);

/* convolution: spectral multiplication */
csound->RealFFTMult(csound, hrtflp, hrtflp, filtpadp, irlengthpad, FL(1.0));
csound->RealFFTMult(csound, hrtfrp, hrtfrp, filtpadp, irlengthpad, FL(1.0));

/* ifft result */
csound->InverseRealFFT(csound, hrtflp, irlengthpad);
csound->InverseRealFFT(csound, hrtfrp, irlengthpad);

/* scale */
for(j = 0; j < irlengthpad; j++)
{
  hrtflp[j] = hrtflp[j]/(sr / FL(38000.0));
  hrtfrp[j] = hrtfrp[j]/(sr / FL(38000.0));
}

for(j = 0; j < irlength; j++)
{
  hrtflp[j] = hrtflp[j] + (j < overlapsize ? olhrtflp[j] : FL(0.0));
  hrtfrp[j] = hrtfrp[j] + (j < overlapsize ? olhrtfrp[j] : FL(0.0));
}

/* store overlap for next time */
for(j = 0; j < overlapsize; j++)
{
  olhrtflp[j] = hrtflp[j + irlength];
  olhrtfrp[j] = hrtfrp[j + irlength];
}

counter = 0;
```

```
    }  /* end of irlength loop */
  }  /* end of ksmps loop */

  /* keep for next time */
  p->counter = counter;

  p->u = u;
  p->v = v;
  p->w = w;
  p->x = x;
  p->y = y;
  p->z = z;
  if(M == 12 || M == 24)
  {
    p->ut = ut;
    p->vt = vt;
    p->wt = wt;
    p->xt = xt;
    p->yt = yt;
    p->zt = zt;
  }
  if(M == 24)
  {
    p->utf1 = utf1;
    p->vtf1 = vtf1;
    p->wtf1 = wtf1;
    p->xtf1 = xtf1;
    p->ytf1 = ytf1;
    p->ztf1 = ztf1;
    p->utf2 = utf2;
    p->vtf2 = vtf2;
    p->wtf2 = wtf2;
    p->xtf2 = xtf2;
    p->ytf2 = ytf2;
    p->ztf2 = ztf2;
  }

  p->inoldl = inoldl;
  p->inoldr = inoldr;

  return OK;
}

static OENTRY localops[] =
{
  {
    "hrtfreverb", sizeof(hrtfreverb), 5, "aai", "aiiSSoop",
    (SUBR)hrtfreverb_init, NULL, (SUBR)hrtfreverb_process
  }
};

LINKAGE
```

Appendix 4: MultiBin

4.1 MultiBin.py

```
#Brian Carty
#PhD Code
#MultiBin, August 2010

import csnd
from Tkinter import *
import tkSimpleDialog
import tkMessageBox
import cmath, math

#inherit frame...class is a frame!
class Application(Frame):

  #global reverb on/off
  reverb = 0
  #how many sources currently active?
  count = 0
  #how many sources removed in current scene
  removed = 0
  #how many sources have been 'clear alled'
  allremoved = 0
  #source instrument on?
  playing = 0

  def move(self, event):
    #widget that called the event
    loc = event.widget
    #set, in case of canvas scroll?...
    x = loc.canvasx(event.x)
    y = loc.canvasy(event.y)
    #find_withtag returns list (tuple) of matching items, in order created:
    #only 1 item will be returned here (or if they are at same loc, most recently
     created)...
    item = loc.find_withtag("current")
    #text only has 2 coords...
    loc.coords(item, x, y)
    #choose value based on default room sizes/inputted size
    #head is item 2, oval 3, rectangle 1, then sources in order created...
    #if > no of items active, subtract no of removed...also subtract total removed...
    if item[0] - 3 - self.allremoved > self.count:
      chno = item[0] - 3 - self.allremoved - self.removed
    else:
      chno = item[0] - 3 - self.allremoved

    self.cs.SetChannel("xsrc%d" %chno, (x / self.sizex) * self.roomarray[0])
    #send inverted y to csound...
    y = self.sizey - y
    self.cs.SetChannel("ysrc%d" %chno, (y / self.sizey) * self.roomarray[1])

  #a simpler move function, as head will always be on same channel
  def movehead(self, event):
    #widget that called the event
    loc = event.widget
    #set, in case of canvas scroll?...
    self.headx = loc.canvasx(event.x)
    #invert y throughout
    self.heady = loc.canvasy(event.y)
    #y = self.sizey - y
    item = loc.find_withtag("current")[0]
    #head coords...include existing measured rotation...
    loc.coords(item, self.headx + self.rot[0].real, self.heady + self.rot[0].imag,
              self.headx + self.rot[1].real, self.heady + self.rot[1].imag,
              self.headx + self.rot[2].real, self.heady + self.rot[2].imag,
              self.headx + self.rot[3].real, self.heady + self.rot[3].imag,
              self.headx + self.rot[4].real, self.heady + self.rot[4].imag,
              self.headx + self.rot[5].real, self.heady + self.rot[5].imag,
              self.headx + self.rot[6].real, self.heady + self.rot[6].imag)
  #move range
  self.canvas.coords(self.oval, self.headx - self.range, self.heady - self.range,
                     self.headx + self.range, self.heady + self.range)
  self.cs.SetChannel("xhead", (self.headx / self.sizex) * self.roomarray[0])
  #send inverted y to csound...
  y = self.sizey - self.heady
  self.cs.SetChannel("yhead", (y / self.sizey) * self.roomarray[1])

def select(self, event):
  loc = event.widget
  item = loc.find_withtag("current")
  #red if selected!
  loc.itemconfig(item, fill = "red")

def deselect(self, event):
  loc = event.widget
  item = loc.find_withtag("current")
  loc.itemconfig(item, fill = "blue")

def play(self):
  self.perf.InputMessage("i1 0 -1")
  self.playing = 1

def stop(self):
  if self.playing:
    self.perf.InputMessage("i-1 0 -1")
    self.playing = 0

def headrotate(self, event):
  degrees = self.rotscale.get()
  self.cs.SetChannel("rot", degrees)
  #rotate polygon...use complex maths here, as it more elegant
  offset = complex(self.headx, self.heady)
  radangle = math.radians(degrees)
  compangle = cmath.exp(radangle * 1j)
  #angle, from centre of non rotated polygon, rotate by point at 0, 0, add offset
   again at end...
  self.rot[0] = compangle * (complex(self.headx + 5, self.heady + 10) - offset)
  self.rot[1] = compangle * (complex(self.headx + 10, self.heady) - offset)
  self.rot[2] = compangle * (complex(self.headx + 5, self.heady - 10) - offset)
  self.rot[3] = compangle * (complex(self.headx, self.heady - 13) - offset)
  self.rot[4] = compangle * (complex(self.headx - 5, self.heady - 10) - offset)
  self.rot[5] = compangle * (complex(self.headx - 10, self.heady) - offset)
  self.rot[6] = compangle * (complex(self.headx - 5, self.heady + 10) - offset)
  #add offset again...
  self.canvas.coords(self.head, self.rot[0].real + self.headx, self.rot[0].imag +
              self.heady, self.rot[1].real + self.headx, self.rot[1].imag + self.heady,
              self.rot[2].real + self.headx, self.rot[2].imag + self.heady,
              self.rot[3].real + self.headx, self.rot[3].imag + self.heady,
              self.rot[4].real + self.headx, self.rot[4].imag + self.heady,
              self.rot[5].real + self.headx, self.rot[5].imag + self.heady,
              self.rot[6].real + self.headx, self.rot[6].imag + self.heady)

def lateamp(self, event):
  vol = self.latescale.get()
  self.cs.SetChannel("lateamp", vol)

def end(self):
  self.perf.Stop()
  self.perf.Join()
  self.master.destroy()

#generic: new speaker
def newspeaker(self, x, y):
  self.count = self.count + 1
```

```
      no = str(self.count)

      self.canvas.create_text(x, y, text = no, fill = 'blue', tags = 'drag')

      #invert y for csound
      y = self.sizey - y

      S = 'i101.{0} 0 -1 {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} \
        {11} {12} {13} {14} {15} {16} {17} {18} {19} {20}' \
        .format(self.count, (x / self.sizex) * self.roomarray[0], (y / self.sizey) *
          self.roomarray[1], self.roomarray[0], self.roomarray[1], self.roomarray[2],
          self.roomarray[3], self.roomarray[4], self.roomarray[5], self.roomarray[6],
          self.roomarray[7], self.roomarray[8], self.roomarray[9], self.roomarray[10],
          self.roomarray[11], self.roomarray[12], self.roomarray[13], self.roomarray[14],
          self.roomarray[15], self.roomarray[16], self.roomarray[17])

      #set channel
      self.cs.SetChannel("xsrc%d" %self.count, (x / self.sizex) * self.roomarray[0])
      self.cs.SetChannel("ysrc%d" %self.count, (y / self.sizey) * self.roomarray[1])
      self.perf.InputMessage(S)

      #turn on global late reverb, if not already on (may have added and removed sources!)
      if self.count == 1:
        if self.reverb == 0:
          self.perf.InputMessage("i102 0 -1")
          self.reverb = 1
          print "reverb"

    def newsrc(self):
      #default add source: in front of listener
      self.src = location(self)
      if self.src.flag:
        mode = self.src.locationdata[0]
        if mode == 1:
          self.newpolar(self.src.locationdata[2], self.src.locationdata[1])
        #simpler scenario...
        elif mode == 2:
          self.srcx = self.src.locationdata[1]
          self.srcy = self.src.locationdata[2]

        #invert y for pixel location
        self.srcy = self.sizey - self.srcy
        #validate location, leave other validation up to csound...wall params...room
         limits...
        #src must be minimum 10 cm from wall, as per csound
        if self.srcx > self.sizex - (.1 / self.roomarray[0]) * self.sizex:
          self.srcx = self.sizex - (.1 / self.roomarray[0]) * self.sizex
        if self.srcx < (.1 / self.roomarray[0]) * self.sizex:
          self.srcx = (.1 / self.roomarray[0]) * self.sizex
        if self.srcy > self.sizey - (.1 / self.roomarray[1]) * self.sizey:
          self.srcy = self.sizey - (.1 / self.roomarray[1]) * self.sizey
        if self.srcy < (.1 / self.roomarray[1]) * self.sizey:
          self.srcy = (.1 / self.roomarray[1]) * self.sizey

        self.newspeaker(self.srcx, self.srcy)

    def newpolar(self, angle, distance):
      radangle = math.radians(angle)
      #angle measured from centre, clockwise...
      #sin gives x coord...cos y...
      #srcx and srcy are temp variables used for each source
      self.srcx = math.sin(radangle)
      self.srcy = math.cos(radangle)
      #radius: same if calculated from x or y params, as per setup
      mult = self.sizex / self.roomarray[0] * distance
      self.srcx *= mult
      self.srcy *= mult
      #relative to listener/centre
      self.srcx += self.sizex / 2
      self.srcy += self.sizey / 2
```

```
#each default setup will call a unique function...
#need different menu for different possible ambi, vbap setups in csound...
#mode 4 ambisonics
def ambi4(self):
  self.clearall()
  #local variables here
  #angles: anticlockwise
  ang = -22.5
  if self.roomarray[0] < self.roomarray[1]:
    dist = self.roomarray[0] / 3
  else:
    dist = self.roomarray[1] / 3
  for i in range(1, 9):
    self.newpolar(ang, dist)
    #invert y for pixel location
    self.srcy = self.sizey - self.srcy
    self.newspeaker(self.srcx, self.srcy)
    ang -= 45

#from manual example...
def vbap8(self):
  self.clearall()
  ang = 15
  if self.roomarray[0] < self.roomarray[1]:
  dist = self.roomarray[0] / 3
  else:
    dist = self.roomarray[1] / 3
  for i in range(1, 9):
    self.newpolar(ang, dist)
    #invert y for pixel location
    self.srcy = self.sizey - self.srcy
    self.newspeaker(self.srcx, self.srcy)
    if i == 4:
      ang += 30
    else:
      ang += 50

#simple, externalised stereo
def stereo(self):
  self.clearall()
  if self.roomarray[0] < self.roomarray[1]:
    dist = self.roomarray[0] / 3
  else:
    dist = self.roomarray[1] / 3
  self.newpolar(30, dist)
  #invert y for pixel location
  self.srcy = self.sizey - self.srcy
  self.newspeaker(self.srcx, self.srcy)
  self.newpolar(-30, dist)
  self.srcy = self.sizey - self.srcy
  self.newspeaker(self.srcx, self.srcy)

def clear(self):
  #last element of array = [-1]
  recent = self.canvas.find_all()[-1]
  self.canvas.delete(recent)
  self.perf.InputMessage("i-101.%d 0 -1" %self.count)
  print "i-101.%d 0 -1 SENT" %self.count

def clearrecent(self):
  if self.count > 0:
    self.clear()
    self.count -= 1
    self.removed += 1

def clearall(self):
  self.stop()
  while self.count > 0:
    self.clear()
    self.count -= 1
    self.allremoved += 1
    self.allremoved += self.removed
    self.removed = 0

  #get parameters...
  def newscene(self):
  #stop playback
    self.stop()
    self.rm = room(self)
    #if not cancel...
    if self.rm.flag:
      self.clearall()
      #turn off reverb
      if self.reverb == 1:
        self.perf.InputMessage("i-102 0 -1")
        self.reverb = 0
        print "reverb off"
      #globals off
      self.perf.InputMessage("i-100 0 -1")

      for i in range(18):
        self.roomarray[i] = self.rm.rmdata[i]

      self.sizey = self.sizex / self.roomarray[0] * self.roomarray[1]
      self.canvas.configure(width = self.sizex, height = self.sizey)
      #useful canvas area...needs to be min .1m from wall
      self.canvas.coords(self.rect, (.1 / self.roomarray[0]) * self.sizex + 1, (.1 /
                          self.roomarray[1]) * self.sizey + 1,
                          self.sizex + 2 - (.1 / self.roomarray[0]) * self.sizex,
                          self.sizey + 2 - (.1 / self.roomarray[1]) * self.sizey)
      self.canvas.itemconfigure(self.rect, fill = "white", outline = "grey")
      #initialise
      self.headx = self.sizex / 2
      self.heady = self.sizey / 2
      #reset rotation
      self.rotscale.set(0)
      #reset amp
      self.latescale.set(0.3)
      #move 'head' to middle
      self.canvas.coords(self.head, self.headx + 5, self.heady + 10, self.headx + 10,
                         self.heady, self.headx + 5, self.heady - 10, self.headx,
                         self.heady - 13, self.headx - 5, self.heady - 10,
                         self.headx - 10, self.heady, self.headx - 5, self.heady + 10)
      #near field range
      self.range = self.sizex / self.roomarray[0] * .45
      self.canvas.coords(self.oval, self.headx - self.range, self.heady - self.range,
                         self.headx + self.range, self.heady + self.range)
      #head back to middle!
      self.cs.SetChannel("xhead", self.roomarray[0] / 2)
      self.cs.SetChannel("yhead", self.roomarray[1] / 2)
      #rotation back to 0
      self.cs.SetChannel("rot", 0)
      self.statusstring = ("x: %.2f, y: %.2f z: %.2f" %(self.roomarray[0],
                           self.roomarray[1], self.roomarray[2]))
      self.status.config(text = self.statusstring)
      self.status.update_idletasks()
      # globals back on
      self.perf.InputMessage("i100 0 -1 %f %f" %(self.roomarray[0] / 2,
                             self.roomarray[1] / 2))

  def about(self):
    tkMessageBox.showwarning("Hi", "MultiBin v 1.0\nAugust 2010\nby Brian Carty\nUses
Csound opcodes hrtfealry & hrtfreverb\nfor help and blurb, see bmcarty.com")

  def __init__(self, master = None):
    master.title("MultiBin")

    Frame.__init__(self, master)

    #csound setup: turn on, wait for input!
    self.cs = csnd.Csound()
    self.cs.Compile("MultiBin.csd")
    self.perf = csnd.CsoundPerformanceThread(self.cs)
    self.perf.Play()

    #create a Menu base
    self.menu = Menu(self)
    #add it
    self.master.config(menu = self.menu)
    #create menu
    self.filemenu = Menu(self.menu)
    #file menu
    self.menu.add_cascade(label = "File", menu = self.filemenu)
    #this choice is required before processing begins!
    self.filemenu.add_command(label = "New Scene(==Restart)", command = self.newscene)
    self.filemenu.add_command(label = "New Source", command = self.newsrc)
    self.filemenu.add_separator()
    self.filemenu.add_command(label = "'Ideal' Stereo", command = self.stereo)
    #ambisonics: layout 4 from bformdec1
    self.filemenu.add_command(label = "Ambi - Octogon", command = self.ambi4)
    self.filemenu.add_command(label = "VBAP - 8 Channel", command = self.vbap8)
    self.filemenu.add_separator()
    #clear
    self.filemenu.add_command(label = "Clear Recent", command = self.clearrecent)
    self.filemenu.add_command(label = "Clear All", command = self.clearall)
    self.filemenu.add_separator()
    self.filemenu.add_command(label = "Exit", command = self.end)
    self.helpmenu = Menu(self.menu)
    self.menu.add_cascade(label = "Help", menu = self.helpmenu)
    self.helpmenu.add_command(label = "About...", command = self.about)

    #button for source on/off, dial for head rotation
    self.playbut = Button(master, text = "Start Playback Instr", command = self.play)
    self.stopbut = Button(master, text = "Stop", command = self.stop)

    #scale widget for head rotation
    self.rotscale = Scale(master, orient = HORIZONTAL, label = "Head Rotation",
                          from_ = -90.0, to = 90.0, length = 120, cursor = "exchange",
                          resolution = .1, command = self.headrotate)

    #scale for output level of late reverb
    self.latescale = Scale(master, orient = HORIZONTAL, label = "Late Amp", from_ = 0,
                          to = .99, length = 120, resolution = .01,
                          command = self.lateamp)

    #status: hints for user
    self.statusstring = "default room: x: 10.00, y: 10.00, z: 3.00"
    self.status = Label(master, text = self.statusstring, relief = SUNKEN, anchor = W)

    #canvas, default size for first run...
    self.canvas = Canvas(master, bg = "grey", width = 400, height = 400)

    #setup grid...
    self.playbut.grid(row = 0, column = 0)
    self.stopbut.grid(row = 0, column = 1)
    self.rotscale.grid(row = 0, column = 2)
    self.latescale.grid(row = 0, column = 3)
    self.canvas.grid(row = 1, column = 0, columnspan = 4)
    self.status.grid(row = 2, column = 0, columnspan = 4, sticky = E + W)

    #set reverb...
    self.latescale.set(0.3)

    #defaults
    #roomarray data order: rmx, rmy, rmz, wlh, wll, wl1, wl2, wl3, flh, fll, fl1, fl2,
     fl3, clh, cll, cl1, cl2, cl3
    self.roomarray = [10.0, 10.0, 10.0, 3.0, .3, .1, .75, .95, .9, .6, .1, .95, .6, .35,
                      .2, .1, 1.0, 1.0, 1.0]

    self.sizex = 400.0
    self.sizey = self.sizex / self.roomarray[0] * self.roomarray[1]
    #useful canvas area...needs to be min .1m from wall
    #canvas goes from 1 - 401, add extra 1 to bottom right corner as rect is contained
```

```
       within this point
      self.rect = self.canvas.create_rectangle((.1 / self.roomarray[0]) * self.sizex + 1,
                             (.1 / self.roomarray[1]) * self.sizey + 1,
                             self.sizex + 2 - (.1 / self.roomarray[0]) * self.sizex,
                             self.sizey + 2 - (.1 / self.roomarray[1]) * self.sizey,
                             fill = "white", outline = "grey")
      #initialise
      self.headx = self.sizex / 2
      self.heady = self.sizey / 2
      #initialise to zero...for rotation
      self.rot = []
      for i in range (7):
        self.rot.append(complex(0, 0))
      #head best as last object, most recent will be selected first!
      self.range = self.sizex / self.roomarray[0] * .45
      self.oval = self.canvas.create_oval(self.headx - self.range, self.heady -
                                          self.range, self.headx + self.range,
                                          self.heady + self.range, outline = "grey")
      #points for polygon of head...
      self.head = self.canvas.create_polygon(self.headx + 5, self.heady + 10,
                                             self.headx + 10, self.heady, self.headx + 5,
                                             self.heady - 10, self.headx, self.heady - 13,
                                             self.headx - 5, self.heady - 10,
                                             self.headx - 10, self.heady, self.headx - 5,
                                             self.heady + 10, fill = "green",
                                             tags = "draghead", outline = "black")
      self.cs.SetChannel("xhead", self.roomarray[0] / 2)
      self.cs.SetChannel("yhead", self.roomarray[1] / 2)
      self.cs.SetChannel("rot", 0)
      self.cs.SetChannel("lateamp", 0.3)
      self.perf.InputMessage("i100 0 -1 %f %f" %(self.roomarray[0] / 2,
                             self.roomarray[1] / 2))
      #link 'drag' to functions
      self.canvas.tag_bind('drag','<B1-Motion>', self.move)
      self.canvas.tag_bind('drag','<ButtonPress>', self.select)
      self.canvas.tag_bind('drag','<ButtonRelease>', self.deselect)
      self.canvas.tag_bind('draghead','<B1-Motion>', self.movehead)
      #closing window also ends csd...
      self.master.protocol("WM_DELETE_WINDOW", self.end)

  #dialog for room creation
  class room(tkSimpleDialog.Dialog):

    flag = 0

    def body(self, master):
      self.complex = IntVar()
      Checkbutton(master, variable = self.complex, text = "Show Complex Params?",
                  command = self.check).grid(row = 0, column = 0)
      Label(master, text = "Room X:").grid(row = 2, column = 0)
      Label(master, text = "Room Y:").grid(row = 3, column = 0)
      Label(master, text = "Room Z:").grid(row = 4, column = 0)

      #empty list
      self.r = []
      for i in range (3):
        self.r.append(Entry(master, width = 10))
      self.r[0].insert(0, "10")
      self.r[1].insert(0, "10")
      self.r[2].insert(0, "3")
      for i in range (3):
        self.r[i].grid(row = i + 2, column = 1)
      Label(master, text = "High Ab Coef:").grid(row = 1, column = 2)
      Label(master, text = "Low Ab Coef:").grid(row = 2, column = 2)
      Label(master, text = "Band 1(cf: 250Hz):").grid(row = 3, column = 2)
      Label(master, text = "Band 2(cf: 1000Hz):").grid(row = 4, column = 2)
      Label(master, text = "Band 3(cf: 4000Hz):").grid(row = 5, column = 2)
      Label(master, text = "Walls").grid(row = 0, column = 3)

      self.w = []
      for i in range (5):
```

```
      self.w.append(Entry(master, width = 10))
    self.w[0].insert(0, ".3")
    self.w[1].insert(0, ".1")
    self.w[2].insert(0, ".75")
    self.w[3].insert(0, ".95")
    self.w[4].insert(0, ".9")
    for i in range (5):
      self.w[i].grid(row = i + 1, column = 3)

    Label(master, text = "Floor").grid(row = 0, column = 4)
    self.f = []
    for i in range (5):
      self.f.append(Entry(master, width = 10))
    self.f[0].insert(0, ".6")
    self.f[1].insert(0, ".1")
    self.f[2].insert(0, ".95")
    self.f[3].insert(0, ".6")
    self.f[4].insert(0, ".35")
    for i in range (5):
      self.f[i].grid(row = i + 1, column = 4)

    Label(master, text = "Ceiling").grid(row = 0, column = 5)
    self.c = []
    for i in range (5):
      self.c.append(Entry(master, width = 10))
    self.c[0].insert(0, ".2")
    self.c[1].insert(0, ".1")
    self.c[2].insert(0, "1.0")
    self.c[3].insert(0, "1.0")
    self.c[4].insert(0, "1.0")
    for i in range (5):
      self.c[i].grid(row = i + 1, column = 5)

    #disable extra parameters by default
    for i in range (5):
      self.w[i].configure(state = DISABLED)
      self.f[i].configure(state = DISABLED)
      self.c[i].configure(state = DISABLED)

    #initial focus
    return self.r[0]

  def apply(self):
    #avoid error on cancel with flag...
    self.flag = 1
    self.rmdata = []
    for i in range (3):
      self.rmdata.append(float(self.r[i].get()))
    for i in range (5):
      self.rmdata.append(float(self.w[i].get()))
    for i in range (5):
      self.rmdata.append(float(self.f[i].get()))
    for i in range (5):
      self.rmdata.append(float(self.c[i].get()))

  def check(self):
    if self.complex.get() == 0:
      for i in range (5):
        self.w[i].configure(state = DISABLED)
        self.f[i].configure(state = DISABLED)
        self.c[i].configure(state = DISABLED)
    else:
      for i in range (5):
        self.w[i].configure(state = NORMAL)
        self.f[i].configure(state = NORMAL)
        self.c[i].configure(state = NORMAL)

class location(tkSimpleDialog.Dialog):

  flag = 0
  def body(self, master):
    self.mode = IntVar()
    self.r1 = Radiobutton(master, text = "Angle, Distance Input", value = 1,
                          variable = self.mode, command = self.rect)
    #need to do this separately to return correct type self.r1...
    self.r1.grid(row = 0, column = 0, columnspan = 2)
    Label(master, text="OR").grid(row = 0, column = 2)
    Radiobutton(master, text = "X, Y Coordinates", value = 2, variable = self.mode,
                command = self.polar).grid(row = 0, column = 3, columnspan = 2)
    Label(master, text="Note: All values must fit on canvas/in room specified, and will
          be truncated accordingly!").grid(row = 1, columnspan = 4)
    Label(master, text="Distance from Centre:").grid(row = 2, column = 0)
    Label(master, text="Angle:").grid(row = 3, column = 0)
    Label(master, text="X:").grid(row = 2, column = 3)
    Label(master, text="Y:").grid(row = 3, column = 3)
    self.r1.select()

    self.e = []
    for i in range (4):
      self.e.append(Entry(master, width = 10))
    self.e[0].insert(0, "1")
    self.e[1].insert(0, "0")
    self.e[2].insert(0, "100")
    self.e[3].insert(0, "100")
    for i in range (2):
      self.e[i].grid(row = i + 2, column = 1)
    for i in range (2):
      self.e[i + 2].grid(row = i + 2, column = 4)

    for i in range (2):
      self.e[i + 2].configure(state = DISABLED)

    #initial focus
    return self.e[0]

  def apply(self):
    self.flag = 1
    dist = float(self.e[0].get())
    angle = float(self.e[1].get())
    x = float(self.e[2].get())
    y = float(self.e[3].get())
    temp = self.mode.get()
    #fill in array...
    if temp == 1:
      self.locationdata = 1, dist, angle
    elif temp == 2:
      self.locationdata = 2, x, y

  def rect(self):
    for i in range (2):
      self.e[i].configure(state = NORMAL)
    for i in range (2):
      self.e[i+ 2].configure(state = DISABLED)

  def polar(self):
    for i in range (2):
      self.e[i].configure(state = DISABLED)
    for i in range (2):
      self.e[i+ 2].configure(state = NORMAL)

app = Application(Tk())
app.mainloop()
```

Appendix 5: Opcode Manual Pages

hrtfmove

hrtfmove - Generates dynamic 3d binaural audio for headphones using magnitude interpolation and phase truncation.

Description

This opcode takes a source signal and spatialises it in the three-dimensional space around a listener by convolving the it with stored head related transfer function (HRTF) based filters.

Syntax

aleft, aright hrtfmove asrc, kAz, kElev, ifilel, ifiler [, imode, ifade, isr]

Initialisation

ifilel - left HRTF spectral data file.
ifiler - right HRTF spectral data file .

Note:
Spectral datafiles (based on the MIT HRTF database) are available in three different sampling rates: 44.1, 48 and 96 kHz and are labelled accordingly. Input and processing sr should match datafile sr. Files should be in the current directory or the SADIR (see Environment Variables).

imode - optional, default 0 for phase truncation, 1 for minimum phase
ifade - optional, number of processing buffers for phase change crossfade (default 8). Legal range is 1-24. A low value is recommended for complex sources (4 or less: a higher value may make the crossfade audible), a higher value (8 or more: a lower value may make the inconsistency when the filter changes phase values audible) for narrowband sources. Does not effect minimum phase processing.

Note:
Occasionally fades may overlap (when unnaturally fast/complex trajectories are requested). In this case, a warning will be printed. Use a smaller crossfade or slightly change trajectory to avoid any possible audible discontinuities that may arise.

isr - optional, default 44.1kHz, legal values: 44100, 48000 and 96000.

Performance

asrc - Input/source signal.
kAz - azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.
kElev - elevation value in degrees. Positive values represent position above horizontal, negative values are positions below horizontal (min -40).

Artefact-free user-defined trajectories are made possible using an interpolation algorithm based on spectral magnitude interpolation and phase truncation. Crossfades are implemented to minimise/eliminate any inconsistencies caused by updating phase values. These crossfades are performed over a user definable number of convolution processing buffers. Complex sources may only need to crossfade over 1 buffer; narrow band sources may need several. The opcode also offers minimum-phase based processing, a more traditional and complex method. In this mode, the HRTF filters used are reduced to minimum-phase representations and the interpolation process then uses the relationship between minimum-phase magnitude and phase spectra. Interaural time difference, which is inherent to the phase truncation process, is reintroduced in the minimum-phase process using variable delay lines.

Example

Here is an example of the hrtfmove opcode.

```
<CsoundSynthesizer>
<CsOptions>
; realtime audio out
; -o dac
; For Non-realtime ouput leave only the line below:
 -o hrtf.wav
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 10
nchnls = 2

gasrc init 0

instr 1        ;a plucked string

  kamp = p4
  kcps = cpspch(p5)
  icps = cpspch(p5)

  a1 pluck kamp, kcps, icps, 0, 1

  gasrc = gasrc + a1

endin

instr 10       ;uses output from instr1 as source

  kaz   linseg 0, p3, 720             ;2 full rotations

  aleft,aright hrtfmove gasrc, kaz, 0, "hrtf-44100-left.dat", "hrtf-44100-right.dat"

  outs  aleft, aright

  gasrc = 0

endin

</CsInstruments>
<CsScore>

; Play Instrument 1: a simple arpeggio
i1 0 .2 15000 8.00
i1 + .2 15000 8.04
i1 + .2 15000 8.07
i1 + .2 15000 8.11
i1 + .2 15000 9.02
i1 + 1.5 15000 8.11
i1 + 1.5 15000 8.07
i1 + 1.5 15000 8.04
i1 + 1.5 15000 8.00
i1 + 1.5 15000 7.09
i1 + 1.5 15000 8.00
```

```
; Play Instrument 10 for 10 seconds.
i10 0 10

</CsScore>
</CsoundSynthesizer>
```

hrtfmove2

hrtfmove2 - Generates dynamic 3D binaural audio for headphones using a Woodworth based spherical-head model with improved low-frequency phase accuracy.

Description

This opcode takes a source signal and spatialises it in the three dimensional space around a listener using head related transfer function (HRTF) based filters.

Syntax

aleft, aright hrtfmove2 asrc, kAz, kElev, ifilel, ifiler [, ioverlap, iradius, isr]

Initialisation

ifilel - left HRTF spectral data file
ifiler - right HRTF spectral data file

Note:
Spectral datafiles (based on the MIT HRTF database) are available in 3 different sampling rates: 44.1, 48 and 96 kHz and are labelled accordingly. Input and processing sr should match datafile sr. Files should be in the current directory or the SADIR (see Environment Variables).

ioverlap - optional, number of overlaps for STFT processing (default 4). See STFT section of manual.
iradius - optional, head radius used for phase spectra calculation in centimetres (default 8.8).
isr - optional, default 44.1kHz, legal values: 44100, 48000 and 96000.

Performance

asrc - Input/source signal.
kAz - azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.
kElev - elevation value in degrees. Positive values represent position above horizontal, negative values are positions below horizontal (min -40).

Artefact-free user-defined trajectories are made possible using an interpolation algorithm based on spectral magnitude interpolation and a derived phase spectrum employing the Woodworth spherical-head model. Accuracy is increased for the data set provided by extracting and applying a frequency-dependent scaling factor to the phase spectra, leading to a more precise low-frequency interaural time difference. Users can control head radius for the phase derivation, allowing a crude level of individualisation. The dynamic source version of the opcode uses a Short Time Fourier Transform algorithm to avoid artefacts caused by derived phase spectra changes. STFT processing means this opcode is more computationally intensive than hrtfmove using phase truncation, but phase is constantly updated by hrtfmove2.

Example

Here is an example of the hrtfmove2 opcode.

```
<CsoundSynthesizer>
<CsOptions>
; Select flags here
; realtime audio out
; -o dac
; For Non-realtime ouput leave only the line below:
 -o hrtf.wav
</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

gasrc init 0

instr 1        ;a plucked string

  kamp = p4
  kcps = cpspch(p5)
  icps = cpspch(p5)

  a1 pluck kamp, kcps, icps, 0, 1

  gasrc = gasrc + a1

endin

instr 10       ;uses output from instr1 as source

  kaz   linseg 0, p3, 720             ;2 full rotations

  aleft,aright hrtfmove2 gasrc, kaz, 0, "hrtf-44100-left.dat", "hrtf-44100-right.dat"

  outs  aleft, aright

  gasrc = 0

endin

</CsInstruments>
<CsScore>

; Play Instrument 1: a simple arpeggio
i1 0 .2 15000 8.00
i1 + .2 15000 8.04
i1 + .2 15000 8.07
i1 + .2 15000 8.11
i1 + .2 15000 9.02
i1 + 1.5 15000 8.11
i1 + 1.5 15000 8.07
i1 + 1.5 15000 8.04
i1 + 1.5 15000 8.00
i1 + 1.5 15000 7.09
i1 + 1.5 15000 8.00

; Play Instrument 10 for 10 seconds.
i10 0 10

</CsScore>
</CsoundSynthesizer>
```

hrtfstat

hrtfstat - Generates static 3D binaural audio for headphones using a Woodworth based spherical-head model with improved low-frequency phase accuracy.

Description

This opcode takes a source signal and spatialises it in the three dimensional space around a listener using head related transfer function (HRTF) based filters. It produces a static output (azimuth and elevation parameters are i-rate), because a static source allows much more efficient processing than hrtfmove and hrtfmove2.

Syntax

aleft, aright hrtfstat asrc, iAz, iElev, ifilel, ifiler [,iradius, isr]

Initialisation

iAz - azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.
iElev - elevation value in degrees. Positive values represent position above horizontal, negative values are positions below horizontal (min -40).
ifilel - left HRTF spectral data file
ifiler - right HRTF spectral data file

Note:
Spectral datafiles (based on the MIT HRTF database) are available in 3 different sampling rates: 44.1, 48 and 96 kHz and are labelled accordingly. Input and processing sr should match datafile sr. Files should be in the current directory or the SADIR (see Environment Variables).

iradius - optional, head radius used for phase spectra calculation in centimetres (default 8.8)
isr - optional (default 44.1kHz). Legal values are 44100, 48000 and 96000.

Performance

asrc - Input/source signal

Accurate static spatialisation is made possible using an interpolation algorithm based on spectral magnitude interpolation and a derived phase employing the Woodworth spherical-head model. Accuracy is increased for the data set provided by extracting and applying a frequency-dependent scaling factor to the phase spectra, leading to a more precise low-frequency interaural time difference. Users can control head radius for the phase derivation, allowing a crude level of individualisation. The static source version of the opcode uses overlap-add convolution (it does not need STFT processing, see hrtfmove2), and is thus considerably more efficient than hrtfmove2 or hrtfmove, but is not designed to generate moving sources.

Example

Here is an example of the hrtfstat opcode.

```
<CsoundSynthesizer>
```

```
<CsOptions>
; Select flags here
; realtime audio out
; -o dac
; For Non-realtime ouput leave only the line below:
-o hrtf.wav

</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

gasrc init 0

instr 1        ;a plucked string

  kamp = p4
  kcps = cpspch(p5)
  icps = cpspch(p5)

  a1 pluck kamp, kcps, icps, 0, 1

  gasrc = gasrc + a1

endin

instr 10       ;uses output from instr1 as source

  aleft,aright hrtfstat gasrc, 90, 0, "hrtf-44100-left.dat","hrtf-44100-right.dat"

  outs  aleft, aright

  gasrc = 0

endin


</CsInstruments>
<CsScore>

; Play Instrument 1: a plucked string
i1 0 2 20000 8.00

; Play Instrument 10 for 2 seconds.
i10 0 2


</CsScore>
</CsoundSynthesizer>
```

hrtfearly

hrtfearly - Generates 3D binaural audio with high-fidelity early reflections in a parametric room using a Phase Truncation algorithm. Although valid as a stand alone opcode, hrtfearly is designed to work with hrtfreverb to provide spatially accurate, dynamic binaural reverberation. A number of sources can be processed dynamically using a number of hrtfearly instances. All can then be processed with one instance of hrtfreverb.

Description

This opcode essentially nests the hrtfmove opcode in an image model for a user-definable shoebox shaped room. A default room can be selected, or advanced room parameters can be used. Room surfaces can be controlled with high and low-frequency absorption coefficients and gain factors of a three-band equaliser.

Syntax

aleft, aright, irt60low, irt60high, imfp hrtfearly asrc, ksrcx, ksrcy, ksrcz, klstnrx, klstnry, klstnrz, ifilel, ifiler, idefroom, [ifade, isr, iorder, ithreed, kheadrot, iroomx, iroomy, iroomz, iwallhigh, iwalllow, iwallgain1, iwallgain2, iwallgain3, ifloorhigh, ifloorlow, ifloorgain1, ifloorgain2, ifloorgain3, iceilinghigh, iceilinglow, iceilinggain1, iceilinggain2, iceilinggain3]

Initialisation

ifilel - left HRTF spectral data file
ifiler - right HRTF spectral data file

Note:
Spectral datafiles (based on the MIT HRTF database) are available in three different sampling rates: 44.1, 48 and 96 kHz and are labelled accordingly. Input and processing sr should match datafile sr. Files should be in the current directory or the SADIR (see Environment Variables).

idefroom – default room, medium (1: 10×10×3), small (2: 3×4×3) or large (3: 20×25×7). Wall details (high coef, low coef, gain1, gain2, gain3): .3, .1, .75, .95, .9. Floor: .6, .1, .95, .6, .35. Ceiling: .2, .1, 1, 1, 1. If any other value is entered for default room (e.g. 0), optional room parameters will be read.

ifade - optional, number of processing buffers for phase change crossfade (default 8). Legal range is 1-24. See hrtfmove.
isr - optional (default 44.1 kHz). Legal values are 44100, 48000 and 96000.
iorder - optional, order of images processed: higher order: more reflections. Defaults to 1, legal range: 0-4.
ithreed - optional, process image sources in three dimensions (1) or two (0: default).
iroomx - optional, x room size in metres, will be read if no valid default room is entered (all below parameters behave similarly). Minimum room size is 2×2×2.
iroomy - optional, y room size.
iroomz - optional, z room size.
iwallhigh - optional, high frequency wall absorption coefficient (all 4 walls are assumed identical). Absorption coefficients will affect reverb time output.
iwalllow - optional, low frequency wall absorption coefficient.
iwallgain1 - optional, gain on filter centred at 250 Hz (all filters have a Q implying 4 octaves)
iwallgain2 - optional, as above, centred on 1000 Hz.
iwallgain3 - optional, as above, centred on 4000 Hz.
ifloorhigh, ifloorlow, ifloorgain1, ifloorgain2, ifloorgain3 - as above for floor.
iceilinghigh, iceilinglow, iceilinggain1, iceilinggain2, iceilinggain3 - as above for ceiling.

Performance

ksrcx – source x location, must be 10 cm inside room. Also, near-field HRTFs not processed, so source will not change spatially within a 45 cm radius of the listener. These restrictions also apply to location parameters below.
ksrcy – source y location.
ksrcz – source z location.
klstnrx, klstnry, klstnrz – listener location, as above.
kheadrot - optional, angular value for head rotation. Positive implies source movement to the right.

asrc - Input/source signal

Output

irt60low - suggested low frequency reverb tme for later binaural reverb.
irt60high - as above, for high frequency.
imfp - mean free path of room, to be used with later reverb.

Example

Here is a simple example of the hrtfearly and hrtfreverb opcodes.

```
<CsoundSynthesizer>
<CsOptions>
; Select flags here
; realtime audio out
; -o dac
; For Non-realtime ouput leave only the line below:
-o hrtf.wav

</CsOptions>
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

gasrc init 0

instr 1        ;a plucked string

  kamp = p4
  kcps = cpspch(p5)
  icps = cpspch(p5)

  a1 pluck kamp, kcps, icps, 0, 1

  gasrc = gasrc + a1

endin

instr 10       ;uses output from instr1 as source

  ;simple path for source
  kx line 2, p3, 9

  ;early reflections: room default 1
  aearlyl,aearlyr, irt60low, irt60high, imfp hrtfearly gasrc, kx, 5, 1, 5, 1, 1,
                             "hrtf-44100-left.dat", "hrtf-44100-right.dat", 1

  ;later reverb, uses outputs from hrtfearly
  arevl, arevr, idel hrtfreverb gasrc, irt60low, irt60high, "hrtf-44100-left.dat",
                                "hrtf-44100-right.dat", 44100, imfp
  ;delayed and scaled
  alatel delay arevl * .1, idel
  alater delay arevr * .1, idel

  outs  aearlyl + alatel, aearlyr + alater

  gasrc = 0

endin


</CsInstruments>
<CsScore>

; Play Instrument 1: a simple arpeggio
i1 0 .2 15000 8.00
i1 + .2 15000 8.04
i1 + .2 15000 8.07
i1 + .2 15000 8.11
i1 + .2 15000 9.02
i1 + 1.5 15000 8.11
i1 + 1.5 15000 8.07
i1 + 1.5 15000 8.04
i1 + 1.5 15000 8.00
i1 + 1.5 15000 7.09
i1 + 1.5 15000 8.00

; Play Instrument 10 for 11 seconds.
i10 0 11

</CsScore>
</CsoundSynthesizer>
```

hrtfreverb

hrtfreverb – A binaural, dynamic FDN based diffuse-field reverberator. The opcode works independently as an efficient, flexible reverberator. It is, however, designed for use with hrtfearly to provide spatially accurate reverberation with user definable source trajectories. Accurate interaural coherence is also provided.

Description

A frequency-dependent, efficient reverberant field is created based on low and high frequency desired reverb times. The opcode is designed to work with hrtfearly, ideally using its outputs as inputs. However, hrtfreverb can be used as a standalone tool. Stability is enforced.

Syntax

aleft, aright, idel hrtfreverb asrc, ilowrt60, ihighrt60, ifilel, ifiler [,isr, imfp, iorder]

Initialisation

ilowrt60 - low frequency reverb time.
Ihighrt60 – high frequency reverb time.
ifilel - left HRTF spectral data file
ifiler - right HRTF spectral data file

Note:
Spectral datafiles (based on the MIT HRTF database) are available in three different sampling rates: 44.1, 48 and 96 kHz and are labelled accordingly. Input and processing sr should match datafile sr. Files should be in the current directory or the SADIR (see Environment Variables).

isr - optional (default 44.1kHz). Legal values are 44100, 48000 and 96000.
imfp - optional, mean free path, defaults to that of a medium room from hrtfearly (.0109). If used with hrtfearly, the mean free path of the room can be used to calculate the appropriate delay for the later reverb. Legal range: the mean free path of the smallest room allowed by hrtfearly (0.003876) – 1.
iorder – optional, order of early reflection processing. If used with hrtfearly, the order of early reflections can be used to calculate the appropriate delay on the later reverb.

Performance

asrc - Input/source signal

Output

idel – if used with hrtfearly, the appropriate delay for the later reverb, based on the room and order of processing.

Example

See the hrtfearly manual page for a simple example of the hrtfearly and hrtfreverb opcodes.